Yii 2.0 決定版ガイド

http://www.yiiframework.com/doc/guide

Qiang Xue,
Alexander Makarov,
Carsten Brandt,
Klimov Paul,
and
many contributors from the Yii community

日本語 translation provided by:
Kazunari Hoshina, @crifff,
Tomoki Morit,a, @jamband,
Atsuhi Sakurai @mocapapa,
Nobuo Kihara, @softark,
Hisateru Tanaka, @tanakahisateru

Contents

Chapter 1

導入

1.1 Yii とは何か

Yii は現代的なウェブ・アプリケーションを迅速に開発するための、高性能な、コンポーネント・ベースの PHP フレームワークです。 Yii という名前 (ィー すなわち `[ji:]` と発音します) は、中国語では「易」であり、「シンプルかつ進化的」であることを意味します。 また **Yes It Is** のアクロニム (頭字語) であると考えることも出来ます。

1.1.1 Yii は何に適しているか

Yii は汎用的なウェブ・プログラミング・フレームワークです。 つまり、あらゆる種類のウェブ・アプリケーションを PHP を使って開発するときに、Yii を使用することが出来ます。 コンポーネント・ベースのアーキテクチャと洗練されたキャッシュ・サポートを有しているため、Yii は大規模なアプリケーション、 たとえば、ポータル、フォーラム、コンテント・マネージメント・システム (CMS)、電子商取引プロジェクト、RESTful ウェブ・サービス、等々を開発するのに特に適しています。

1.1.2 Yii を他のフレームワークと比べると

あなたが既に他のフレームワークに親しんでいる場合は、Yii を比較するとどうなのかを知りたいでしょう。

- ほとんどの PHP フレームワーク同様、Yii は MVC (Model-View-Controller) アーキテクチャ・パターンを実装し、 このパターンに基づいたコードの編成を推進しています。
- Yii は、コードはシンプルかつエレガントに書かれるべきである、という哲学を採用しています。 何らかのデザイン・パターンの厳密な遵守を主目的とする凝りすぎた設計は、Yii が決して試みようとしないものです。

- Yii はフル装備のフレームワークです。 クエリ・ビルダ、リレーショナル・データベースと NoSQL データベース双方のためのアクティブ・レコード、RESTful API 開発サポート、多層構成のキャッシュ・サポート、 等々、検証済みで直ちに使える多数の機能を提供します。
- Yii は極めて拡張性の高いフレームワークです。あなたはコアのコードのほとんど全ての要素をカスタマイズしたり置き換えたりすることが出来ます。 また、Yii の堅固なエクステンション・アーキテクチャを利用して、再配布可能なエクステンションを使用したり開発したりすることも出来ます。
- 高性能であることは常に Yii の主たる目標です。

Yii はワンマン・ショーではありません。Yii は 強力なコア開発チーム[1] および Yii 開発に間断なく貢献してくれるプロフェッショナルの大きなコミュニティーに支えられたプロジェクトです。 Yii 開発チームは、最新のウェブ開発の潮流と、他のフレームワークやプロジェクトに見出される最善のプラクティスと機能を、注意深く見守り続けています。 他のところで見出された最善のプラクティスと機能で最も適切なものは、定期的にコア・フレームワークに組み込まれ、 シンプルかつエレガントなインタフェイスを通じて公開されます。

1.1.3 Yii のバージョン

Yii は現在、利用可能な二つのメジャー・バージョン、すなわち 1.1 と 2.0 を持っています。バージョン 1.1 は古い世代のもので、現在はメンテナンス・モードにあります。 バージョン 2.0 は、最新のテクノロジーとプロトコル、例えば、Composer、PSR、名前空間、トレイトなどを採用して、Yii を完全に書き直したものです。 バージョン 2.0 がこのフレームワークの現世代を表すものであり、今後数年間にわたって主要な開発努力の対象となるものです。 このガイドは主としてバージョン 2.0 について述べます。

1.1.4 必要条件と前提条件

Yii 2.0 は PHP 5.4.0 以上を必要とし、PHP 7 の最新バージョンで最高の力を発揮します。 個々の機能に対する詳細な必要条件は、全ての Yii リリースに含まれている必要条件チェッカを走らせることによって知ることが出来ます。

Yii を使うためには、オブジェクト指向プログラミング (OOP) の基本的な知識が必要です。 なぜなら、Yii は純粋な OOP ベースのフレームワークだからです。 また、Yii 2.0 は 名前空間[2] や トレイト[3] のよう

[1] http://www.yiiframework.com/team/
[2] https://secure.php.net/manual/ja/language.namespaces.php
[3] https://secure.php.net/manual/ja/language.oop5.traits.php

な PHP の最新の機能を利用しています。 これらの概念を理解することは、Yii 2.0 を採用することを一層容易にするでしょう。

1.2 バージョン 1.1 からアップグレードする

Yii フレームワークは 2.0 のために完全に書き直されたため、バージョン 1.1 と 2.0 の間には数多くの違いがあります。 結果として、バージョン 1.1 からのアップグレードは、マイナー・バージョン間でのアップグレードのような些細な仕事ではなくなりました。 このセクションでは、二つのバージョン間の主要な違いを説明します。

もしあなたが以前に Yii 1.1 を使ったことがなければ、このガイドを飛ばして直接に “始めよう“ に進んでも、問題はありません。

Yii 2.0 はこの要約でカバーされているよりも多くの新機能を導入していることに注意してください。 決定版ガイド全体を通読して全ての新機能について学習することを強く推奨します。 おそらく、以前は自分自身で開発する必要があったいくつかの機能が、今ではコア・コードの一部になっていることに気付くでしょう。

1.2.1 インストール

Yii 2.0 は、事実上の標準的 PHP パッケージ管理ソフトである Composer[4] を全面的に採用しています。 コア・フレームワークも、エクステンションも、インストールは Composer を通じて処理されます。 Yii をインストールする のセクションを参照して、Yii 2.0 をインストールする方法を学習してください。 新しいエクステンションを作成したい場合、または既存の 1.1 エクステンションを 2.0 互換のエクステンションに作り直したい場合は、ガイドの エクステンションを作成する のセクションを参照してください。

1.2.2 PHP の必要条件

Yii 2.0 は PHP 5.4 以上を必要とします。PHP 5.4 は、Yii 1.1 によって必要とされていた PHP 5.2 に比べて、非常に大きく改良されています。 この結果として、注意を払うべき言語レベルでの違いが数多くあります。 以下は PHP に関する主要な変更点の要約です。

- 名前空間[5]。
- 無名関数[6]。
- 配列の短縮構文 `要素[.....]` が `array要素(.....)` の代りに使われています。
- 短縮形の echo タグ `<?=` がビュー・ファイルに使われています。PHP 5.4 以降は、この形を使っても安全です。

[4] https://getcomposer.org/
[5] https://secure.php.net/manual/ja/language.namespaces.php
[6] https://secure.php.net/manual/ja/functions.anonymous.php

- SPL のクラスとインタフェイス[7]。
- 遅延静的束縛(Late Static Bindings)[8]。
- 日付と時刻[9]。
- トレイト[10]。
- 国際化(intl)[11]。 Yii 2.0 は国際化の機能をサポートするために `intl` PHP 拡張を利用しています。

1.2.3 名前空間

Yii 2.0 での最も顕著な変更は名前空間の使用です。 ほとんど全てのコア・クラスが、例えば、`yii\web\Request` のように名前空間に属します。 クラス名に “C” の接頭辞はもう使われません。 命名のスキームはディレクトリ構造に従うようになりました。 例えば、`yii\web\Request` は、対応するクラス・ファイルが Yii フレームワーク・フォルダの下の `web/Request.php` であることを示します。

(全てのコア・クラスは、Yii のクラス・ローダのおかげで、そのクラス・ファイルを明示的にインクルードせずに使うことが出来ます。)

1.2.4 コンポーネントとオブジェクト

Yii 2.0 は、1.1 の `CComponent` クラスを二つのクラス、すなわち、`yii\base\BaseObject` と `yii\base\Component` に分割しました。 `BaseObject` クラスは、ゲッターとセッターを通じて オブジェクト・プロパティ を定義することを可能にする、軽量な基底クラスです。 `Component` クラスは `BaseObject` からの拡張であり、 イベント と ビヘイビア をサポートします。

あなたのクラスがイベントやビヘイビアの機能を必要としない場合は、`BaseObject` を基底クラスとして使うことを考慮すべきです。 基本的なデータ構造を表すクラスに対して、通常、このことが当てはまります。

1.2.5 オブジェクトの構成

`BaseObject` クラスはオブジェクトを構成するための統一された方法を導入しています。 `BaseObject` の全ての派生クラスは、コンストラクタが必要な場合には、インスタンスが正しく構成されるように、 コンストラクタを以下のようにして宣言しなければなりません。

```
class MyClass extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
```

[7] https://secure.php.net/manual/ja/book.spl.php
[8] https://secure.php.net/manual/ja/language.oop5.late-static-bindings.php
[9] https://secure.php.net/manual/ja/book.datetime.php
[10] https://secure.php.net/manual/ja/language.oop5.traits.php
[11] https://secure.php.net/manual/ja/book.intl.php

```
    {
        // ... 構成情報が適用される前の初期化処理

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... 構成情報が適用された後の初期化処理
    }
}
```

上記のように、コンストラクタは最後のパラメータとして構成情報の配列を取らなければなりません。 構成情報の配列に含まれる「名前・値」のペアが、コンストラクタの最後でプロパティを構成します。 `init()` メソッドをオーバーライドして、 構成情報が適用された後に行うべき初期化処理を行うことが出来ます。

この規約に従うことによって、構成情報配列を使って新しいオブジェクトを生成して構成することが 出来るようになります。

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

構成情報に関する詳細は、構成情報 のセクションで見ることが出来ます。

1.2.6 イベント

Yii 1 では、イベントは `on` メソッド (例えば、`onBeforeSave`) を定義することによって作成されました。Yii 2 では、どのようなイベント名でも使うことが出来るようになりました。 `trigger()` メソッドを呼んでイベントを発生させます。

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

イベントにハンドラをアタッチするためには、`on()` メソッドを使います。

```
$component->on($eventName, $handler);
// ハンドラをデタッチするためには、以下のようにします。
// $component->off($eventName, $handler);
```

イベントの機能には数多くの改良がなされました。詳細は イベント のセクションを参照してください。

1.2.7 パス・エイリアス

Yii 2.0 は、パス・エイリアスの使用を、ファイル/ディレクトリのパスと URL の両方に広げました。 また、Yii 2.0 では、通常のファイル/ディレクトリのパスや URL と区別するために、エイリアス名は `@` という文字で始まることが要求されるようになりました。 例えば、`@yii` というエイリアスは Yii のインストール・ディレクトリを指します。 パス・エイリアスは Yii のコア・コードのほとんどの場所でサポートされています。 例えば `yii\caching\FileCache::$cachePath` はパス・エイリアスと通常のディレクトリ・パスの両方を受け取ることが出来ます。

パス・エイリアスは、また、クラスの名前空間とも密接に関係しています。 ルートの名前空間に対しては、それぞれ、パス・エイリアスを定義することが推奨されます。 そうすれば、余計な構成をしなくても、Yii のクラス・オートローダを使うことが出来るようになります。 例えば、`@yii` が Yii のインストール・ディレクトリを指しているので、`yii\web\Request` というようなクラスをオートロードすることが出来る訳です。 サード・パーティのライブラリ、例えば Zend フレームワークなどを使う場合にも、そのフレームワークのインストール・ディレクトリを指す `@Zend` というパス・エイリアスを定義することが出来ます。 一旦そうしてしまえば、その Zend フレームワークのライブラリ内のどんなクラスでも、Yii からオートロードすることが出来るようになります。

パス・エイリアスに関する詳細は エイリアス のセクションを参照してください。

1.2.8 ビュー

Yii 2 のビューについての最も顕著な変更は、ビューの中の `$this` という特殊な変数が現在のコントローラやウィジェットを指すものではなくなった、ということです。 今や `$this` は 2.0 で新しく導入された概念である ビュー・オブジェクトを指します。 ビュー・オブジェクトは `yii\web\View` という型であり、MVC パターンのビューの部分を表すものです。 ビューにおいてコントローラやウィジェットにアクセスしたい場合は、`$this->context` を使うことが出来ます。

パーシャル・ビューを別のビューの中でレンダリングするためには、`$this->renderPartial()` ではなく、`$this->render()` を使います。 さらに、`render` の呼び出しは、2.0 では明示的に echo しなくてはなりません。 と言うのは、`render()` メソッドは、レンダリング結果を返すものであり、それを直接に表示するものではないからです。例えば、

```
echo $this->render('_item', ['item' => $item]);
```

PHP を主たるテンプレート言語として使う以外に、Yii 2.0 は人気のある二つのテンプレート・エンジン、Smarty と Twig に対する正式なサポートを備えています。 Prado テンプレート・エンジンはもはやサポートされていません。 これらのテンプレート・エンジンを使うためには、`view` アプリケーション・コンポーネントを構成して `View::$renderers` プロパ

ティをセットする必要があります。 詳細は テンプレート・エンジン のセクションを参照してください。

1.2.9 モデル

Yii 2.0 は 1.1 における CModel と同様な yii\base\Model を基底モデルとして使います。CFormModel というクラスは完全に廃止されました。 Yii 2 では、それの代りに yii\base\Model を拡張して、フォームのモデル・クラスを作成しなければなりません。

Yii 2.0 は サポートされるシナリオを宣言するための scenarios() という新しいメソッドを導入しました。 このメソッドを使って、どのシナリオの下で、ある属性が検証される必要があるか、また、安全とみなされるか否か、などを宣言します。例えば、

```
public function scenarios()
{
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!role'],
    ];
}
```

上記では二つのシナリオ、すなわち、backend と frontend が宣言されています。 backend シナリオでは、email と role の属性が両方とも安全であり、一括代入が可能です。 frontend シナリオでは、email は一括代入が可能ですが、role は不可能です。email と role は、両方とも、規則を使って検証されなければなりません。

rules() メソッドが、Yii 1.1 に引き続き、検証規則を宣言するために使われます。 scenarios() が導入されたことにより、unsafe バリデータが無くなったことに注意してください。

ほとんどの場合、すなわち、rules() メソッドが存在しうるシナリオを完全に指定しており、 そして unsafe な属性を宣言する必要が無い場合であれば、scenarios() をオーバーライドする必要はありません。

モデルについての詳細を学習するためには、モデル のセクションを参照してください。

1.2.10 コントローラ

Yii 2.0 は yii\web\Controller を基底のコントローラ・クラスとして使います。これは Yii 1.1 におけるCController と同様なクラスです。 yii\base\Action がアクション・クラスの基底クラスです。

コントローラに関して、あなたのコードに最も顕著な影響を及ぼす変更点は、 コントローラのアクションは表示したいコンテントを、エコーするのでなく、返さなければならなくなった、ということです。

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
```

```
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

コントローラに関する詳細については コントローラ のセクションを参照してください。

1.2.11 ウィジェット

Yii 2.0 は `yii\base\Widget` を基底のウィジェット・クラスとして使用します。これは Yii 1.1 の `CWidget` と同様なクラスです。

いろんな IDE においてフレームワークに対するより良いサポートを得るために、Yii 2.0 はウィジェットを使うための新しい構文を導入しました。 スタティックなメソッド `begin()`、`end()`、そして `widget()` が導入されました。 以下のようにして使います。

```
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// 表示するためには結果を "echo" しなければならないことに注意
echo Menu::widget(['items' => $items]);

// オブジェクトのプロパティを初期化するための配列を渡す
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... ここにフォームの入力フィールド ...
ActiveForm::end();
```

詳細については ウィジェット のセクションを参照してください。

1.2.12 テーマ

テーマは 2.0 では完全に違う動き方をします。 テーマは、ソースのビュー・ファイル・パスをテーマのビュー・ファイル・パスにマップするパス・マッピング機構に基づくものになりました。 例えば、あるテーマのパス・マップが `['/web/views' => '/web/themes/basic']` である場合、ビュー・ファイル `/web/views/site/index.php` のテーマ版は `/web/themes/basic/site/index.php` になります。 この理由により、テーマはどのようなビュー・ファイルに対してでも適用することが出来るようになりました。 コントローラやウィジェットのコンテキストの外で表示されるビューに対してすら、適用できます。

また、`CThemeManager` コンポーネントはもうありません。 その代りに、`theme` は `view` アプリケーション・コンポーネントの構成可能なプロパティになりました。

詳細については テーマ のセクションを参照してください。

1.2.13 コンソール・アプリケーション

コンソール・アプリケーションは、ウェブ・アプリケーションと同じように、コントローラとして編成されるようになりました。 1.1 における CConsoleCommand と同様に、コンソール・コントローラは yii\console\Controller を拡張したものでなければなりません。

コンソール・コマンドを走らせるためには、yii <route> という構文を使います。 ここで <route> はコントローラのルート (例えば sitemap/index) を表します。 追加の無名の引数は、対応するコントローラのアクション・メソッドに引数として渡されます。 一方、名前付きの引数は、yii\console\Controller::options() での宣言に従って解析されます。

Yii 2.0 はコメント・ブロックからコマンドのヘルプ情報を自動的に生成する機能をサポートしています。

詳細については コンソール・コマンド のセクションを参照してください。

1.2.14 国際化

Yii 2.0 は PECL intl PHP モジュール[12] に賛同して、内蔵の日付フォーマッタと数字フォーマッタの部品を取り除きました。

メッセージは i18n アプリケーション・コンポーネント経由で翻訳されるようになりました。 このコンポーネントは一連のメッセージ・ソースを管理するもので、 メッセージのカテゴリに基づいて異なるメッセージ・ソースを使うことを可能にするものです。

詳細については 国際化 のセクションを参照してください。

1.2.15 アクション・フィルタ

アクション・フィルタはビヘイビアによって実装されるようになりました。新しいカスタム・フィルタを定義するためには、yii\base\ActionFilter を拡張します。 フィルタを使うためには、そのフィルタ・クラスをビヘイビアとしてコントローラにアタッチします。 例えば、yii\filters\AccessControl フィルタを使うためには、コントローラに次のコードを書くことになります。

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

[12] https://pecl.php.net/package/intl

詳細については フィルタ のセクションを参照してください。

1.2.16 アセット

Yii 2.0 は、アセット・バンドル と呼ばれる新しい概念を導入しました。これは、Yii 1.1 にあったスクリプト・パッケージの概念を置き換えるものです。

アセット・バンドルは、あるディレクトリの下に集められた一群のアセット・ファイル (例えば、JavaScript ファイル、CSS ファイル、イメージ・ファイルなど) です。 それぞれのアセット・バンドルは `yii\web\AssetBundle` を拡張したクラスとして表わされます。 アセット・バンドルを `yii\web\AssetBundle::register()` を通じて登録することによって、 そのバンドルに含まれるアセットにウェブ経由でアクセスできるようになります。 Yii 1 とは異なり、バンドルを登録したページは、そのバンドルで指定されている JavaScript と CSS ファイルへの参照を自動的に含むようになります。

詳細については アセット のセクションを参照してください。

1.2.17 ヘルパ

Yii 2.0 はよく使われるスタティックなヘルパ・クラスを数多く導入しました。それには以下のものが含まれます。

- `yii\helpers\Html`
- `yii\helpers\ArrayHelper`
- `yii\helpers\StringHelper`
- `yii\helpers\FileHelper`
- `yii\helpers\Json`

詳細については、ヘルパの概要 のセクションを参照してください。

1.2.18 フォーム

Yii 2.0 は `yii\widgets\ActiveForm` を使ってフォームを作成する際に使用する フィールド の概念を導入しました。 フィールドは、ラベル、インプット、エラー・メッセージ および/または ヒント・テキストを含むコンテナです。 フィールドは `ActiveField` のオブジェクトとして表現されます。 フィールドを使うことによって、以前よりもすっきりとフォームを作成することが出来るようになりました。

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <div class="form-group">
        <?= Html::submitButton('ログイン') ?>
    </div>
<?php yii\widgets\ActiveForm::end(); ?>
```

詳細については フォームを作成する のセクションを参照してください。

1.2.19 クエリ・ビルダ

1.1 においては、クエリの構築が CDbCommand、CDbCriteria、CDbCommandBuilder など、いくつかのクラスに散らばっていました。 Yii 2.0 は DB クエリを Query オブジェクトの形で表現します。 このオブジェクトが舞台裏で QueryBuilder の助けを得て SQL 文に変換されます。 例えば、

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);

$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

何より良いのは、このようなクエリ構築メソッドが アクティブ・レコード を扱う時にも使える、ということです。

詳細については クエリ・ビルダ のセクションを参照してください。

1.2.20 アクティブ・レコード

Yii 2.0 は アクティブ・レコード に数多くの変更を導入しました。 最も顕著な違いは、クエリの構築方法とリレーショナル・クエリの処理の二つです。

1.1 の CDbCriteria クラスは Yii 2 では yii\db\ActiveQuery に置き換えられました。このクラスは yii\db\Query を拡張したものであり、従って全てのクエリ構築メソッドを継承します。 以下のように、yii\db\ActiveRecord::find() を呼んでクエリの構築を開始します。

```
// 全てのアクティブな顧客を読み出し、ID によって並べる
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

リレーションを宣言するために必要なことは、ActiveQuery オブジェクトを返す getter メソッドを定義するだけのことです。 getter によって定義されたプロパティの名前がリレーションの名前を表します。例えば、以下のコードは orders リレーションを宣言するものです (1.1 では relations() という一個の中枢でリレーションを宣言しなければなりませんでした)。

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

こうすることで、$customer->orders という構文によって関連テーブルにある顧客のオーダにアクセスすることが出来るようになります。 また、下記のコードを用いて、カスタマイズしたクエリ条件によるリレーショナル・クエリをその場で実行することも出来ます。

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

リレーションをイーガー・ロードするとき、Yii 2.0 は 1.1 とは異なる動きをします。 具体的に言うと、1.1 では JOIN クエリが生成されて、主レコードと関連レコードの両方がセレクトされていました。 Yii 2.0 では、JOIN を使わずに二つの SQL 文が実行されます。 すなわち、第一の SQL 文が主たるレコードを返し、第二の SQL 文は主レコードのプライマリ・キーを使うフィルタリングによって関連レコードを返します。

多数のレコードを返すクエリを構築するときは、ActiveRecord オブジェクトを返す代りに、asArray() メソッドをチェインすることが出来ます。 そうすると、クエリ結果は配列として返されることになり、レコードの数が多い場合は、必要とされる CPU 時間とメモリを著しく削減することが出来ます。 例えば、

```
$customers = Customer::find()->asArray()->all();
```

もう一つの変更点は、属性のデフォルト値を public なプロパティによって定義することは出来なくなった、ということです。 デフォルト値を定義する必要がある場合は、アクティブ・レコード・クラスの init メソッドの中で設定しなければなりません。

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

1.1 では、アクティブ・レコード・クラスのコンストラクタをオーバーライドすることについて、いくつか問題がありました。バージョン 2.0 では、もう問題はありません。 コンストラクタにパラメータを追加する場合は、yii\db\ActiveRecord::instantiate() をオーバーライドする必要があるかもしれないことに注意してください。

アクティブ・レコードについては、他にも多くの変更と機能強化がなされています。 詳細については アクティブ・レコード のセクションを参照してください。

1.2.21 アクティブ・レコードのビヘイビア

2.0 では基底のビヘイビア・クラス CActiveRecordBehavior を廃止しました。 アクティブ・レコードのビヘイビアを作成したいときは、直接に yii\base\Behavior を拡張しなければなりません。 ビヘイビア・クラスがオーナーの何らかのイベントに反応する必要がある場合は、以下のように events() メソッドをオーバーライドしなければなりません。

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

1.2.22 User と IdentityInterface

1.1 の `CWebUser` クラスは `yii\web\User` に取って換られました。 そして `CUserIdentity` クラスはもうありません。代りに、使い方がもっと単純な `yii\web\IdentityInterface` を実装しなければなりません。 アドバンスト・プロジェクト・テンプレートがそういう例を提供しています。

詳細は 認証、権限付与、そして アドバンスト・プロジェクト・テンプレート[13] のセクションを参照してください。

1.2.23 URL 管理

Yii 2 の URL 管理は 1.1 のそれと似たようなものです。 主な機能強化は、URL 管理がオプションのパラメータをサポートするようになったことです。 例えば、下記のような規則を宣言した場合に、`post/popular` と `post/1/popular` の両方に合致するようになります。 1.1 では、同じ目的を達成するためには、二つの規則を使う必要がありました。

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]
```

詳細については ルーティングと URL 生成 のセクションを参照してください。

[13] https://www.yiiframework.com/extension/yiisoft/yii2-app-advanced/doc/guide

ルートの命名規約における重要な変更は、コントローラとアクションのキャメル・ケースの名前が 各単語をハイフンで分けた小文字の名前になるようになった、という点です。 例えば、`CamelCaseController` のコントローラ ID は `camel-case` となります。 詳細については、コントローラ ID と アクション ID のセクションを参照してください。

1.2.24 Yii 1.1 と 2.x を一緒に使う

Yii 2.0 と一緒に使いたい Yii 1.1 のレガシー・コードを持っている場合は、 Yii 1.1 と 2.0 を一緒に使う のセクションを参照してください。

Chapter 2

始めよう

2.1 何を知っている必要があるか

Yii の学習曲線は他の PHP フレームワークほど急峻ではありませんが、それでも Yii を使い始める前に学習すべき事がいくつかはあります。

2.1.1 PHP

Yii は PHP フレームワークですから、必ず 言語リファレンスを読んで理解する[1] ようにして下さい。 Yii を使って開発するときはオブジェクト指向の流儀でコードを書くことになりますから、必ず、クラスとオブジェクト[2] および 名前空間[3] には慣れ親しんでおいて下さい。

2.1.2 オブジェクト指向プログラミング

オブジェクト指向プログラミングの基礎的な理解が必要です。これに慣れていない場合は、利用できるチュートリアルが沢山ありますので、その中の一つをチェックして下さい。 例えば、tuts+ の中の一つ[4] など。

あなたのアプリケーションが複雑であればあるほど、その複雑性をうまく管理するために、 より高度な OOP 概念を学ぶ必要があることに留意して下さい。

2.1.3 コマンド・ラインと Composer

Yii は業界標準の PHP パッケージ管理ソフトである Composer[5] を広範囲に使用していますので、必ずその ガイド[6] を読んで理解して下さい。 あなたがコマンド・ラインの使用に慣れていないのであれば、今こそ使い

[1] https://secure.php.net/manual/ja/langref.php
[2] https://secure.php.net/manual/ja/language.oop5.basic.php
[3] https://secure.php.net/manual/ja/language.namespaces.php
[4] https://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762
[5] https://getcomposer.org/
[6] https://getcomposer.org/doc/01-basic-usage.md

始めてみるべき時です。 いったん基礎さえ学習してしまえば、二度とコマンド・ラインなしで仕事をしようとは思わなくなりますよ。

2.2 Yii をインストールする

Yii は二つの方法でインストールすることが出来ます。すなわち、Composer[7] を使うか、アーカイブ・ファイルをダウンロードするかです。 前者がお薦めの方法です。と言うのは、一つのコマンドを走らせるだけで、新しい エクステンション をインストールしたり、Yii をアップデートしたりすることが出来るからです。

Yii の標準的なインストールを実行すると、フレームワークとプロジェクト・テンプレートの両方がダウンロードされてインストールされます。 プロジェクト・テンプレートは、いくつかの基本的な機能、例えば、ログインやコンタクト・フォームなどを実装した、動作する Yii アプリケーションです。 そのコードは推奨される方法に従って編成されています。そのため、プロジェクト・テンプレートは、あなたのプロジェクトのための良い開始点としての役割を果たしうるものです。

ここから続くいくつかのセクションにおいては、いわゆる ベーシック・プロジェクト・テンプレート とともに Yii をインストールする方法、 および、このテンプレートの上に新しい機能を実装する方法を説明します。 Yii はもう一つ、アドバンスト・プロジェクト・テンプレート[8] と呼ばれるテンプレートも提供しています。 こちらは、チーム開発環境において多層構造のアプリケーションを開発するときに使用する方が望ましいものです。

> 情報: ベーシック・プロジェクト・テンプレートは、ウェブ・アプリケーションの 90 パーセントを開発するのに適したものです。 アドバンスト・プロジェクト・テンプレートとの主な違いは、コードがどのように編成されているかという点にあります。 あなたが Yii は初めてだという場合は、シンプルでありながら十分な機能を持っているベーシック・プロジェクト・テンプレートに留まることを強く推奨します。

2.2.1 Composer によるインストール

Composer をインストールする

まだ Composer をインストールしていない場合は、getcomposer.org[9] の指示に従ってインストールすることが出来ます。 Linux や Mac OS X では、次のコマンドを実行します。

---

[7] https://getcomposer.org/

[8] https://www.yiiframework.com/extension/yiisoft/yii2-app-advanced/doc/guide

[9] https://getcomposer.org/download/

```
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
```

Windows では、Composer-Setup.exe[10] をダウンロードして実行します。

何か問題が生じたときは、Composer ドキュメントのトラブル・シューティングのセクション[11] を参照してください。 Composer は初めてだという場合は、少なくとも、Composer ドキュメントの 基本的な使い方のセクション[12] も参照することを推奨します。

このガイドでは、composer のコマンドの全ては、あなたが composer を グローバル[13] にインストールし、 `composer` コマンドとして使用できるようにしているものと想定しています。 そうではなく、ローカル・ディレクトリにある `composer.phar` を使おうとする場合は、例に出てくるコマンドをそれに合せて修正しなければなりません。

以前に Composer をインストールしたことがある場合は、確実に最新のバージョンを使うようにしてください。 Composer は `composer self-update` コマンドを実行してアップデートすることが出来ます。

> 補足: Yii のインストールを実行する際に、Composer は大量の情報を Github API から要求する必要が生じます。 リクエストの数は、あなたのアプリケーションが持つ依存の数によりますが、**Github API** レート制限 より大きくなることがあり得ます。 この制限にかかった場合、Composer は Github API アクセス・トークンを取得するために、あなたの Github ログイン認証情報を要求するでしょう。 高速な接続においては、Composer が対処できるよりも早い段階でこの制限にかかることもありますので、 Yii のインストールの前に、このアクセス・トークンを構成することを推奨します。 アクセス・トークンの構成の仕方については、Github API トークンに関する Composer ドキュメント[14] の指示を参照して下さい。

Yii をインストールする

Composer がインストールされたら、ウェブ・アクセス可能なフォルダで下記のコマンドを実行することによって Yii アプリケーション・テンプレートをインストールすることが出来ます。

```
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

このコマンドが `basic` という名前のディレクトリに Yii アプリケーション／テンプレートの最新の安定版をインストールします。 必要なら別のディレクトリ名を選ぶことも出来ます。

[10] https://getcomposer.org/Composer-Setup.exe
[11] https://getcomposer.org/doc/articles/troubleshooting.md
[12] https://getcomposer.org/doc/01-basic-usage.md
[13] https://getcomposer.org/doc/00-intro.md#globally
[14] https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens

情報: `composer create-project` コマンドが失敗するときは、 よくあるエラーについて Composer ドキュメントのトラブル・シューティングのセクション[15] を参照して下さい。 エラーを修正した後は、`basic` ディレクトリの中で `composer update` を実行して、 中断されたインストールを再開することが出来ます。

ヒント: Yii の最新の開発バージョンをインストールしたい場合は、stability option[16] を追加した次のコマンドを代りに使うことが出来ます。

```
composer create-project --prefer-dist --stability=dev yiisoft/
    yii2-app-basic basic
```

開発バージョンは動いているあなたのコードを動かなくするかもしれませんので、本番環境では使うべきでないことに注意してください。

2.2.2 アーカイブ・ファイルからインストールする

アーカイブ・ファイルから Yii をインストールするには、三つの手順を踏みます。

1. yiiframework.com[17] からアーカイブ・ファイルをダウンロードします。
2. ダウンロードしたファイルをウェブ・アクセス可能なフォルダに展開します。
3. `config/web.php` ファイルを編集して、`cookieValidationKey` という構成情報の項目に秘密キーを入力します (Composer を使って Yii をインストールするときは、これは自動的に実行されます)。

```
// !!! 下記にもし空白なら()秘密キーを入力する - これはクッキー検証のため
    に必要
'cookieValidationKey' => '秘密キーをここに入力',
```

2.2.3 他のインストール・オプション

上記のインストール方法の説明は Yii のインストールの仕方を示すものですが、それは同時に、直ちに動作する基本的なウェブ・アプリケーションを作成するものでもあります。 これは、規模の大小に関わらず、ほとんどのプロジェクトを開始するのに良い方法です。 特に、Yii の学習を始めたばかりの場合には、この方法が適しています。

しかし、他のインストール・オプションも利用可能です。

[15] https://getcomposer.org/doc/articles/troubleshooting.md
[16] https://getcomposer.org/doc/04-schema.md#minimum-stability
[17] http://www.yiiframework.com/download/

- コア・フレームワークだけをインストールし、アプリケーション全体を一から構築したい場合は、アプリケーションを一から構築するで説明されている指示に従うことが出来ます。
- もっと洗練された、チーム開発環境により適したアプリケーションから開始したい場合は、 アドバンスト・プロジェクト・テンプレート[18] をインストールすることを考慮することが出来ます。

2.2.4 アセットをインストールする

Yii は、アセット (CSS および JavaScript) ライブラリのインストールについて Bower[19] および/または NPM[20] のパッケージに依存しています。 Yii はこれらのライブラリを取得するのに Composer を使って、PHP と CSS/JavaScript のパッケージ・バージョンを同時に解決できるようにしています。 このことは、asset-packagist.org[21] または composer asset plugin[22] を使用することによって達成されます。 詳細は アセットのドキュメント を参照して下さい。

あなたは、アセットの管理をネイティブの Bower/NPM クライアントで行ったり、CND を使ったり、アセットのインストールを完全に避けたりしたいかも知れません。 Composer によるアセットのインストールを抑止するためには、`composer.json` に次の記述を追加して下さい。

```
"replace": {
    "bower-asset/jquery": ">=1.11.0",
    "bower-asset/inputmask": ">=3.2.0",
    "bower-asset/punycode": ">=1.3.0",
    "bower-asset/yii2-pjax": ">=2.0.0"
},
```

> 補足: Composer によるアセットのインストールをバイパスする場合は、アセットのインストールとバージョン衝突の解決についてあなたが責任を持たなければなりません。 さまざまなエクステンションに由来するアセット・ファイル間で不整合が生じうることを覚悟して下さい。

2.2.5 インストールを検証する

インストール完了後、あなたのウェブ・サーバを構成してください (次のセクションを参照してください)。 あるいは、プロジェクトの `web` ディレクトリで次のコマンドを実行して、 PHP の内蔵ウェブ・サーバ[23] を使ってください。

[18] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/README.md
[19] http://bower.io/
[20] https://www.npmjs.org/
[21] https://asset-packagist.org
[22] https://github.com/francoispluchino/composer-asset-plugin/
[23] https://secure.php.net/manual/ja/features.commandline.webserver.php

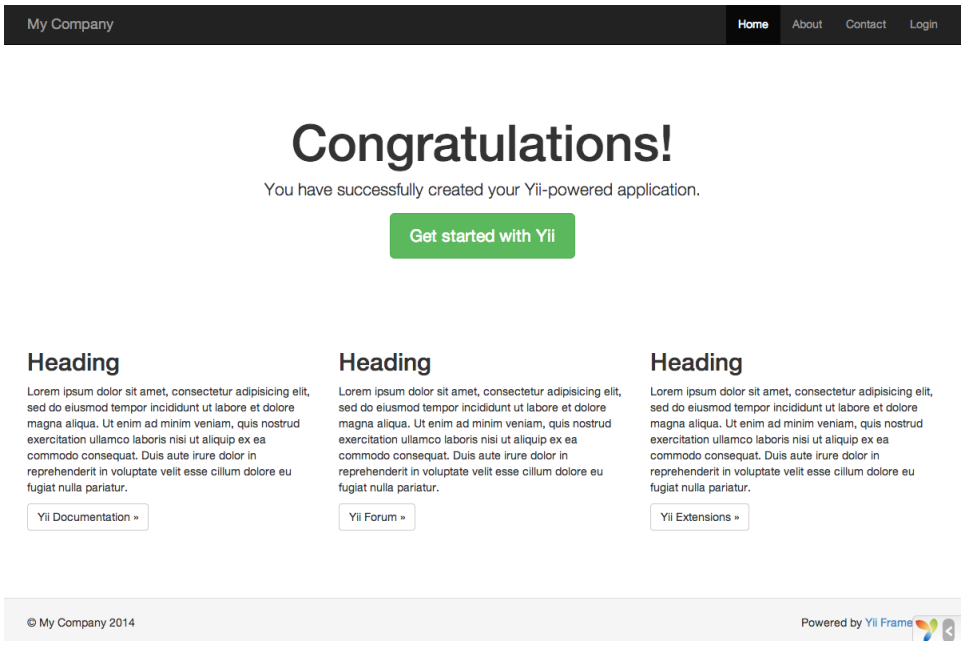
```
php yii serve
```

> 補足: デフォルトでは、この HTTP サーバは 8080 ポートをリスンします。しかし、このポートがすでに使われていたり、複数のアプリケーションをこの方法で動かしたい場合は、 どのポートを使うかを指定したいと思うでしょう。単に –port 引数を追加して下さい。

```
php yii serve --port=8888
```

下記の URL によって、インストールされた Yii アプリケーションにブラウザを使ってアクセスすることが出来ます。

```
http://localhost:8080/
```



ブラウザに上のような “おめでとう!“ のページが表示されるはずです。もし表示されなかったら、PHP のインストールが Yii の必要条件を満たしているかどうか、チェックしてください。 最低限の必要条件を満たしているかどうかは、次の方法のどちらかによってチェックすることが出来ます。

- `/requirements.php` を `/web/requirements.php` としてコピーし、ブラウザを使って URL `http://localhost/requirements.php` にアクセスする。
- 次のコマンドを実行する。

```
cd basic
php requirements.php
```

Yii の最低必要条件を満たすように PHP のインストールを構成しなければなりません。 最も重要なことは、PHP 5.4 以上でなければならないと

いうことです。最新の PHP 7 なら理想的です。 また、アプリケーションがデータベースを必要とする場合は、PDO PHP 拡張[24] および対応するデータベース・ドライバ (MySQL データベースのための `pdo_mysql` など) をインストールしなければなりません。

2.2.6 ウェブ・サーバを構成する

> 情報: もし Yii の試運転をしているだけで、本番サーバに配備する意図がないのであれば、 当面、この項は飛ばしても構いません。

上記の説明に従ってインストールされたアプリケーションは、Apache HTTP サーバ[25] と Nginx HTTP サーバ[26] のどちらでも、また、Windows、Mac OS X、Linux のどれでも、 PHP 5.4 以上を走らせている環境であれば、そのままの状態で動作するはずです。 Yii 2.0 は、また、facebook の HHVM[27] とも互換性があります。 ただし HHVM がネイティブの PHP とは異なる振舞いをする特殊なケースもいくつかありますので、HHVM を使うときはいくらか余分に注意を払う必要があります。

本番用のサーバでは、`http://www.example.com/basic/web/index.php` の代りに `http://www.example.com/index.php` という URL でアプリケーションにアクセス出来るようにウェブ・サーバを設定したいでしょう。 そういう設定をするためには、ウェブ・サーバのドキュメント・ルートを `basic/web` フォルダに向けることが必要になります。 また、ルーティングと URL 生成 のセクションで述べられているように、URL から `index.php` を隠したいとも思うでしょう。 この項では、これらの目的を達するために Apache または Nginx サーバをどのように設定すれば良いかを学びます。

> 情報: `basic/web` をドキュメント・ルートに設定することは、`basic/web` の兄弟ディレクトリに保存されたプライベートなアプリケーション・コードや 公開できないデータ・ファイルにエンド・ユーザがアクセスすることを防止することにもなります。 `basic/web` 以外のフォルダに対するアクセスを拒否することはセキュリティ強化の一つです。

> 情報: あなたがウェブ・サーバの設定を修正する権限を持たない共用ホスティング環境でアプリケーションが走る場合であっても、 セキュリティ強化のためにアプリケーションの構造を調整することがまだ出来ます。 詳細については、共有ホスティング環境 のセクションを参照してください。

[24] https://secure.php.net/manual/ja/pdo.installation.php
[25] http://httpd.apache.org/
[26] http://nginx.org/
[27] http://hhvm.com/

> 情報: あなたのアプリケーションをリバース・プロキシの背後で動かそうとする場合は、 リクエスト・コンポーネントの 信頼できるプロキシとヘッダ を構成する必要があるかもしれません。

推奨される Apache の構成

下記の設定を Apache の `httpd.conf` ファイルまたはバーチャル・ホスト設定の中で使います。 `path/to/basic/web` の部分を `basic/web` の実際のパスに置き換えなければならないことに注意してください。

```
# ドキュメント・ルートを "basic/web" に設定
DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">
    # 綺麗な URL をサポートするために mod_rewrite を使う
    RewriteEngine on

    # UrlManager の $showScriptName が false の場合は、スクリプト名で URL にアクセスすることを許さない
    RewriteRule ^index.php/ - [L,R=404]

    # ディレクトリかファイルが存在する場合は、リクエストをそのまま通す
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d

    # そうでなければ、リクエストを index.php に送付する
    RewriteRule . index.php

    # ... 他の設定 ...
</Directory>
```

推奨される Nginx の構成

Nginx[28] を使うためには、PHP を FPM SAPI[29] としてインストールしなければなりません。 下記の Nginx の設定を使うことができます。 `path/to/basic/web` の部分を `basic/web` の実際のパスに置き換え、`mysite.test` を実際のサーバのホスト名に置き換えてください。

```
server {
    charset utf-8;
    client_max_body_size 128M;

    listen 80; ## listen for ipv4
    #listen [::]:80 default_server ipv6only=on; ## listen for ipv6

    server_name mysite.test;
    root        /path/to/basic/web;
```

[28] http://wiki.nginx.org/
[29] https://secure.php.net/manual/ja/install.fpm.php

```
    index       index.php;

    access_log  /path/to/basic/log/access.log;
    error_log   /path/to/basic/log/error.log;

    location / {
        # 本当のファイルでないものは全て index.php にリダイレクト
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # 存在しない静的ファイルの呼び出しを Yii に処理させたくない場合はコメントを
    外す
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    # /assets ディレクトリの php ファイルへのアクセスを拒否する
    location ~ ^/assets/.*\.php$ {
        deny all;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
        try_files $uri =404;
    }

    location ~* /\. {
        deny all;
    }
}
```

この構成を使う場合は、多数の不要な `stat()` システム・コールを避けるために、 `php.ini` ファイルで `cgi.fix_pathinfo=0` を同時に設定しておくべきです。

また、HTTPS サーバを走らせている場合には、安全な接続であることを Yii が正しく検知できるように、 `fastcgi_param HTTPS on;` を追加しなければならないことにも注意を払ってください。

推奨される NGINX Unit の構成

NGINX Unit[30] と PHP 言語モジュールを使って Yii ベースのアプリを走らせることが出来ます。 その構成のサンプルです。

```
{
    "listeners": {
        "*:80": {
```

[30] https://unit.nginx.org/

```
            "pass": "routes/yii"
        }
    },

    "routes": {
        "yii": [
            {
                "match": {
                    "uri": [
                        "!/assets/*",
                        "*.php",
                        "*.php/*"
                    ]
                },

                "action": {
                    "pass": "applications/yii/direct"
                }
            },
            {
                "action": {
                    "share": "/path/to/app/web/",
                    "fallback": {
                        "pass": "applications/yii/index"
                    }
                }
            }
        ]
    },

    "applications": {
        "yii": {
            "type": "php",
            "user": "www-data",
            "targets": {
                "direct": {
                    "root": "/path/to/app/web/"
                },

                "index": {
                    "root": "/path/to/app/web/",
                    "script": "index.php"
                }
            }
        }
    }
}
```

また、自分の PHP 環境を セットアップ[31] したり、この同じ構成でカスタマイズした `php.ini` を提供したりすることも出来ます。

[31] https://unit.nginx.org/configuration/#php

IIS の構成

ドキュメント・ルートが `path/to/app/web` フォルダを指し、PHP を実行するように構成された仮想ホスト (ウェブ・サイト) でアプリケーションをホストすることを推奨します。その `web` フォルダに `web.config` という名前のファイル、すなわち `path/to/app/web/web.config` を配置しなければなりません。ファイルの内容は以下の通りです。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
<system.webServer>
<directoryBrowse enabled="false" />
  <rewrite>
    <rules>
      <rule name="Hide Yii Index" stopProcessing="true">
        <match url="." ignoreCase="false" />
        <conditions>
        <add input="{REQUEST_FILENAME}" matchType="IsFile"
              ignoreCase="false" negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory"
              ignoreCase="false" negate="true" />
        </conditions>
        <action type="Rewrite" url="index.php" appendQueryString="true" />
      </rule>
    </rules>
  </rewrite>
</system.webServer>
</configuration>
```

また、IIS 上で PHP を構成するためには、以下にリストした Microsoft の公式リソースが有用でしょう。

1. IIS の最初の Web サイトを構成する方法[32]
2. Configure a PHP Website on IIS[33]

2.3 アプリケーションを走らせる

Yii のインストールが終ると、実際に動く Yii のアプリケーションにアクセスすることが出来ます。 その URL は、`http://hostname/basic/web/index.php` あるいは `http://hostname/index.php` など、設定によって異なります。このセクションでは、アプリケーションに組み込み済みの機能を紹介し、コードがどのように編成されているか、 そして、一般にアプリケーションがリクエストをどのように処理するかを説明します。

[32] https://support.microsoft.com/ja-jp/help/323972/how-to-set-up-your-first-iis-web-site

[33] https://docs.microsoft.com/en-us/iis/application-frameworks/scenario-build-a-php-website-on-iis/configure-a-php-website-on-iis

> 情報: 話を簡単にするために、この「始めよう」のチュートリアルを通じて、`basic/web` をウェブ・サーバのドキュメント・ルートとして設定したと仮定します。 そして、アプリケーションにアクセスするための URL は `http://hostname/index.php` またはそれに似たものになるように設定したと仮定します。 必要に応じて、説明の中の URL を読み替えてください。

フレームワークそのものとは異なり、プロジェクト・テンプレートはインストール後は完全にあなたのものであることに注意してください。 必要に応じてコードを追加したり削除したり、完全に書き換えたりするのはあなたの自由です。

2.3.1 機能

インストールされたベーシック・アプリケーションは四つのページを持っています。

- ホームページ: `http://hostname/index.php` の URL にアクセスすると表示されます。
- 「について」のページ。
- 「コンタクト」のページ: エンド・ユーザがメールであなたに連絡を取ることが出来るコンタクト・フォームが表示されます。
- 「ログイン」ページ: エンド・ユーザを認証するためのログイン・フォームが表示されます。 “admin/admin” でログインしてみてください。「ログイン」のメイン・メニュー項目が「ログアウト」に変ることに気付くでしょう。

これらのページは共通のヘッダとフッタを持っています。 ヘッダには、異なるページ間を行き来することを可能にするメイン・メニュー・バーがあります。

ブラウザのウィンドウの下部にツールバーがあることにも気がつくはずです。 これは Yii によって提供される便利な デバッグ・ツール[34] であり、たくさんのデバッグ情報、例えば、ログ・メッセージ、レスポンスのステータス、実行されたデータベース・クエリなどを記録して表示するものです。

ウェブ・アプリケーションに加えて、`yii` というコンソール・スクリプトがアプリケーションのベース・ディレクトリにあります。 このスクリプトは、バックグラウンドのタスクまたはメンテナンスのタスクを実行するために使用することが出来ます。 これについては、コンソール・アプリケーションのセクション で説明されています。

2.3.2 アプリケーションの構造

アプリケーションにとって最も重要なディレクトリとファイルは (アプリ

[34] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide-ja/README.md

ケーションのルート・ディレクトリが basic だと仮定すると) 以下の通りです。

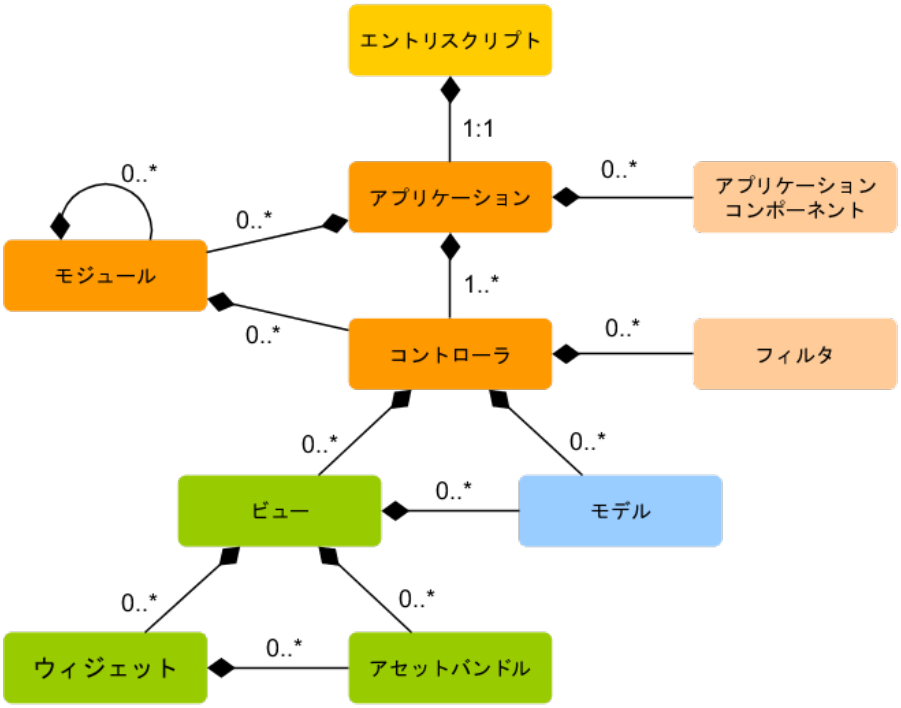
basic/	アプリケーションのベース・パス
composer.json	Composer によって使用される。パッケージ情報を記述
config/	アプリケーションその他の構成情報を格納
console.php	コンソール・アプリケーションの構成情報
web.php	ウェブ・アプリケーションの構成情報
commands/	コンソール・コマンドのクラスを格納
controllers/	コントローラのクラスを格納
models/	モデルのクラスを格納
runtime/	実行時に Yii によって生成されるファイル ログやキャッシュなど() を格納
vendor/	インストールされた Composer パッケージ (Yii フレームワークそのものを含む) を格納
views/	ビュー・ファイルを格納
web/	アプリケーションのウェブ・ルート。ウェブ・アクセス可能なファイルを格納
assets/	Yii によって発行されるアセット・ファイル (javascript と CSS) を格納
index.php	アプリケーションのエントリ・スクリプト ブートストラップ・スクリプト()
yii	Yii コンソール・コマンド実行スクリプト

一般に、アプリケーションのファイルは二種類に分けることが出来ます。すなわち、basic/web の下にあるファイルとその他のディレクトリの下にあるファイルです。 前者は HTTP で (すなわちブラウザで) 直接にアクセスすることが出来ますが、後者は直接のアクセスは出来ませんし、許可すべきでもありません。

Yii は モデル・ビュー・コントローラ (MVC)[35] アーキテクチャ・パターンを実装していますが、 それが上記のディレクトリ構成にも反映されています。 models ディレクトリが全ての モデル・クラス を格納し、views ディレクトリが全ての ビュー・スクリプト を格納し、controllers ディレクトリが全ての コントローラ・クラス を格納しています。

次の図がアプリケーションの静的な構造を示すものです。

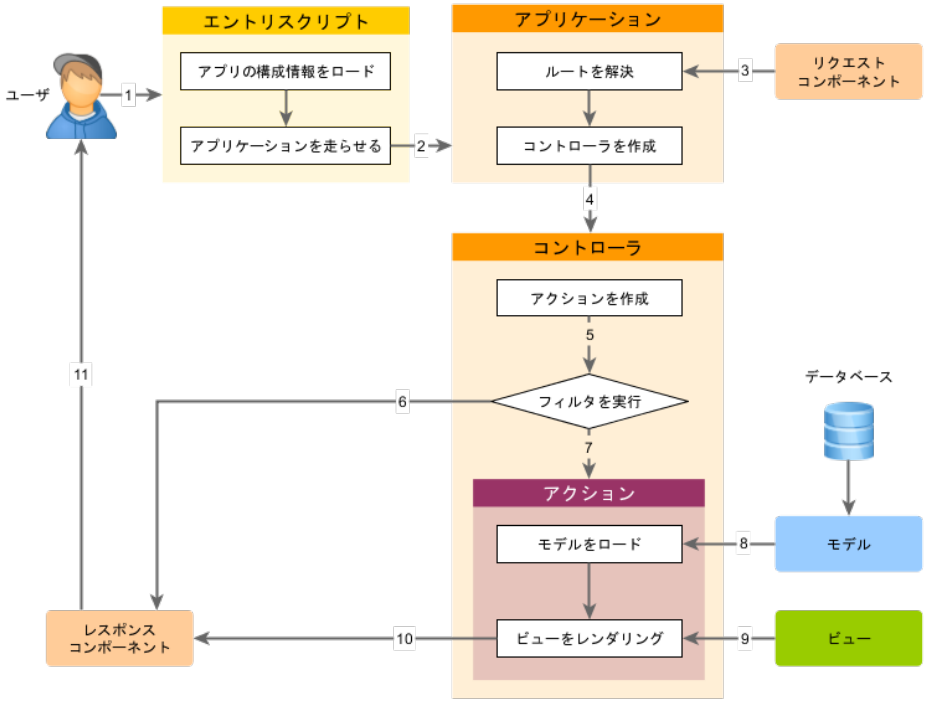
[35] http://wikipedia.org/wiki/Model-view-controller



各アプリケーションは一つのエントリ・スクリプト `web/index.php` を持ちます。 これはアプリケーション中で唯一ウェブ・アクセス可能な PHP スクリプトです。 エントリ・スクリプトは入力されたリクエストを受け取って、アプリケーション のインスタンスを作成します。 アプリケーション は コンポーネント の助力を得てリクエストを解決し、リクエストを MVC に送付します。 ウィジェット は、複雑で動的なユーザ・インタフェイス要素を構築するために、ビュー の中で使われます。

2.3.3 リクエストのライフサイクル

次の図は、アプリケーションがどのようにリクエストを処理するかを示すものです。



1. ユーザが エントリ・スクリプト `web/index.php` に対してリクエストを出します。
2. エントリ・スクリプトはアプリケーションの 構成情報 を読み出して、 リクエストを処理する アプリケーション のインスタンスを作成します。
3. アプリケーションは、リクエスト アプリケーション・コンポーネントの助力を得て、 リクエストされた ルート を解決します。
4. アプリケーションがリクエストを処理する コントローラ のインスタンスを作成します。
5. コントローラが アクション のインスタンスを作成し、アクションのためのフィルタを実行します。
6. 一つでもフィルタが失敗したときは、アクションはキャンセルされます。
7. すべてのフィルタを通ったとき、アクションが実行されます。
8. アクションはデータ・モデルを、おそらくはデータベースから、読み出します。
9. アクションはデータ・モデルをビューに提供して、ビューをレンダリングします。

10. レンダリング結果が レスポンス アプリケーション・コンポーネントに返されます。
11. レスポンス・コンポーネントがレンダリング結果をユーザのブラウザに送信します。

2.4 こんにちは、と言う

このセクションでは、アプリケーションに「こんにちは」という新しいページを作成する方法を説明します。 この目的を達するために、アクション と ビュー を作成します。

- アプリケーションは、このページへのリクエストをそのアクションに送付します。
- 次にそのアクションが「こんにちは」という言葉をエンド・ユーザに示すビューを表示します。

このチュートリアルを通じて、三つのことを学びます。

1. リクエストに応える アクション を作成する方法
2. レスポンスのコンテントを作成する ビュー を作成する方法
3. アプリケーションがリクエストを アクション に送付する仕組み

2.4.1 アクションを作成する

「こんにちは」のタスクのために、リクエストから `message` パラメータを読んで、そのメッセージをユーザに表示して返す `say` アクション を作ります。 リクエストが `message` パラメータを提供しなかった場合は、アクションはデフォルト値として “こんにちは” というメッセージを表示するものとします。

> 情報: アクション は、エンド・ユーザが直接に参照して実行できるオブジェクトです。 アクションは コントローラ によってグループ化されます。 アクションの実行結果が、エンド・ユーザが受け取るレスポンスです。

アクションは コントローラ の中で宣言されなければなりません。 話を簡単にするために、`say` アクションを既存の `SiteController` の中で宣言しましょう。 このコントローラは `controllers/SiteController.php` というクラス・ファイルの中で定義されています。次のようにして、新しいアクションが始まります。

```
<?php

namespace app\controllers;
```

```
use yii\web\Controller;

class SiteController extends Controller
{
    // ... 既存のコード ...

    public function actionSay($message = 'こんにちは')
    {
        return $this->render('say', ['message' => $message]);
    }
}
```

上記のコードでは、SiteController クラスの中で、say アクションが actionSay という名前のメソッドとして定義されています。 Yii はコントローラ・クラスの中で、アクション・メソッドと非アクション・メソッドを区別するために、action という接頭辞を使います。 action という接頭辞の後に続く名前がアクション ID にマップされます。

アクションを命名するについては、Yii がアクション ID をどのように取り扱うかを知っていなければなりません。 アクション ID は常に小文字で参照されます。 アクション ID が複数の単語を必要とするときは、単語がダッシュ (-) で連結されます (例えば、create-comment)。 アクション・メソッドの名前は、アクション ID からダッシュを全て削除し、各単語の先頭の文字を大文字にした結果に action という接頭辞を付けたものになります。 例えば、アクション ID create-comment はアクション・メソッド名 actionCreateComment に対応します。

私たちの例では、アクション・メソッドは $message というパラメータを取り、そのデフォルト値は "こんにちは" です (PHP で関数やメソッドの引数にデフォルト値を設定するのと全く同じ方法です)。 アプリケーションがリクエストを受け取って、当該リクエストの処理を say アクションが担当すべきであると決定した場合は、 リクエストの中に見つかった同じ名前のパラメータの値をこの $message パラメータに代入します。 言い換えれば、もしリクエストの中に "さようなら" という値の message パラメータが入っていれば、アクションの $message 変数にその値が割り当てられます。

アクション・メソッドの中では、render() が呼ばれて say と言う名前の ビュー ファイルがレンダリングされます。 message パラメータも同時にビューに渡され、そこで使用されます。 レンダリング結果はアクション・メソッドによって返されます。 返された結果はアプリケーションによって受け取られ、ブラウザ上でエンド・ユーザに (完全な HTML ページの一部として) 表示されます。

2.4.2 ビューを作成する

ビュー は、レスポンスのコンテントを生成するために書かれるスクリプトです。 「こんにちは」のタスクのためには、アクション・メソッドから受け取った message パラメータを出力する say ビューを作成します。

```
<?php
use yii\helpers\Html;
?>
<?= Html::encode($message) ?>
```

say ビューは views/site/say.php というファイルに保存しなければなりません。 アクションの中で render() メソッドが呼ばれるとき、render() メソッドは views/ControllerID/ViewName.php という名前の PHP ファイルを探します。

上記のコードで message パラメータが出力される前に HTML-エンコード されていることに注意してください。 パラメータはエンド・ユーザから来るものであり、悪意のある JavaScript コードを埋め込まれて クロス・サイト・スクリプティング (XSS) 攻撃[36] に使われうるものですから、 脆弱性を防止するためにこうすることが必要です。

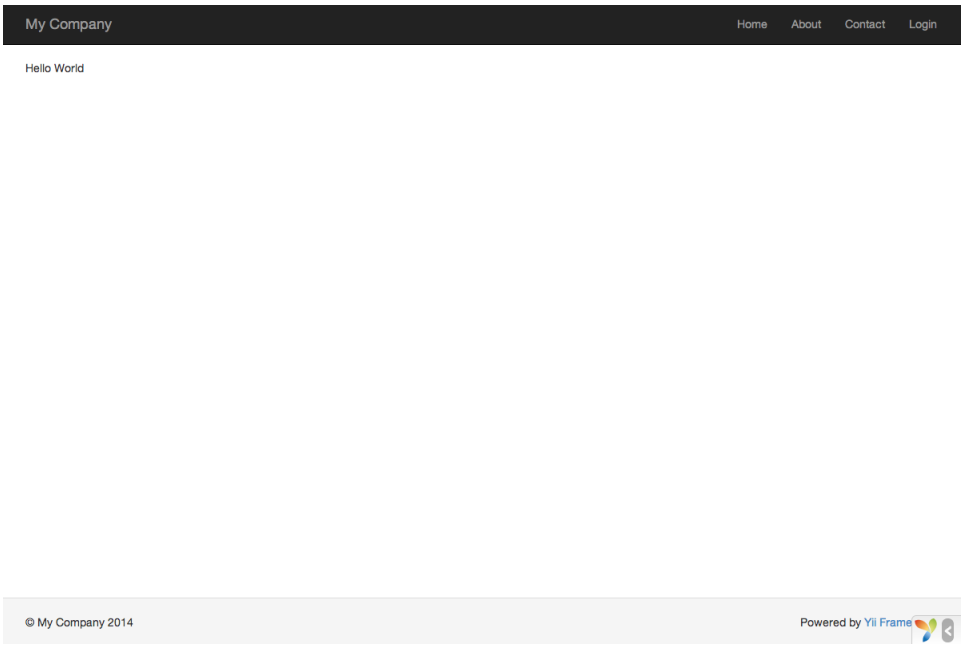
当然ながら、say ビューにはもっと多くのコンテントを入れても構いません。コンテントには、HTML タグ、平文テキスト、さらには PHP 文を含めることが出来ます。 実際、say ビューは render() メソッドによって実行される PHP スクリプトであるに過ぎません。 ビュー・スクリプトによって出力されたコンテントはレスポンス結果としてアプリケーションに返されます。そしてアプリケーションがこの結果をエンド・ユーザに対して出力します。

2.4.3 試してみる

アクションとビューを作成したら、下記の URL で新しいページにアクセスすることが出来ます。

```
http://hostname/index.php?r=site%2Fsay&message=Hello+World
```

[36] http://ja.wikipedia.org/wiki/%E3%82%AF%E3%83%AD%E3%82%B9%E3%82%B5%E3%82%A4%E3%83%88%E3%82%B9%E3%82%AF%E3%83%AA%E3%83%97%E3%83%86%E3%82%A3%E3%83%B3%E3%82%B0



この URL は、結果として、"Hello World" を表示するページになります。このページはアプリケーションの他のページと同じヘッダとフッタを共有しています。

URL から `message` パラメータを省略すると、"こんにちは" を表示するページを見ることになるでしょう。 これは、`message` が `actionSay()` メソッドにパラメータとして渡されるものであり、それが省略された場合には、デフォルト値である こんにちは"" が代りに使われるからです。

> 情報: 新しいページは他のページと同じヘッダとフッタを共有していますが、それは `render()` メソッドが `say` ビューの結果を いわゆる レイアウト に自動的に埋め込むからです。 レイアウトは、この場合、`views/layouts/main.php` にあります。

上記の URL の `r` パラメータについては、さらに説明が必要でしょう。これは ルート、すなわち、アクションを指し示すアプリケーションを通じて一意な ID を表します。 ルートの書式は `ControllerID/ActionID` です。 アプリケーションはリクエストを受け取ると、このパラメータ `r` をチェックし、`ControllerID` の部分を使って、このリクエストを処理するためにどのコントローラ・クラスのインスタンスを作成すべきかを決定します。 そして、コントローラは `ActionID` の部分を使って、実際の仕事をするためにどのアクションを呼び出すべきかを決定します。 この例で言えば、`site/say` というルートは、`SiteController` コントローラ・クラスと `say` アクションとして解決されます。 結果として、`SiteController::actionSay()` メソッドがリクエストを処理するために呼び出されます。

> 情報: アクションと同じく、コントローラもまたアプリケーションの中で一意に定義される ID を持ちます。 コントロー

> ラ ID も、アクション ID と同じ命名規則を使います。 コントローラ・クラスの名前は、コントローラ ID からダッシュを削除し、各単語の最初の文字を大文字にし、 結果として出来る文字列に `Controller` という接尾辞を追加したものとなります。 例えば、`post-comment` というコントローラ ID に対応するコントローラ・クラスの名前は `PostCommentController` です。

2.4.4 まとめ

このセクションでは、MVC アーキテクチャ・パターンのうちのコントローラとビューの部分に触れました。 特定のリクエストを処理するためのアクションをコントローラの一部として作成しました。また、レスポンスのコンテントを作成するためのビューも作成しました。 この単純な例においては、使用される唯一のデータが `message` パラメータであったため、モデルは関係していません。

また、Yii におけるルートについても学びました。ルートはユーザのリクエストとコントローラのアクションとの橋渡しとして働くものです。

次のセクションでは、モデルを作成する方法を学びます。そして、HTML フォームを含むページを追加します。

2.5 フォームを扱う

このセクションでは、ユーザからデータを取得するためのフォームを持つ新しいページを作る方法を説明します。 このページは名前のインプット・フィールドとメールのインプット・フィールドを持つフォームを表示します。 ユーザからこれら二つの情報を受け取った後、ウェブ・ページは確認のために入力された値をエコー・バックします。

この目的を達するために、一つの アクション と 二つの ビュー を作成する以外に、 一つの モデル をも作成します。

このチュートリアルを通じて、次の方法を学びます。

- フォームを通じてユーザによって入力されるデータを表す モデル を作成する方法
- 入力されたデータを検証する規則を宣言する方法
- ビュー の中で HTML フォームを構築する方法

2.5.1 モデルを作成する

ユーザに入力してもらうデータは、下に示されているように `EntryForm` モデル・クラスとして表現され、 `models/EntryForm.php` というファイルに保存されます。 クラス・ファイルの命名規約についての詳細は クラスのオートロード のセクションを参照してください。

```
<?php
```

```
namespace app\models;

use Yii;
use yii\base\Model;

class EntryForm extends Model
{
    public $name;
    public $email;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

このクラスは、Yii によって提供される基底クラス yii\base\Model を拡張するものです。 通常、この基底クラスがフォーム・データを表現するのに使われます。

> 情報: yii\base\Model はデータベース・テーブルと関連しないモデル・クラスの親として使われます。 データベース・テーブルと対応するモデル・クラスでは、通常は yii\db\ActiveRecord が親になります。

EntryForm クラスは二つのパブリック・メンバー、name と email を持っており、これらがユーザによって入力されるデータを保管するのに使われます。 このクラスはまた rules() という名前のメソッドを持っています。このメソッドがデータを検証する一連の規則を返します。 上記で宣言されている検証規則は次のことを述べています。

- name と email は、ともに値を要求される
- email のデータは構文的に有効なメール・アドレスでなければならない

ユーザによって入力されたデータを EntryForm オブジェクトに投入した後、validate() メソッドを呼んでデータ検証ルーチンを始動することが出来ます。 データ検証が失敗すると hasErrors プロパティが true に設定されます。 そして、errors を通じて、どのような検証エラーが発生したかを知ることが出来ます。

```
<?php
$model = new EntryForm();
$model->name = 'Qiang';
$model->email = 'bad';
if ($model->validate()) {
    // 良し!
} else {
    // 失敗!
```

```
    // $model->getErrors() を使う
}
```

2.5.2 アクションを作成する

次に、この新しいモデルを使う entry アクションを site コントローラに作る必要があります。 アクションを作成して使うプロセスについては、こんにちは、と言う のセクションで既に説明されています。

```
<?php

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ... 既存のコード ...

    public function actionEntry()
    {
        $model = new EntryForm();

        if ($model->load(Yii::$app->request->post()) && $model->validate())
    {
            // $model に有効なデータを受け取った場合

            // ここで $model について何か意味のあることをする ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // ページの初期表示か、または、何か検証エラーがある場合
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

アクションは最初に EntryForm オブジェクトを生成します。 次に、モデルに $_POST のデータ、Yii においては yii\web\Request::post() によって提供されるデータを投入しようと試みます。 モデルへのデータ投入が成功した場合（つまり、ユーザが HTML フォームを送信した場合)、アクションはvalidate() を呼んで、 入力された値が有効なものであるかどうかを確認します。

> 情報: Yii::$app という式は アプリケーション インスタンスを表現します。 これはグローバルにアクセス可能なシングルトンです。 これは、また、特定の機能性をサポートする request 、response、db などのコンポーネントを提供する サービス・

> ロケータ でもあります。 上記のコードでは、アプリケーション・インスタンスの request コンポーネントが $_POST データにアクセスするために使われています。

すべてが適正である場合、アクションは entry-confirm という名前のビューを表示して、データの送信が成功したことをユーザに確認させます。 データが送信されなかったり、データがエラーを含んでいたりする場合は、entry ビューが表示され、 その中で HTML フォームが (もし有れば) 検証エラーのメッセージとともに表示されます。

> 補足: この簡単な例では、有効なデータ送信に対して単純に確認ページを表示しています。 実際の仕事では、フォーム送信の諸問題[37] を避けるために、 refresh() または redirect() を使うことを考慮すべきです。

2.5.3 ビューを作成する

最後に、entry-confirm と entry という名前の二つのビュー・ファイルを作成します。 今まさに説明したように、これらが entry アクションによって表示されます。

entry-confirm ビューは単純に名前とメールのデータを表示するものです。このビューは views/site/entry-confirm.php というファイルに保存しなければなりません。

```
<?php
use yii\helpers\Html;
?>
<pあなたは次の情報を入力しました></p>

<ul>
    <li><label名前></label>: <?= Html::encode($model->name) ?></li>
    <li><labelメール></label>: <?= Html::encode($model->email) ?></li>
</ul>
```

entry ビューは HTML フォームを表示します。これは views/site/entry.php というファイルに保存しなければなりません。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'name') ?>

    <?= $form->field($model, 'email') ?>

    <div class="form-group">
```

[37] http://en.wikipedia.org/wiki/Post/Redirect/Get

```
        <?= Html::submitButton('送信', ['class' => 'btn btn-primary']) ?>
    </div>

<?php ActiveForm::end(); ?>
```

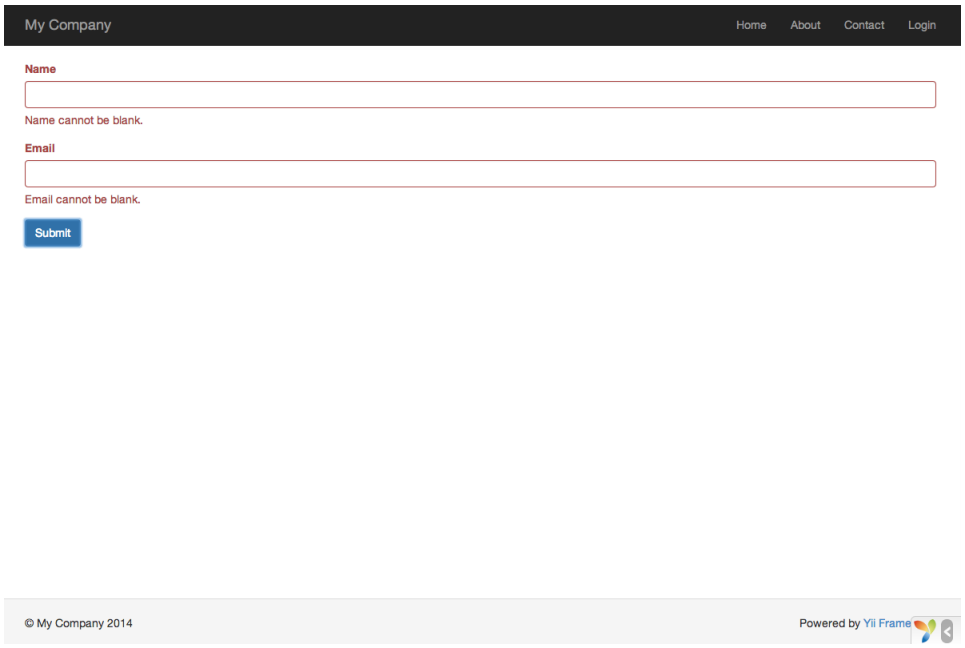
このビューは HTML フォームを構築するのに、ActiveForm と呼ばれる強力な ウィジェット を使います。 ウィジェットの begin() メソッドと end() メソッドが、それぞれ、フォームの開始タグと終了タグをレンダリングします。 この二つのメソッドの呼び出しの間に、field() メソッドによってインプット・フィールドが作成されます。 最初のインプット・フィールドは “name” のデータ、第二のインプット・フィールドは “email” のデータのためのものです。 インプット・フィールドの後に、yii\helpers\Html::submitButton() メソッドが呼ばれて、送信ボタンを生成しています。

2.5.4 試してみる

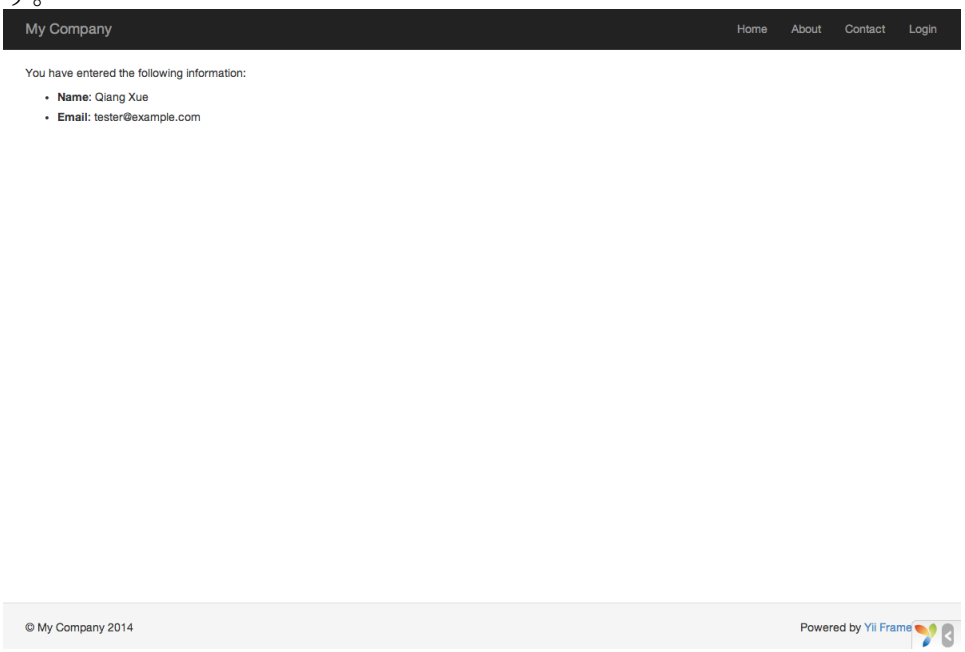
どのように動作するかを見るために、ブラウザで下記の URL にアクセスしてください。

```
http://hostname/index.php?r=site%2Fentry
```

二つのインプット・フィールドを持つフォームを表示するページが表示されるでしょう。それぞれのインプット・フィールドの前には、どんなデータを入力すべきかを示すラベルがあります。 何も入力せずに、あるいは、無効なメール・アドレスを入力して送信ボタンをクリックすると、それぞれ問題のあるインプット・フィールドの後ろにエラー・メッセージが表示されます。



有効な名前とメール・アドレスを入力してから送信ボタンをクリックすると、 たった今入力したデータを表示する新しいページが表示されます。



魔法の説明

あなたは、舞台裏で HTML フォームがどのように動いているのか、不思議に思うかも知れません。 なぜなら、フォームが、ほとんど魔法のように、各インプット・フィールドのラベルを表示し、データを正しく入力しなかった場合には、 ページをリロードすることなく、エラー・メッセージを表示するからです。

そう、データの検証は、最初に JavaScript を使ってクライアント・サイドで実行され、次に PHP によってサーバ・サイドで実行されます。 `yii\widgets\ActiveForm` は、賢いことに、`EntryForm` で宣言した検証規則を抽出し、それを実行可能な JavaScript コードに変換して、 JavaScript を使ってデータ検証を実行します。 ブラウザで JavaScript を無効にした場合でも、`actionEntry()` メソッドで示されているように、サーバ・サイドでの検証は引き続き実行されます。 これにより、どのような状況であっても、データの有効性が保証されます。

> 警告: クライアント・サイドの検証は、ユーザにとってのより良い使い心地のために利便性を提供するものです。 クライアント・サイドの検証の有無にかかわらず、サーバ・サイドの検証は常に必要です。

インプット・フィールドのラベルは、モデルのプロパティ名を使用して、`field()` メソッドによって生成されます。 例えば、`name` というプロパ

ティから Name というラベルが生成されます。

ビューの中で、下記のコードのように、 ラベルをカスタマイズすることも出来ます。

```
<?= $form->field($model, 'name')->label('お名前') ?>
<?= $form->field($model, 'email')->label('メール・アドレス') ?>
```

> 情報: Yii はこのようなウィジェットを数多く提供して、複雑で動的なビューを素速く作成することを手助けしてくれます。 後で学ぶように、新しいウィジェットを書くことも非常に簡単です。 あなたは、将来のビュー開発を単純化するために、多くのビュー・コードを再利用可能なウィジェットに変換したいと思うことでしょう。

2.5.5 まとめ

ガイドのこのセクションにおいては、MVC アーキテクチャ・パターンの全ての部分に触れました。 そして、ユーザ・データを表現し、当該データを検証するモデル・クラスを作成する方法を学びました。

また、ユーザからデータを取得する方法と、ブラウザにデータを表示して返す方法も学びました。 この作業は、アプリケーションを開発するときに、多大な時間を必要とするものになり得るものです。 しかし、Yii はこの作業を非常に容易にする強力なウィジェットを提供しています。

次のセクションでは、ほとんど全てのアプリケーションで必要とされるデータベースを取り扱う方法を学びます。

2.6 データベースを扱う

このセクションでは、country という名前のデータベース・テーブルから読み出した国データを表示する新しいページの作り方を説明します。 この目的を達するために、データベース接続を構成し、 アクティブ・レコード クラスを作成し、アクション を定義し、 そして ビュー を作成します。

このチュートリアルを通じて、次のことを学びます。

- DB 接続を構成する方法
- アクティブ・レコードのクラスを定義する方法
- アクティブ・レコードのクラスを使ってデータを検索する方法
- 改ページを伴う仕方でビューにデータを表示する方法

このセクションを完了するためには、データベースを使うことについて基本的な知識と経験が無ければならないことに注意してください。 具体的に言えば、DB クライアント・ツールを用いてデータベースを作成する方法と、SQL 文を実行する方法を知っていなければなりません。

2.6.1 データベースを準備する

まず初めに、yii2basic という名前のデータベースを作成してください。このデータベースからアプリケーションにデータを読み出すことになります。 Yii は多数のデータベース製品に対するサポートを内蔵しており、作成するデータベースは、SQLite、MySQL、PosttreSQL、MSSQL または Oracle から選ぶことが出来ます。以下の説明では、話を単純にするために、MySQL を前提とします。

> 情報: MariaDB は、かつては MySQL と差し替え可能な代替物でしたが、現在では完全にそうだとは言えません。MariaDB で JSON サポートのような高度な機能を使いたいときは、後述する MariaDB エスステンションの使用を検討して下さい。

次に、データベースに country という名前のテーブルを作り、いくつかのサンプル・データを挿入します。そうするためには、次の SQL 文を実行することが出来ます。

```
CREATE TABLE `country` (
  `code` CHAR(2) NOT NULL PRIMARY KEY,
  `name` CHAR(52) NOT NULL,
  `population` INT(11) NOT NULL DEFAULT '0'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `country` VALUES ('AU','Australia',24016400);
INSERT INTO `country` VALUES ('BR','Brazil',205722000);
INSERT INTO `country` VALUES ('CA','Canada',35985751);
INSERT INTO `country` VALUES ('CN','China',1375210000);
INSERT INTO `country` VALUES ('DE','Germany',81459000);
INSERT INTO `country` VALUES ('FR','France',64513242);
INSERT INTO `country` VALUES ('GB','United Kingdom',65097000);
INSERT INTO `country` VALUES ('IN','India',1285400000);
INSERT INTO `country` VALUES ('RU','Russia',146519759);
INSERT INTO `country` VALUES ('US','United States',322976000);
```

この時点で、あなたは yii2basic という名前のデータベースを持ち、その中に三つのカラムを持つ country というテーブルがあり、country テーブルは 10 行のデータを持っている、ということになります。

2.6.2 DB 接続を構成する

先に進む前に、PDO[38] PHP 拡張および使用しているデータベースの PDO ドライバ (例えば、MySQL のための pdo_mysql) の両方をインストール済みであることを確認してください。 アプリケーションがリレーショナル・データベースを使う場合、これは基本的な必要条件です。

これらがインストール済みなら、config/db.php というファイルを開いて、あなたのデータベースに適合するようにパラメータを変更してください。 デフォルトでは、このファイルは下記の記述を含んでいます。

[38] https://secure.php.net/manual/en/book.pdo.php

```
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

この config/db.php というファイルは典型的なファイル・ベースの 構成情報 ツールです。 この構成情報ファイルが、背後のデータベースに対する SQL クエリの実行を可能にする yii\db\Connection インスタンスの作成と初期化に必要なパラメータを指定するものです。

上記のようにして構成された DB 接続は、アプリケーション・コードの中で Yii::$app->db という式でアクセスすることが出来ます。

> 情報: config/db.php は、メインのアプリケーション構成情報ファイルである config/web.php によってインクルードされます。 この config/web.php が アプリケーション インスタンスが初期化される仕方を指定するものです。 詳しい情報については、構成情報 のセクションを参照してください。

Yii がサポートを内蔵していないデータベースを扱う必要がある場合は、以下のエクステンションの利用を検討してください。

- Informix[39]
- IBM DB2[40]
- Firebird[41]
- MariaDB[42]

2.6.3 アクティブ・レコードを作成する

country テーブルの中のデータを表現し取得するために、アクティブ・レコード から派生した Country という名前のクラスを作成し、 それを models/Country.php というファイルに保存します。

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

[39] https://github.com/edgardmessias/yii2-informix
[40] https://github.com/edgardmessias/yii2-ibm-db2
[41] https://github.com/edgardmessias/yii2-firebird
[42] https://github.com/sam-it/yii2-mariadb

Country クラスは yii\db\ActiveRecord を拡張しています。この中には一つもコードを書く必要はありません。 単に上記のコードだけで、Yii は関連付けられたテーブル名をクラス名から推測します。

> 情報: クラス名とテーブル名を直接に合致させることが出来ない場合は、yii\db\ActiveRecord::tableName() メソッドをオーバーライドして、関連づけられたテーブル名を明示的に指定することが出来ます。

Country クラスを使うことによって、以下のコード断片で示すように、country テーブルの中のデータを簡単に操作することが出来ます。

```
use app\models\Country;

// country テーブルから全ての行を取得して "name" 順に並べる
$countries = Country::find()->orderBy('name')->all();

// プライマリ・キーが "US" である行を取得する
$country = Country::findOne('US');

// "United States" を表示する
echo $country->name;

// 国名を "U.S.A." に修正してデータベースに保存する
$country->name = 'U.S.A.';
$country->save();
```

> 情報: アクティブ・レコードは、オブジェクト指向の流儀でデータベースのデータにアクセスし、操作する強力な方法です。アクティブ・レコード のセクションで、詳細な情報を得ることが出来ます。 もう一つの方法として、データベース・アクセス・オブジェクト と呼ばれる、より低レベルなデータ・アクセス方法を使ってデータベースを操作することも出来ます。

2.6.4 アクションを作成する

国データをエンド・ユーザに公開するために、新しいアクションを作成する必要があります。 これまでのセクションでしたように site コントローラの中に新しいアクションを置くのではなく、国データに関係する全てのアクションに限定した新しいコントローラを作成する方が理にかなうでしょう。 この新しいコントローラを CountryController と名付けます。そして、下記に示すように、index アクションをその中に作成します。

```
<?php

namespace app\controllers;
```

```
use yii\web\Controller;
use yii\data\Pagination;
use app\models\Country;

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();

        return $this->render('index', [
            'countries' => $countries,
            'pagination' => $pagination,
        ]);
    }
}
```

上記のコードを `controllers/CountryController.php` というファイルに保存します。

最初に `index` アクションは `Country::find()` を呼び出します。この find()[43] メソッドが `country` テーブルからデータを取得するメソッドを提供する ActiveQuery[44] クエリ・オブジェクトオブジェクトを生成します。

一回のリクエストで返される国の数を制限するために、クエリ・オブジェクトは `yii\data\Pagination` オブジェクトの助けを借りてページ付けされます。 `Pagination` オブジェクトは二つの目的に奉仕します。

- クエリによって表現される SQL 文に `offset` 句と `limit` 句をセットして、 一度に一ページ分のデータだけ (1ページ最大5行) を返すようにします。
- 次の項で説明されるように、一連のページ・ボタンからなるページャを ビューに表示するために使われます。

次に、all()[45] メソッドがクエリ結果に基づいて全ての `country` レコードを返します。

コードの最後で、`index` アクションは `index` と言う名前のビューをレン

[43] https://www.yiiframework.com/doc/api/2.0/yii-db-activerecord#find()-detail
[44] https://www.yiiframework.com/doc/api/2.0/yii-db-activequery
[45] https://www.yiiframework.com/doc/api/2.0/yii-db-activequery#all()-detail

ダリングします。 このときに、返された国データとそのページネーション情報がビューに渡されます。

2.6.5 ビューを作成する

最初に、views ディレクトリの下に country という名前のサブ・ディレクトリを作ってください。 このフォルダが country コントローラによって表示される全てのビューを保持するのに使われます。 views/country ディレクトリの中に、下記のコードを含む index.php という名前のファイルを作成します。

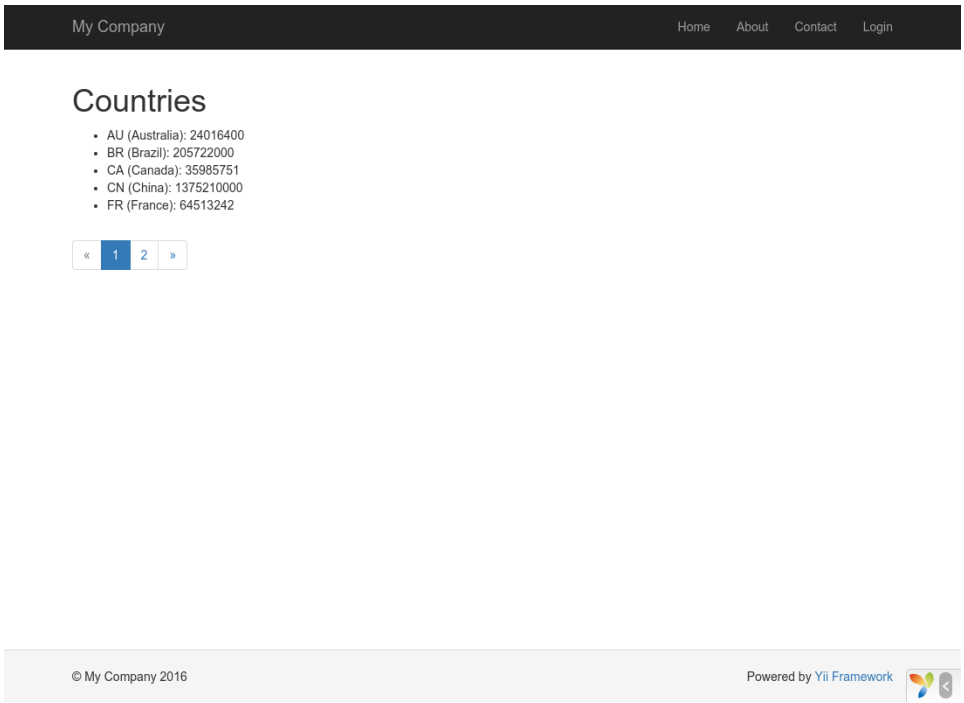
```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1国リスト></h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->code} ({$country->name})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

ビューは国データの表示に関連して二つの部分に分けられます。 最初の部分では、提供された国データがたどられて、HTML の順序無しリストとしてレンダリングされます。 第二の部分では、アクションから渡されたページネーション情報を使って、yii\widgets\LinkPager ウィジェットがレンダリングされます。 LinkPager ウィジェットはページ・ボタンのリストを表示します。ボタンのどれかをクリックすると、対応するページの国データが更新表示されます。

2.6.6 試してみる

上記のコード全てがどのように動作するかを見るために、ブラウザで下記の URL をアクセスします。

```
http://hostname/index.php?r=country%2Findex
```



最初、ページは5つの国を表示しています。そして、国リストの下には、4つのボタンを持ったページャがあります。 “2” のボタンをクリックすると、ページはデータベースにある次の5つの国、すなわち、2ページ目のレコードを表示します。 注意深く観察すると、ブラウザの URL も次のように変ったことに気付くでしょう。

```
http://hostname/index.php?r=country%2Findex&page=2
```

舞台裏では、`Pagination` が、データ・セットをページ付けするのに必要な全ての機能を提供しています。

- 初期状態では、`Pagination` は、1ページ目を表しています。 これを反映して、国の SELECT クエリは `LIMIT 5 OFFSET 0` という句を伴うことになります。その結果、最初の5つの国が取得されて表示されます。
- `LinkPager` ウィジェットは、`Pagination` によって作成された URL を使ってページ・ボタンをレンダリングします。 URL は、別々のページ番号を表現する `page` というクエリ・パラメータを含んだものになります。
- ページ・ボタン “2” をクリックすると、`country/index` のルートに対する新しいリクエストが発行され、処理されます。 `Pagination` が URL から `page` クエリ・パラメータを読み取って、カレント・ページ番号を 2 にセットします。 こうして、新しい国のクエリは `LIMIT 5 OFFSET 5` という句を持ち、 次の5つの国を表示のために返すことになります。

2.6.7 まとめ

このセクションでは、データベースを扱う方法を学びました。 また、yii\data\Pagination と yii\widgets\LinkPager の助けを借りて、ページ付けされたデータを取得し表示する方法も学びました。

次のセクションでは、Gii[46] と呼ばれる強力なコード生成ツールを使う方法を学びます。 このツールは、データベース・テーブルのデータを取り扱うための「作成・読出し・更新・削除 (CRUD)」操作のような、通常必要とされることが多い諸機能の迅速な実装を手助けしてくれるものです。 実際のところ、あなたがたった今書いたばかりのコードは、Gii ツールを使えば、全部、Yii が自動的に生成してくれるものです。

2.7 Gii でコードを生成する

このセクションでは、Gii[47] を使って、ウェブ・サイトの一般的な機能のいくつかを実装するコードを自動的に生成する方法を説明します。 Gii を使ってコードを自動生成することは、Gii のウェブ・ページに表示される指示に対して正しい情報を入力するだけのことです。

このチュートリアルを通じて、次のことを学びます。

- アプリケーションで Gii を有効にする方法
- Gii を使って、アクティブ・レコードのクラスを生成する方法
- Gii を使って、DB テーブルの CRUD 操作を実装するコードを生成する方法
- Gii によって生成されるコードをカスタマイズする方法

2.7.1 Gii を開始する

Gii[48] は Yii の モジュール として提供されています。 Gii は、アプリケーションの modules プロパティの中で構成することで有効にすることが出来ます。アプリケーションを生成した仕方にもよりますが、config/web.php の構成情報ファイルの中に、多分、下記のコードが既に提供されているでしょう。

```
$config = [ ... ];

if (YII_ENV_DEV) {
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
    ];
}
```

[46] https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
[47] https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
[48] https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide

上記の構成情報は、開発環境 において、アプリケーションは gii という名前のモジュールをインクルードすべきこと、 そして gii は yii\gii\Module というクラスであることを記述しています。

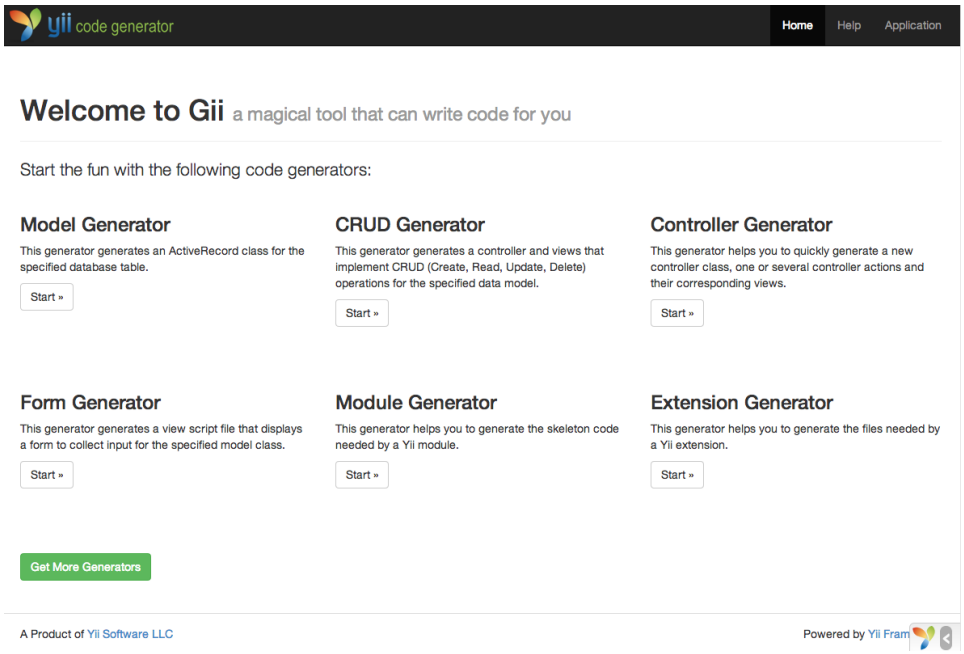
アプリケーションの エントリ・スクリプト である web/index.php をチェックすると、次の行があることに気付くでしょう。 これは本質的には YII_ENV_DEV を true に設定するものです。

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

この行のおかげで、アプリケーションは開発モードになっており、上記の構成情報によって、Gii が既に有効になっています。これで、下記の URL によって Gii にアクセスすることが出来ます。

```
http://hostname/index.php?r=gii
```

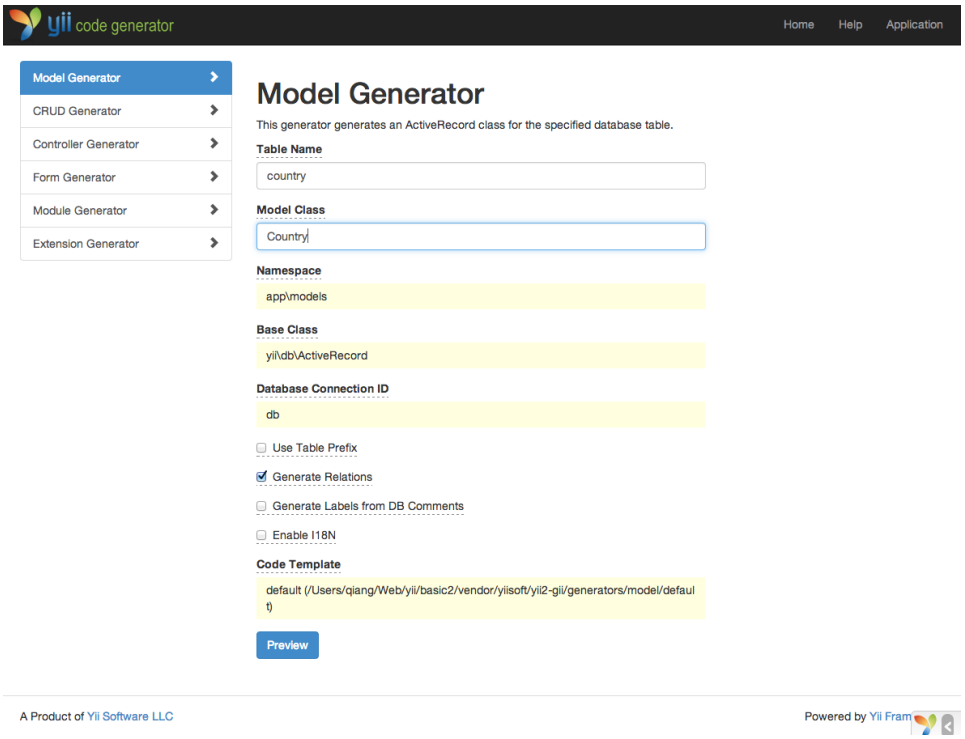
> 補足: ローカルホスト以外のマシンから Gii にアクセスしようとすると、デフォルトではセキュリティ上の理由でアクセスが拒否されます。 下記のように Gii を構成して、許可される IP アドレスを追加することが出来ます。
>
> ```
> 'gii' => [
>     'class' => 'yii\gii\Module',
>     'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '
>     192.168.178.20'] // 必要に応じて調
>     整
> ],
> ```



2.7.2 アクティブ・レコードのクラスを生成する

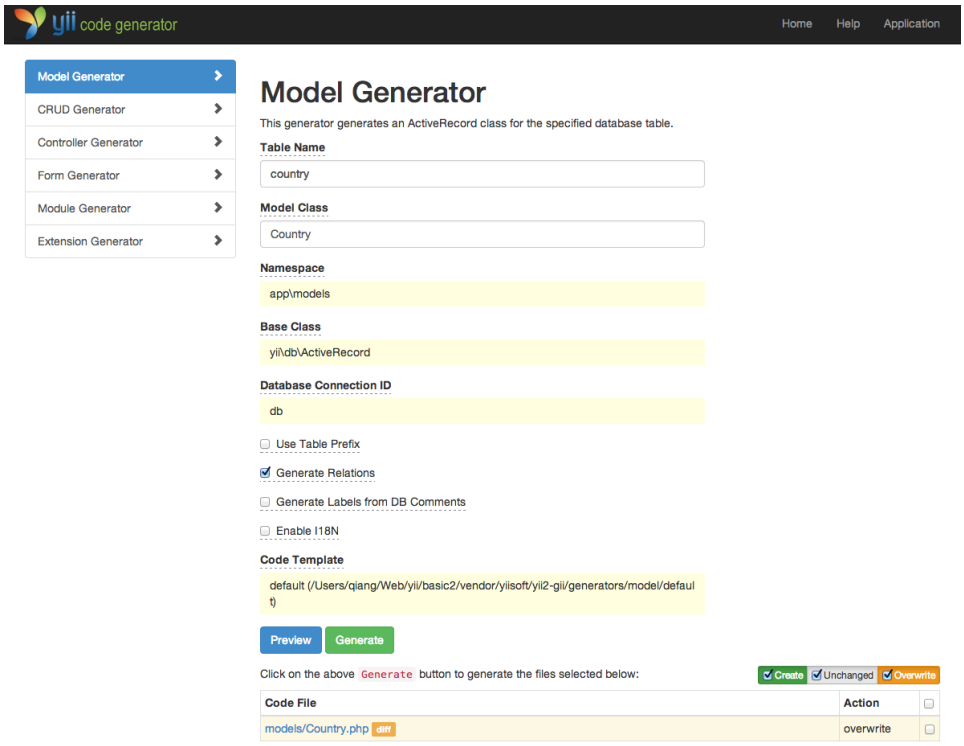
Gii を使ってアクティブ・レコードのクラスを生成するためには、"Model Generator" を選びます (Gii のインデックス・ページのリンクをクリックして下さい)。そして、次のようにフォームに入力します。

- Table Name: `country`
- Model Class: `Country`



次に、"Preview" ボタンをクリックします。そうすると、結果として作成されるクラス・ファイルのリストに `models/Country.php` が挙ってきます。クラス・ファイルの名前をクリックすると、内容をプレビューすることが出来ます。

Gii を使うときに、既に同じファイルを作成していて、それを上書きしようとしている場合は、 ファイル名の隣の `diff` ボタンをクリックして、 生成されようとしているコードと既存のバージョンの違いを見てください。



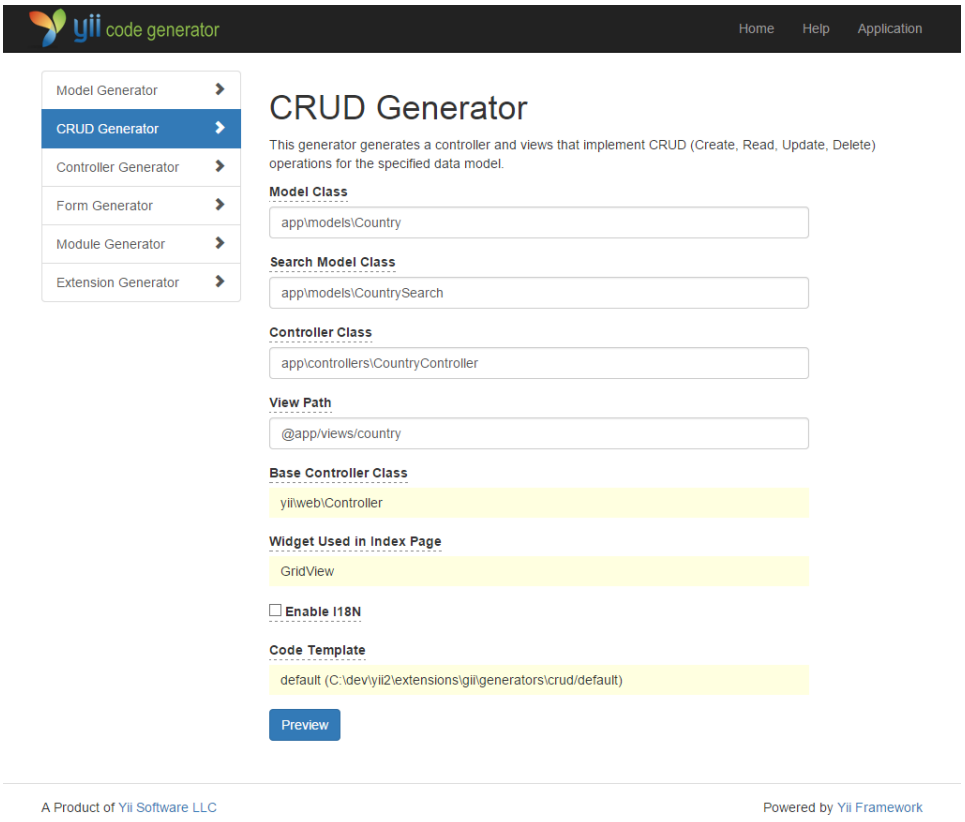
既存のファイルを上書きするときは、"overwrite” の隣のチェックボックスをチェックしてから “Generate” ボタンをクリックします。新しいファイルを作成するときは、単に “Generate” をクリックすれば十分です。

次に、コードの生成が成功したことを示す確認ページが表示されます。 既存のファイルがあった場合は、それが新しく生成されたコードで上書きされたことを示すメッセージも同じく表示されます。

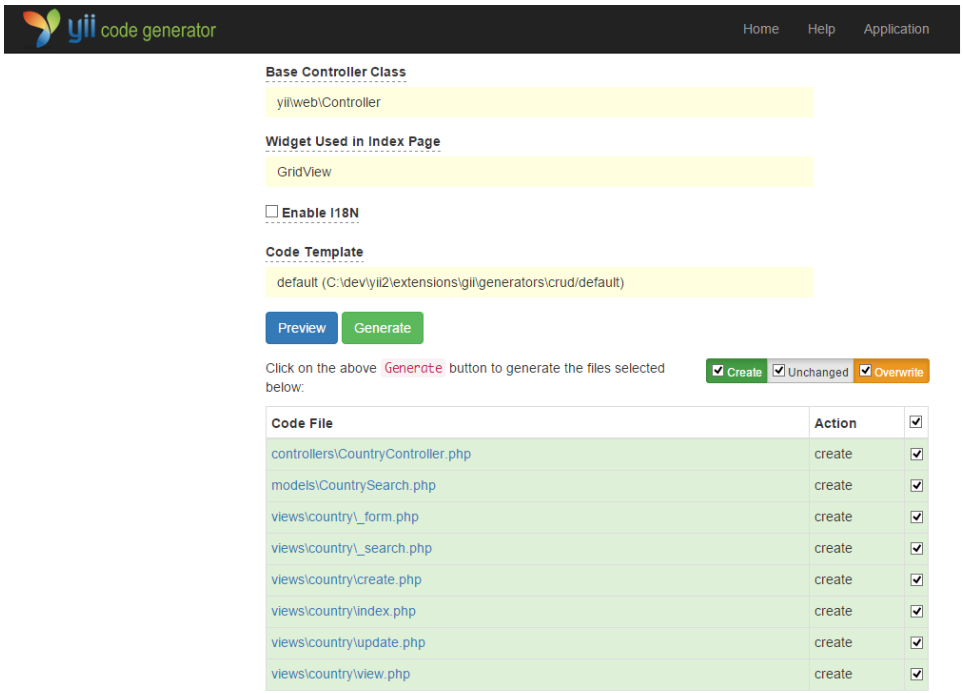
2.7.3 CRUD コードを生成する

CRUD は Create(作成)、Read(読出し)、Update(更新)、そして Delete(削除) を意味しており、ほとんどのウェブ・サイトでデータを扱うときによく用いられる4つのタスクを表しています。Gii を使って CRUD 機能を作成するためには、"CRUD Generator” を選びます (Gii のインデックス・ページのリンクをクリックしてください) 。「国リスト」のサンプルのためには、表示されたフォームに以下のように入力します。

- Model Class: `app\models\Country`
- Search Model Class: `app\models\CountrySearch`
- Controller Class: `app\controllers\CountryController`



次に、"Preview” ボタンをクリックします。生成されるファイルのリストは、次のようになります。



以前に（ガイドのデータベースのセクションで）`controllers/CountryController.php` と `views/country/index.php` のファイルを作成していた場合は、それらを置き換えるために “overwrite” のチェックボックスをチェックしてください。 (以前のバージョンは フル機能の CRUD をサポートしていません。)

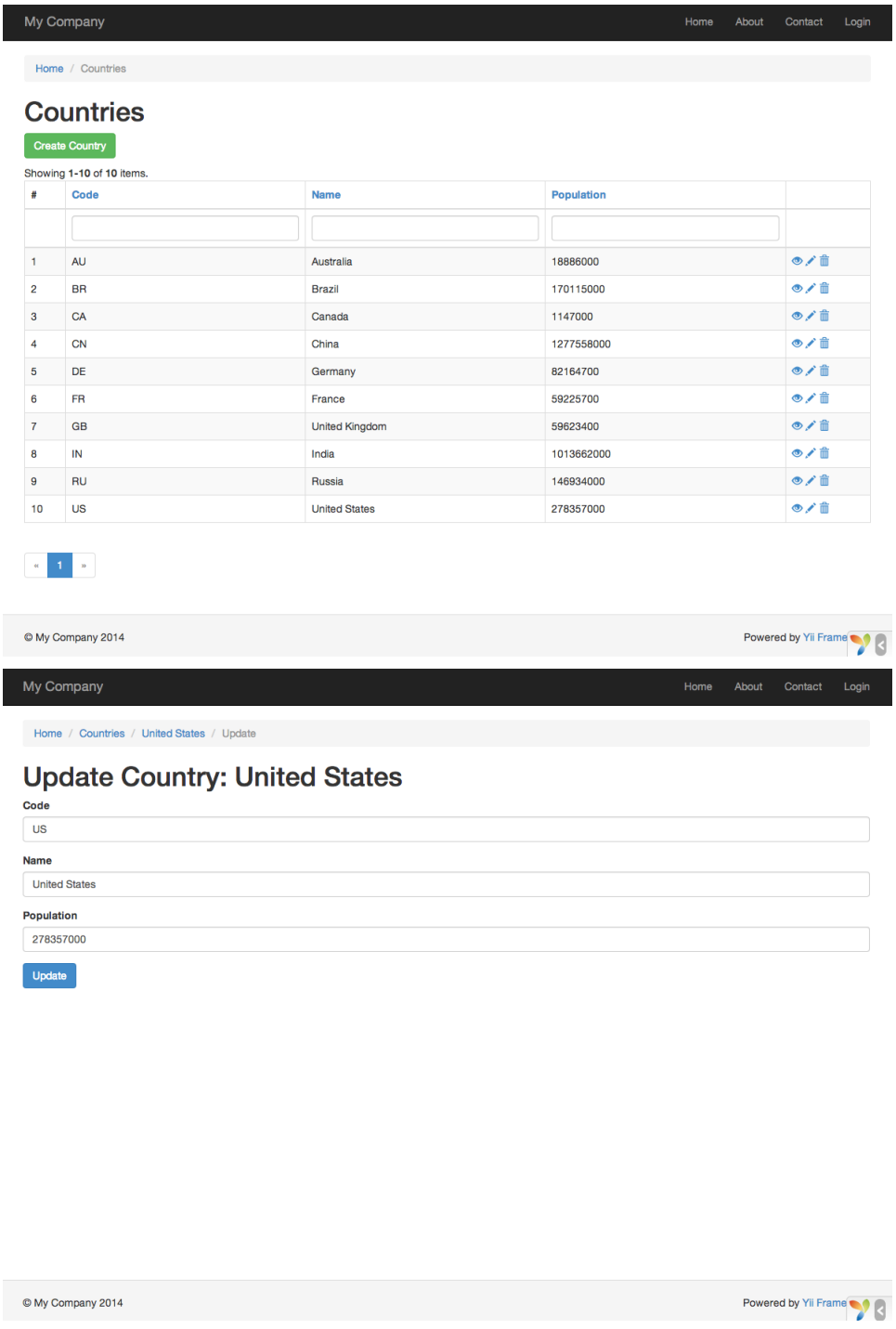
2.7.4 試してみる

どのように動作するかを見るために、ブラウザを使って下記の URL にアクセスしてください。

```
http://hostname/index.php?r=country%2Findex
```

データ・グリッドがデータベース・テーブルから取得した国を表示しているページが表示されます。 グリッドをソートしたり、カラムのヘッダに検索条件を入力してグリッドにフィルタを適用したりすることが出来ます。

グリッドに表示されているそれぞれの国について、詳細を見たり、更新したり、または削除したりすることが出来ます。 また、グリッドの上にある “Create Country” ボタンをクリックすると、新しい国データを作成するためのフォームが利用に供されます。



下記が Gii によって生成されるファイルのリストです。 これらの機能がどのように実装されているかを調査したい場合、また、これらの機能をカスタマイズしたいときに参照してください。

- Controller: `controllers/CountryController.php`

- Models: `models/Country.php` と `models/CountrySearch.php`
- Views: `views/country/*.php`

情報: Gii は非常にカスタマイズしやすく拡張しやすいコード生成ツールとして設計されています。 これを賢く使うと、アプリケーションの開発速度を大いに高めることが出来ます。詳細については、Gii[49] のセクションを参照してください。

2.7.5 まとめ

このセクションでは、Gii を使ってコードを生成して、データベース・テーブルに保存されているコンテントのための 完全な CRUD 機能を実装する方法を学びました。

2.8 先を見通す

「はじめよう」の章全体を読み通したなら、いまやあなたは、完全な Yii のアプリケーションを作成したことがある、ということになります。その過程で、あなたは必要とされることが多いいくつかの機能、例えば、HTML フォームを通じてユーザからデータを取得することや、データベースからデータを取得すること、また、ページ付けをしてデータを表示することなどを実装する方法を学びました。 また、Gii[50] を使ってコードを自動的に生成する方法も学びました。 Gii をコード生成に使うと、ウェブ開発のプロセスの大部分が、いくつかのフォームに入力していくだけの簡単な仕事になります。

このセクションでは、Yii フレームワークを使うときの生産性を更に高めるために利用できるリソースについてまとめます。

- ドキュメント
  - 決定版ガイド[51]: 名前が示すように、このガイドは Yii がどのように動作すべきものかを正確に記述し、Yii を使用するについての全般的な手引きを提供するものです。 これは唯一の最も重要な Yii のチュートリアルであり、Yii のコードを少しでも書く前にあなたが読まなければならないものです。
  - クラス・リファレンス[52]: これは Yii によって提供される全てのクラスの使用法を記述しています。 主として、コードを書いている時に、特定のクラス、メソッド、プロパティについて理解したい場合に読まれるべきものです。 クラス・リファレンスの使用は、フレームワーク全体の文脈的な理解が出来てからにするのが最善です。
  - Wiki の記事[53]: Wiki の記事は、Yii のユーザが自身の経験に基

[49] https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
[50] https://www.yiiframework.com/extension/yiisoft/yii2-gii/doc/guide
[51] http://www.yiiframework.com/doc-2.0/guide-README.html
[52] http://www.yiiframework.com/doc-2.0/index.html
[53] http://www.yiiframework.com/wiki/?tag=yii2

づいて書いたものです。 ほとんどの記事は、料理本のレシピのように書かれており、特定の問題を Yii を使って解決する方法を示しています。 これらの記事の品質は決定版ガイドほどには良くないかもしれませんが、より広範なトピックをカバーしていることと、たいていは即座に使えるソリューションを提供してくれることにおいて有用なものです。
  - 書籍[54]
- エクステンション[55]: Yii は、ユーザによって作られた数千におよぶエクステンションのライブラリを誇りとしています。 エクステンションはあなたのアプリケーションに簡単に組み込むことが出来、そうすることでアプリケーションの開発作業をより一層速くて簡単なものにします。
- コミュニティ
  - フォーラム: http://www.yiiframework.com/forum/
  - IRC チャット: freenode ネットワーク (irc://irc.freenode.net/yii) の #yii チャンネル
  - Slack チャンネル: https://join.slack.com/t/yii/shared_invite/enQtMzQ4MDExMDcyNTk2LTc0NDQ2ZTZhNjkzZDgwYjE4YjZlNGQxZjFmZDBjZTU3NjViMDE4ZTMxNDRk
  - Gitter チャット: https://gitter.im/yiisoft/yii2
  - GitHub: https://github.com/yiisoft/yii2
  - Facebook: https://www.facebook.com/groups/yiitalk/
  - Twitter: https://twitter.com/yiiframework
  - LinkedIn: https://www.linkedin.com/groups/yii-framework-1483367
  - Stackoverflow: http://stackoverflow.com/questions/tagged/yii2

[54] http://www.yiiframework.com/doc/
[55] http://www.yiiframework.com/extensions/

Chapter 3

アプリケーションの構造

3.1 概要

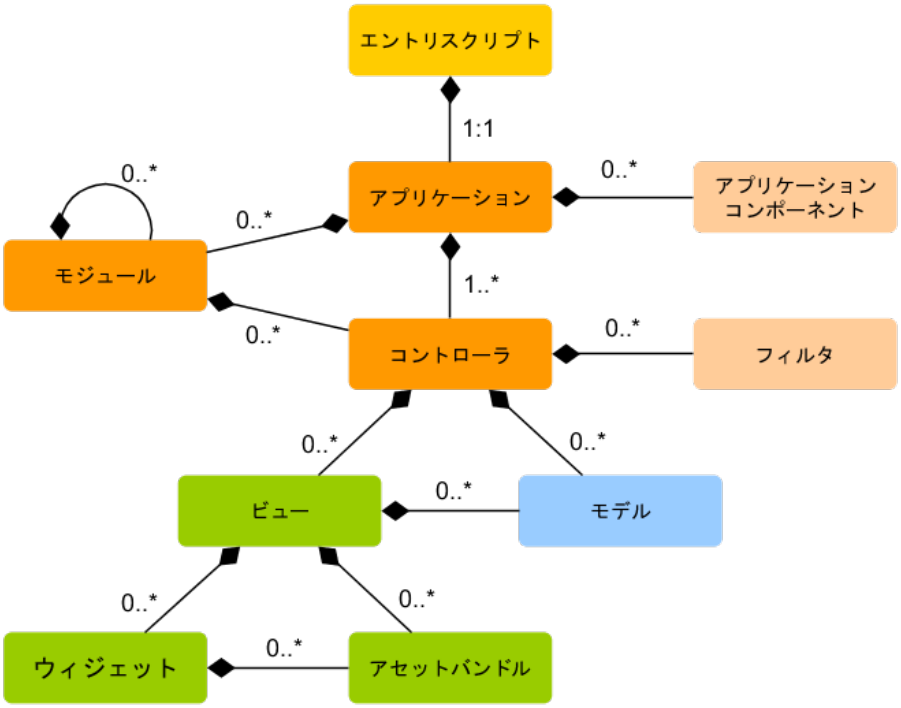
Yii のアプリケーションは モデル・ビュー・コントローラ (MVC)[1] アーキテクチャ・パターンに従って編成されています。 モデル は、データ、ビジネス・ロジック、規則を表現します。 ビュー は、モデルの出力表現です。 そして コントローラ は入力を受け取って、それを モデル と ビュー のためのコマンドに変換します。

MVC 以外にも、Yii のアプリケーションは下記の要素を持っています。

- エントリ・スクリプト: エンド・ユーザから直接アクセスできる PHP スクリプトです。 これはリクエスト処理サイクルを開始する役目を持っています。
- アプリケーション: グローバルにアクセス可能なオブジェクトであり、 アプリケーション・コンポーネントを管理し、連携させて、リクエストに応えます。
- アプリケーション・コンポーネント: アプリケーションと共に登録されたオブジェクトであり、 リクエストに応えるための様々なサービスを提供します。
- モジュール: それ自身に完全な MVC を含む自己完結的なパッケージです。 アプリケーションは複数のモジュールとして編成することが出来ます。
- フィルタ: 各リクエストが実際に処理される前と後に、 コントローラから呼び出される必要があるコードを表現します。
- ウィジェット: ビュー に埋め込むことが出来るオブジェクトです。 コントローラのロジックを含むことが可能で、異なるビューで再利用することが出来ます。

下の図がアプリケーションの静的な構造を示すものです。

[1] http://ja.wikipedia.org/wiki/Model_View_Controller



3.2 エントリ・スクリプト

エントリ・スクリプトは、アプリケーションのブートストラップの過程における最初のステップです。 アプリケーションは (ウェブ・アプリケーションであれ、コンソール・アプリケーションであれ) 単一のエントリ・スクリプトを持ちます。 エンド・ユーザはエントリ・スクリプトに対してリクエストを発行し、エントリ・スクリプトはアプリケーションのインスタンスを作成して、それにリクエストを送付します。

ウェブ・アプリケーションのエントリ・スクリプトは、エンド・ユーザからアクセス出来るように、 ウェブからのアクセスが可能なディレクトリの下に保管されなければなりません。 たいていは `index.php` と名付けられますが、ウェブ・サーバが見つけることが出来る限り、どのような名前を使っても構いません。

コンソール・アプリケーションのエントリ・スクリプトは、通常は、アプリケーションの ベース・パス の下に保管され、`yii` と名付けられます (`.php` の拡張子を伴います) 。 これは、ユーザが `./yii <route> 引数[] オプション[]` というコマンドによってコンソール・アプリケーションを走らせることが出来るようにするためのスクリプトであり、 実行可能なパーミッションを与えられるべきものです。

エントリ・スクリプトは主として次の仕事をします。

- グローバルな定数を定義する。

- Composer のオートローダ[2] を登録する。
- `Yii` クラス・ファイルをインクルードする。
- アプリケーションの構成情報を読み出す。
- アプリケーション のインスタンスを生成して構成する。
- `yii\base\Application::run()` を呼んで、受け取ったリクエストを処理する。

3.2.1 ウェブ・アプリケーション

次に示すのが、ベーシック・ウェブ・プロジェクト・テンプレート のエントリ・スクリプトです。

```
<?php

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// Composer のオートローダを登録
require __DIR__ . '/../vendor/autoload.php';

// Yii クラス・ファイルをインクルード
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// アプリケーションの構成情報を読み出す
$config = require __DIR__ . '/../config/web.php';

// アプリケーションを作成し、構成して、走らせる
(new yii\web\Application($config))->run();
```

3.2.2 コンソール・アプリケーション

同様に、下記がコンソール・アプリケーションのエントリ・スクリプトです。

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 *
 * @link http://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license http://www.yiiframework.com/license/
 */

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// Composer のオートローダを登録
require __DIR__ . '/vendor/autoload.php';
```

[2] https://getcomposer.org/doc/01-basic-usage.md#autoloading

```
// Yii クラス・ファイルをインクルード
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';

// アプリケーションの構成情報を読み出す
$config = require __DIR__ . '/config/console.php';

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

3.2.3 定数を定義する

グローバルな定数を定義するには、エントリ・スクリプトが最善の場所です。Yii は下記の三つの定数をサポートしています。

- YII_DEBUG: アプリケーションがデバッグ・モードで走るかどうかを指定します。 デバッグ・モードにおいては、アプリケーションはより多くのログ情報を保持し、例外が投げられたときに、より詳細なエラーのコール・スタックを表示します。 この理由により、デバッグ・モードは主として開発時に使用されるべきものとなります。YII_DEBUG のデフォルト値は false です。
- YII_ENV: どういう環境でアプリケーションが走っているかを指定します。 詳細は、構成情報 のセクションで説明されます。 YII_ENV のデフォルト値は 'prod' であり、アプリケーションが本番環境で走っていることを意味します。
- YII_ENABLE_ERROR_HANDLER: Yii によって提供されるエラー・ハンドラを有効にするかどうかを指定します。 この定数のデフォルト値は true です。

定数を定義するときには、しばしば次のようなコードを用います。

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

これは下記のコードと同じ意味のものです。

```
if (!defined('YII_DEBUG')) {
    define('YII_DEBUG', true);
}
```

明らかに前者の方が簡潔で理解しやすいでしょう。

他のPHP ファイルがインクルードされる時に定数の効力が生じるようにするために、 定数はエントリ・スクリプトの冒頭で定義されなければなりません。

3.3 アプリケーション

アプリケーションは Yii アプリケーション・システム全体の構造とライフサイクルを統制するオブジェクトです。 全ての Yii アプリケーション・システムは、それぞれ、単一のアプリケーション・オブジェクトを持ち

ます。 アプリケーション・オブジェクトは、エントリ・スクリプト において作成され、\Yii::$app という式でグローバルにアクセスすることが出来るオブジェクトです。

> 情報: ガイドの中で「アプリケーション」という言葉は、文脈に応じて、アプリケーション・オブジェクトを意味したり、アプリケーション・システムを意味したりします。

二種類のアプリケーション、すなわち、ウェブ・アプリケーション と コンソール・アプリケーション があります。 名前が示すように、前者は主にウェブのリクエストを処理し、後者はコンソール・コマンドのリクエストを処理します。

3.3.1 アプリケーションの構成情報

エントリ・スクリプト は、アプリケーションを作成するときに、下記のように、 構成情報 を読み込んで、それをアプリケーションに適用します。

```
require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// アプリケーションの構成情報を読み込む
$config = require __DIR__ . '/../config/web.php';

// アプリケーションのインスタンスを作成し、構成情報を適用する
(new yii\web\Application($config))->run();
```

通常の 構成情報 と同じように、アプリケーションの構成情報は、 アプリケーション・オブジェクトのプロパティをどのように初期化するかを指定するものです。 アプリケーションの構成情報は、たいていは非常に複雑なものですから、通常は、上記の例の web.php ファイルのように、 構成情報ファイル に保管されます。

3.3.2 アプリケーションのプロパティ

アプリケーションの構成情報で構成すべき重要なアプリケーションのプロパティは数多くあります。 それらのプロパティの典型的なものは、アプリケーションが走る環境を記述するものです。 例えば、アプリケーションは、どのようにして コントローラ をロードするか、また、どこにテンポラリフ・ァイルを保存するかなどを知らなければなりません。 以下において、それらのプロパティを要約します。

必須のプロパティ

どのアプリケーションでも、最低二つのプロパティは構成しなければなりません。 すなわち、id と basePath です。

`id` id プロパティは、アプリケーションを他のアプリケーションから区別するユニークな ID を指定するものです。 このプロパティは主としてプログラム内部で使われます。 必須ではありませんが、最良の相互運用性を確保するために、アプリケーション ID を指定するときには英数字だけを使うことが推奨されます。

`basePath` basePath プロパティは、アプリケーションのルート・ディレクトリを指定するものです。 これは、アプリケーション・システムの全ての保護されたソース・コードを収容するディレクトリです。 通常、このディレクトリの下に、MVC パターンに対応するソース・コードを収容した `models`、`views`、`controllers` などのサブ・ディレクトリがあります。

basePath プロパティの構成には、ディレクトリ・パスを使っても、パス・エイリアス を使っても構いません。 どちらの形式においても、対応するディレクトリが存在しなければなりません。 さもなくば、例外が投げられます。パスは `realpath()` 関数を呼び出して正規化されます。

basePath プロパティは、しばしば、他の重要なパス (例えば、runtime のパス) を派生させるために使われます。 このため、`basePath` を示す `@app` というパス・エイリアスが、あらかじめ定義されています。 その結果、派生的なパスはこのエイリアスを使って形成することが出来ます(例えば、runtime ディレクトリを示す `@app/runtime` など)。

重要なプロパティ

この項で説明するプロパティは、アプリケーションごとに異なるものであるため、 構成する必要がよく生じるものです。

`aliases` このプロパティを使って、配列形式で一連の エイリアス を定義することが出来ます。 配列のキーがエイリアスの名前であり、配列の値が対応するパスの定義です。 例えば、

```
[
    'aliases' => [
        '@name1' => 'path/to/path1',
        '@name2' => 'path/to/path2',
    ],
]
```

このプロパティが提供されているのは、`Yii::setAlias()` メソッドを呼び出す代りに、 アプリケーションの構成情報を使ってエイリアスを定義することが出来るようにするためです。

`bootstrap` これは非常に有用なプロパティです。 これによって、アプリケーションの ブートストラップの過程 において走らせるべきコンポーネントを配列として指定することが出来ます。 例えば、ある モジュール

に URL 規則 をカスタマイズさせたいときに、 モジュールの ID をこのプロパティの要素として挙げることが出来ます。

このプロパティにリストする各コンポーネントは、以下の形式のいずれかによって指定することが出来ます。

- components によって指定されているアプリケーション・コンポーネントの ID
- modules によって指定されているモジュールの ID
- クラス名
- 構成情報の配列
- コンポーネントを作成して返す無名関数

例えば、

```
[
    'bootstrap' => [
        // アプリケーション・コンポーネント 、または、モジュールID ID
        'demo',

        // クラス名
        'app\components\Profiler',

        // 構成情報の配列
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ],

        // 無名関数
        function () {
            return new app\components\Profiler();
        }
    ],
]
```

> 情報: モジュール ID と同じ ID のアプリケーション・コンポーネントがある場合は、 ブートストラップの過程ではアプリケーション・コンポーネントが使われます。 代りにモジュールを使いたいときは、次のように、無名関数を使って指定することが出来ます。
>
> ```
> [
>     function () {
>         return Yii::$app->getModule('user');
>     },
> ]
> ```

ブートストラップの過程で、各コンポーネントのインスタンスが作成されます。 そして、コンポーネント・クラスが yii\base\BootstrapInterface を実装している場合は、 その bootstrap() メソッドも呼び出されます。

もう一つの実用的な例が ベーシック・プロジェクト・テンプレート のアプリケーションの構成情報の中にあります。 そこでは、アプリケーションが開発環境で走るときには debug モジュールと gii モジュールがブートストラップ・コンポーネントとして構成されています。

```
if (YII_ENV_DEV) {
    // 'dev' 環境のための構成情報の修正
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

> 補足: あまり多くのコンポーネントを bootstrap に置くと、アプリケーションのパフォーマンスを劣化させます。 なぜなら、リクエストごとに同じ一連のコンポーネントを走らせなければならないからです。ですから、ブートストラップ・コンポーネントは賢く使ってください。

catchAll このプロパティは ウェブ・アプリケーション においてのみサポートされます。 これは、全てのユーザ・リクエストを処理すべき コントローラ・アクション を指定するものです。 これは主としてアプリケーションがメンテナンス・モードにあって、入ってくる全てのリクエストを単一のアクションで処理する必要があるときに使われます。

構成情報は配列の形を取り、最初の要素はアクションのルートを指定します。 そして、配列の残りの要素 (キー・値のペア) は、アクションに渡されるパラメータを指定します。例えば、

```
[
    'catchAll' => [
        'offline/notice',
        'param1' => 'value1',
        'param2' => 'value2',
    ],
]
```

> 情報: このプロパティを有効にすると、開発環境でデバッグ・パネルが動作しなくなります。

components これこそが、唯一の最も重要なプロパティです。 これによって、アプリケーション・コンポーネント と呼ばれる一連の名前付きのコンポーネントを登録して、それらを他の場所で使うことが出来るようになります。例えば、

```
[
    'components' => [
        'cache' => [
```

```
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
    ],
]
```

全てのアプリケーション・コンポーネントは、それぞれ、配列の中で「キー・値」のペアとして指定されます。 キーはコンポーネントの ID を示し、値はコンポーネントのクラス名または 構成情報 を示します。

どのようなコンポーネントでもアプリケーションに登録することが出来ます。 そして登録されたコンポーネントは、後で、`\Yii::$app->componentID` という式を使ってグローバルにアクセスすることが出来ます。

詳細は アプリケーション・コンポーネント のセクションを読んでください。

`controllerMap` このプロパティは、コントローラ ID を任意のコントローラ・クラスに割り付けることを可能にするものです。 デフォルトでは、Yii は 規約 に基づいてコントローラ ID をコントローラ・クラスに割り付けます (例えば、`post` という ID は `app\controllers\PostController` に割り付けられます)。 このプロパティを構成することによって、特定のコントローラに対する規約を破ることが出来ます。 下記の例では、`account` は `app\controllers\UserController` に割り付けられ、`article` は `app\controllers\PostController` に割り付けられることになります。

```
[
    'controllerMap' => [
        'account' => 'app\controllers\UserController',
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

このプロパティの配列のキーはコントローラ ID を表し、配列の値は対応するコントローラ・クラスの名前または 構成情報 を表します。

`controllerNamespace` このプロパティは、コントローラ・クラスが配置されるべきデフォルトの名前空間を指定するものです。 デフォルト値は `app\controllers` です。 コントローラ ID が `post` である場合、規約によって対応するコントローラの (名前空間を略した) クラス名は `PostController` となり、完全修飾クラス名は `app\controllers\PostController` となります。

コントローラ・クラスは、この名前空間に対応するディレクトリのサブ・ディレクトリに配置されても構いません。 例えば、コントローラ ID

として admin/post を仮定すると、対応するコントローラの完全修飾クラス名は app\controllers\admin\PostController となります。

重要なことは、完全修飾されたコントローラ・クラスが オートロード可能 でなければならず、 コントローラ・クラスの実際の名前空間がこのプロパティと合致していなければならない、ということです。 そうでないと、アプリケーションにアクセスしたときに “ページがみつかりません” というエラーを受け取ることになります。

上で説明された規約を破りたい場合は、 controllerMap プロパティを構成することが出来ます。

language このプロパティは、アプリケーションがコンテントをエンド・ユーザに表示するときに使うべき言語を指定するものです。 このプロパティのデフォルト値は en であり、英語を意味します。 アプリケーションが多言語をサポートする必要があるときは、このプロパティを構成しなければなりません。

このプロパティの値が、メッセージの翻訳、日付の書式、数字の書式などを含む 国際化 のさまざまな側面を決定します。 例えば、yii\jui\DatePicker ウィジェットは、どの言語でカレンダーを表示すべきか、そして日付をどのように書式設定すべきかを、デフォルトでは、 このプロパティを使用して決定します。

言語を指定するのには、IETF 言語タグ[3] に従うことが推奨されます。 例えば、en は英語を意味し、en-US はアメリカ合衆国の英語を意味します。

このプロパティに関する詳細は 国際化 のセクションで読むことが出来ます。

modules このプロパティはアプリケーションが含む モジュール を指定するものです。

このプロパティは、モジュールの ID をキーとする、モジュールのクラスまたは 構成情報 の配列です。 例えば、

```
[
    'modules' => [
        // モジュール・クラスで指定された "booking" モジュール
        'booking' => 'app\modules\booking\BookingModule',

        // 構成情報の配列で指定された "comment" モジュール
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

詳細は モジュール のセクションを参照してください。

[3] http://ja.wikipedia.org/wiki/IETF%E8%A8%80%E8%AA%9E%E3%82%BF%E3%82%B0

name　このプロパティは、エンド・ユーザに対して表示されるアプリケーション名を指定するものです。 id プロパティがユニークな値を取らなければならないのとは違って、 このプロパティの値は主として表示目的であり、ユニークである必要はありません。

コードのどこにも使わないのであれば、このプロパティは必ずしも構成する必要はありません。

params　このプロパティは、グローバルにアクセス可能なアプリケーション・パラメータの配列を指定するものです。 コードの中のいたる処でハードコードされた数値や文字列を使う代りに、それらをアプリケーション・パラメータとして一ヶ所で定義し、 必要な場所ではそのパラメータを使うというのが良いプラクティスです。 例えば、次のように、サムネール画像のサイズをパラメータとして定義することが出来ます。

```
[
    'params' => [
        'thumbnail.size' => [128, 128],
    ],
]
```

そして、このサイズの値を使う必要があるコードにおいては、下記のようなコードを使うだけで済ませることが出来ます。

```
$size = \Yii::$app->params['thumbnail.size'];
$width = \Yii::$app->params['thumbnail.size'][0];
```

後でサムネールのサイズを変更すると決めたときは、アプリケーションの構成情報においてのみサイズを修正すればよく、 これに依存するコードには少しも触れる必要がありません。

sourceLanguage　このプロパティはアプリケーション・コードが書かれている言語を指定するものです。デフォルト値は 'en-US'、アメリカ合衆国の英語です。 あなたのコードのテキストのコンテントが英語以外で書かれているときは、このプロパティを構成しなければなりません。

language プロパティと同様に、このプロパティは IETF 言語タグ[4] に従って構成しなければなりません。 例えば、en は英語を意味し、en-US はアメリカ合衆国の英語を意味します。

このプロパティに関する詳細は 国際化 のセクションで読むことが出来ます。

timeZone　このプロパティは、PHP ランタイムのデフォルト・タイム・ゾーンを設定する代替手段として提供されています。 このプロパティを構成することによって、本質的には PHP 関数 date_default_timezone_set()[5] を呼び出すことになります。 例えば、

[4] http://ja.wikipedia.org/wiki/IETF%E8%A8%80%E8%AA%9E%E3%82%BF%E3%82%B0

[5] https://secure.php.net/manual/ja/function.date-default-timezone-set.php

```
[
    'timeZone' => 'Asia/Tokyo',
]
```

タイム・ゾーンを設定することの意味合いについては、日付のフォーマッティングのセクション で詳細を参照して下さい。

`version` このプロパティはアプリケーションのバージョンを指定するものです。デフォルト値は `'1.0'` です。 コードの中で全く使わないのであれば、必ずしも構成する必要はありません。

有用なプロパティ

この項で説明されるプロパティは通常は構成されません。というのは、そのデフォルト値が通常の規約に由来するものであるからです。 しかしながら、規約を破る必要がある場合には、これらのプロパティを構成することが出来ます。

`charset` このプロパティはアプリケーションが使う文字セットを指定するものです。 デフォルト値は `'UTF-8'` であり、多数の非ユニコード・データを使うレガシー・システムを扱っている場合を除けば、たいていのアプリケーションでは、そのままにしておくべきです。

`defaultRoute` このプロパティは、リクエストがルートを指定していないときにアプリケーションが使用すべき ルート を指定するものです。ルートは、チャイルド・モジュール ID、コントローラ ID、および/または アクション ID を構成要素とすることが出来ます。 例えば、`help`、`post/create`、`admin/post/create` などです。 アクション ID が与えられていない場合は、`yii\base\Controller::$defaultAction` で指定されるデフォルト値を取ります。

ウェブ・アプリケーション では、このプロパティのデフォルト値は `'site'` であり、 その意味するところは、`SiteController` コントローラとそのデフォルト・アクションが使用されるべきである、ということです。 結果として、ルートを指定せずにアプリケーションにアクセスすると、`app\controllers\SiteController::actionIndex()` の結果が表示されます。

コンソール・アプリケーション では、デフォルト値は `'help'` であり、 コア・コマンドの `yii\console\controllers\HelpController::actionIndex()` が使用されるべきであるという意味です。 結果として、何も引数を与えずに `yii` というコマンドを実行すると、ヘルプ情報が表示されることになります。

`extensions` このプロパティは、アプリケーションにインストールされて使われている エクステンション のリストを指定するものです。 デフォルトでは、`@vendor/yiisoft/extensions.php` というファイルによって返され

る配列を取ります。 extensions.php は、Composer[6] を使ってエクステンションをインストールすると、自動的に生成され保守されます。 ですから、たいていの場合、このプロパティをあなたが構成する必要はありません。

エクステンションを手作業で保守したいという特殊なケースにおいては、次のようにしてこのプロパティを構成することが出来ます。

```
[
    'extensions' => [
        [
            'name' => 'extension name',
            'version' => 'version number',
            'bootstrap' => 'BootstrapClassName',  // オプション、構成情報の配
    列でもよい
            'alias' => [  // optional
                '@alias1' => 'to/path1',
                '@alias2' => 'to/path2',
            ],
        ],

        // ... 上記と同じように、更にエクステンションを構成 ...

    ],
]
```

ご覧のように、このプロパティはエクステンションの仕様を示す配列を取ります。 それぞれのエクステンションは、name と version の要素を含む配列によって指定されます。 エクステンションが ブートストラップ の過程で走る必要がある場合には、bootstrap 要素をブートストラップのクラス名または 構成情報 の配列によって指定することが出来ます。 また、エクステンションはいくつかの エイリアス を定義することも出来ます。

layout　このプロパティは、ビュー をレンダリングするときに使われるべきデフォルトのレイアウトを指定するものです。 デフォルト値は 'main' であり、レイアウト・パス の下にある main.php というファイルが使われるべきことを意味します。 レイアウト・パス と ビュー・パス の両方がデフォルト値を取る場合、デフォルトのレイアウト・ファイルは @app/views/layouts/main.php というパス・エイリアスとして表すことが出来ます。

滅多には無いことですが、レイアウトをデフォルトで無効にしたい場合は、このプロパティを false として構成することが出来ます。

layoutPath　このプロパティは、レイアウト・ファイルが捜されるべきパスを指定するものです。 デフォルト値は、ビュー・パス の下の layouts サブ・ディレクトリです。 ビュー・パス がデフォルト値を取る場合、デ

[6] https://getcomposer.org

フォルトのレイアウト・パスは `@app/views/layouts` というパス・エイリアスとして表すことが出来ます。

このプロパティはディレクトリまたはパス・エイリアス として構成することが出来ます。

`runtimePath` このプロパティは、ログ・ファイルやキャッシュ・ファイルなどの一時的ファイルを生成することが出来るパスを指定するものです。 デフォルト値は、`@app/runtime` というエイリアスで表現されるディレクトリです。

このプロパティはディレクトリまたはパス・エイリアス として構成することが出来ます。 ランタイムパスは、アプリケーションを実行するプロセスによって書き込みが可能なものでなければならないことに注意してください。 そして、この下にある一時的ファイルは秘匿を要する情報を含みうるものですので、ランタイム・パスはエンド・ユーザによるアクセスから保護されなければなりません。

このパスに簡単にアクセスできるように、Yii は `@runtime` というパス・エイリアスを事前に定義しています。

`viewPath` このプロパティはビュー・ファイルが配置されるルート・ディレクトリを指定するものです。デフォルト値は、`@app/views` というエイリアスで表現されるディレクトリです。 このプロパティはディレクトリまたはパス・エイリアス として構成することが出来ます。

`vendorPath` このプロパティは、Composer[7] によって管理される vendor ディレクトリを指定するものです。 Yii フレームワークを含めて、あなたのアプリケーションによって使われる全てのサード・パーティ・ライブラリを格納するディレクトリです。 デフォルト値は、`@app/vendor` というエイリアスで表現されるディレクトリです。

このプロパティはディレクトリまたはパス・エイリアス として構成することが出来ます。 このプロパティを修正するときは、必ず、Composer の構成もそれに合せて修正してください。

このパスに簡単にアクセスできるように、Yii は `@vendor` というパス・エイリアスを事前に定義しています。

`enableCoreCommands` このプロパティは コンソール・アプリケーション においてのみサポートされています。 Yii リリースに含まれているコア・コマンドを有効にすべきか否かを指定するものです。デフォルト値は `true` です。

3.3.3 アプリケーションのイベント

アプリケーションはリクエストを処理するライフサイクルの中でいくつ

[7] https://getcomposer.org

かのイベントをトリガします。 これらのイベントに対して、下記のようにして、アプリケーションの構成情報の中でイベント・ハンドラをアタッチすることが出来ます。

```
[
    'on beforeRequest' => function ($event) {
        // ...
    },
]
```

on eventName という構文の使い方については、構成情報 のセクションで説明されています。

別の方法として、アプリケーションのインスタンスが生成された後、ブートストラップの過程 の中でイベント・ハンドラをアタッチすることも出来ます。 例えば、

```
\Yii::$app->on(\yii\base\Application::EVENT_BEFORE_REQUEST, function ($event
    ) {
    // ...
});
```

EVENT_BEFORE_REQUEST

このイベントは、アプリケーションがリクエストを処理する 前 にトリガされます。実際のイベント名は beforeRequest です。

このイベントがトリガされるときには、アプリケーションのインスタンスは既に構成されて初期化されています。 ですから、イベント・メカニズムを使って、リクエスト処理のプロセスに干渉するカスタム・コードを挿入するのには、ちょうど良い場所です。 例えば、このイベント・ハンドラの中で、何らかのパラメータに基づいて yii\base\Application ::$language プロパティを動的にセットすることが出来ます。

EVENT_AFTER_REQUEST

このイベントは、アプリケーションがリクエストの処理を完了した 後、レスポンスを送信する 前 にトリガされます。 実際のイベント名は afterRequest です。

このイベントがトリガされるときにはリクエストの処理は完了していますので、この機をとらえて、 リクエストに対する何らかの後処理をしたり、レスポンスをカスタマイズしたりすることが出来ます。

response コンポーネントも、エンド・ユーザにレスポンスのコンテントを送出する間にいくつかのイベントをトリガすることに注意してください。 それらのイベントは、このイベントの 後 にトリガされます。

EVENT_BEFORE_ACTION

このイベントは、コントローラ・アクション を実行する 前 に毎回トリガされます。 実際のイベント名は beforeAction です。

イベントのパラメータは yii\base\ActionEvent のインスタンスです。 イベント・ハンドラは、yii\base\ActionEvent::$isValid プロパティを false にセットして、アクションの実行を中止することが出来ます。 例えば、

```
[
    'on beforeAction' => function ($event) {
        if 何らかの条件() {
            $event->isValid = false;
        } else {
        }
    },
]
```

同じ beforeAction イベントが、モジュール と コントローラ からもトリガされることに注意してください。 アプリケーション・オブジェクトが最初にこのイベントをトリガし、次に (もし有れば) モジュールが、 そして最後にコントローラがこのイベントをトリガします。 いずれかのイベント・ハンドラが yii\base\ActionEvent::$isValid を false にセットすると、後続のイベントはトリガされません。

EVENT_AFTER_ACTION

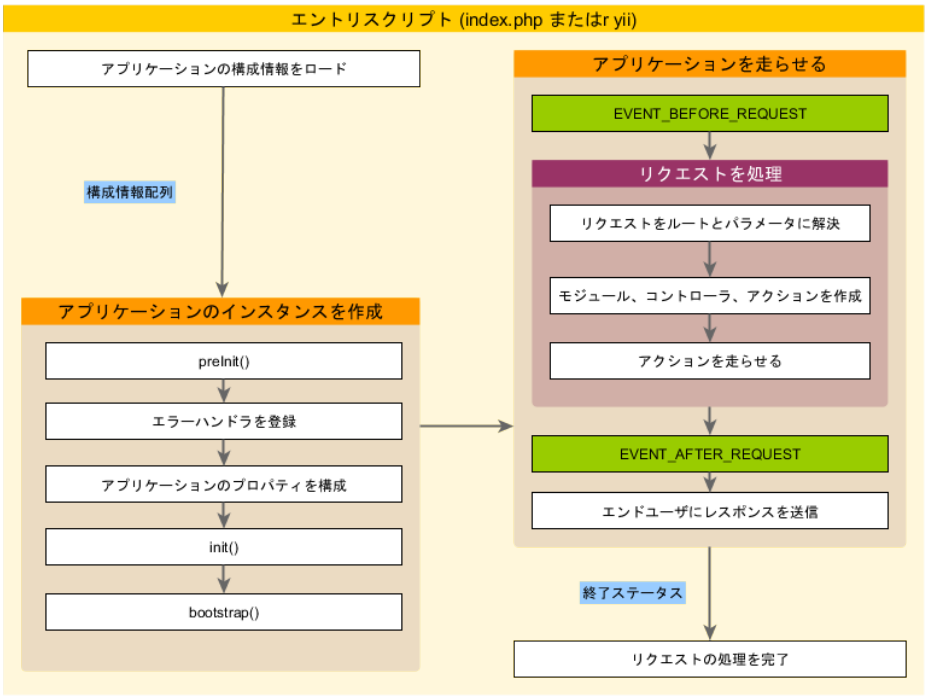
このイベントは、コントローラ・アクション を実行した 後 に毎回トリガされます。 実際のイベント名は afterAction です。

イベントのパラメータは yii\base\ActionEvent のインスタンスです。 yii\base\ActionEvent::$result プロパティを通じて、イベント・ハンドラはアクションの結果にアクセスしたり、またはアクションの結果を修正したり出来ます。 例えば、

```
[
    'on afterAction' => function ($event) {
        if 何らかの条件() {
            // $event->result を修正する
        } else {
        }
    },
]
```

同じ afterAction イベントが、モジュール と コントローラ からもトリガされることに注意してください。 これらのオブジェクトは、beforeAction の場合とは逆の順でイベントをトリガします。 すなわち、コントローラ・オブジェクトが最初にこのイベントをトリガし、次に (もし有れば) モジュールが、 そして最後にアプリケーションがこのイベントをトリガします。

3.3.4 アプリケーションのライフサイクル



エントリ・スクリプト が実行されて、リクエストが処理されるとき、アプリケーションは次のようなライフサイクルを経ます。

1. エントリ・スクリプトがアプリケーションの構成情報を配列として読み出す。
2. エントリ・スクリプトがアプリケーションの新しいインスタンスを作成する。
   - `preInit()` が呼び出されて、`basePath` のような、 優先度の高いアプリケーション・プロパティを構成する。
   - エラー・ハンドラ を登録する。
   - アプリケーションのプロパティを構成する。
   - `init()` が呼ばれ、そこから更に、ブートストラップ・コンポーネントを走らせるために、 `bootstrap()` が呼ばれる。
3. エントリ・スクリプトが `yii\base\Application::run()` を呼んで、アプリケーションを走らせる。
   - `EVENT_BEFORE_REQUEST` イベントをトリガする。
   - リクエストを処理する: リクエストを ルート とそれに結び付くパラメータとして解決する。 ルートによって指定されたモジュール、コントローラ、および、アクションを作成する。そしてアクションを実行する。

- EVENT_AFTER_REQUEST イベントをトリガする。
- エンド・ユーザにレスポンスを送信する。

4. エントリ・スクリプトがアプリケーションから終了ステータスを受け取り、リクエストの処理を完了する。

3.4 アプリケーション・コンポーネント

アプリケーションは サービス・ロケータ です。 アプリケーションは、リクエストを処理するためのいろいろなサービスを提供する一組の アプリケーション・コンポーネント と呼ばれるものをホストします。 例えば、urlManager がウェブ・リクエストを適切なコントローラにルーティングする役割を負い、 db コンポーネントが DB 関連のサービスを提供する、等々です。

全てのアプリケーション・コンポーネントは、それぞれ、同一のアプリケーション内で他のアプリケーション・コンポーネントから区別できるように、ユニークな ID を持ちます。 アプリケーション・コンポーネントには、次の式によってアクセス出来ます。

```
\Yii::$app->componentID
```

例えば、\Yii::$app->db を使って、アプリケーションに登録された DB 接続 を取得することが出来ます。 また、\Yii::$app->cache を使って、プライマリ・キャッシュ を取得できます。

アプリケーション・コンポーネントは、上記の式を使ってアクセスされた最初の時に作成されます。 二度目以降のアクセスでは、同じコンポーネント・インスタンスが返されます。

どのようなオブジェクトでも、アプリケーション・コンポーネントとすることが可能です。 アプリケーションの構成情報 の中で yii\base\Application::$components プロパティを構成することによって、アプリケーション・コンポーネントを登録することが出来ます。 例えば、

```
[
    'components' => [
        // クラス名を使って "cache" コンポーネントを登録
        'cache' => 'yii\caching\ApcCache',

        // 構成情報の配列を使って "db" コンポーネントを登録
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // 無名関数を使って "search" コンポーネントを登録
        'search' => function () {
            return new app\components\SolrService;
```

```
        },
    ],
]
```

> 情報: 必要なだけ多くのアプリケーション・コンポーネントを登録することが出来ますが、慎重にしなければなりません。アプリケーション・コンポーネントはグローバル変数のようなものです。 あまり多くのアプリケーション・コンポーネントを使うと、コードのテストと保守が困難になるおそれがあります。 多くの場合、必要なときにローカルなコンポーネントを作成して使用するだけで十分です。

3.4.1 コンポーネントをブートストラップに含める

上述のように、アプリケーション・コンポーネントは最初にアクセスされた時に初めてインスタンスが作成されます。 リクエストの間に全くアクセスされなかった時は、インスタンスは作成されません。 けれども、場合によっては、明示的にアクセスされないときでも、リクエストごとにアプリケーション・コンポーネントのインスタンスを作成したいことがあります。 そうするために、アプリケーションの `bootstrap` プロパティのリストにそのコンポーネントの ID を挙げることが出来ます。

また、カスタマイズしたコンポーネントをブートストラップするためにクロージャを用いることも出来ます。インスタンス化されたコンポーネントを返すことは要求されません。 単に `yii\base\Application` のインスタンス化の後にコードを走らせるだけのためにクロージャを使うことも出来ます。

例えば、次のアプリケーション構成情報は、`log` コンポーネントが常にロードされることを保証するものです。

```
[
    'bootstrap' => [
        'log',
        function($app){
            return new ComponentX();
        },
        function($app){
            // 何らかのコード
           return;
        }
    ],
    'components' => [
        'log' => [
            // "log" コンポーネントの構成情報
        ],
    ],
]
```

3.4.2 コア・アプリケーション・コンポーネント

Yii は固定の ID とデフォルトの構成情報を持つ一連の コア・アプリケーション・コンポーネントを定義しています。 例えば、request コンポーネントは、ユーザ・リクエストに関する情報を収集して、 それを ルート として解決するために使用されます。 また、db コンポーネントは、それを通じてデータ・ベースクエリを実行できるデータベース接続を表現します。 Yii のアプリケーションがユーザ・リクエストを処理出来るのは、まさにこれらのコア・アプリケーション・コンポーネントの助けがあってこそです。

下記が事前に定義されたコア・アプリケーション・コンポーネントです。 通常のアプリケーション・コンポーネントに対するのと同様に、これらを構成し、カスタマイズすることが出来ます。 コア・アプリケーション・コンポーネントを構成するときは、クラスを指定しない場合は、デフォルトのクラスが使用されます。

- assetManager: アセット・バンドルとアセットの発行を管理します。 詳細は アセット のセクションを参照してください。
- db: データベース接続を表します。これを通じて、DB クエリを実行することが出来ます。 このコンポーネントを構成するときは、コンポーネントのクラスはもちろん、yii\db\Connection::$dsn のような必須のコンポーネント・プロパティを指定しなければならないことに注意してください。 詳細は データベース・アクセス・オブジェクト のセクションを参照してください。
- errorHandler: PHP のエラーと例外を処理します。 詳細は エラー処理 のセクションを参照してください。
- formatter: エンド・ユーザに表示されるデータに書式を設定します。 例えば、数字が3桁ごとの区切りを使って表示されたり、日付が長い書式で表示されたりします。 詳細は データの書式設定 のセクションを参照してください。
- i18n: メッセージの翻訳と書式設定をサポートします。 詳細は 国際化 のセクションを参照してください。
- log: ログ・ターゲットを管理します。 詳細は ロギング のセクションを参照してください。
- yii\swiftmailer\Mailer: メールの作成と送信をサポートします。詳細は メール のセクションを参照してください。
- response: エンド・ユーザに送信されるレスポンスを表現します。詳細は レスポンス のセクションを参照してください。
- request: エンド・ユーザから受信したリクエストを表現します。詳細は リクエスト のセクションを参照してください。
- session: セッション情報を表現します。 このコンポーネントは、ウェブ・アプリケーション においてのみ利用できます。 詳細は セッションとクッキー のセクションを参照してください。
- urlManager: URL の解析と生成をサポートします。 詳細は ルーティング と URL 生成 のセクションを参照してください。

- user: ユーザの認証情報を表現します。 このコンポーネントは、ウェブ・アプリケーション においてのみ利用できます。 詳細は 認証 のセクションを参照してください。
- view: ビューのレンダリングをサポートします。 詳細は ビュー のセクションを参照してください。

3.5 コントローラ

コントローラは MVC[8] アーキテクチャの一部を成すものです。 それは yii\base\Controller を拡張したクラスのオブジェクトであり、リクエストの処理とレスポンスの生成について責任を負います。 具体的には、コントローラは、アプリケーション から制御を引き継いだ後、 入ってきたリクエストのデータを分析し、それを モデル に引き渡して、 モデルが生成した結果を ビュー に投入し、最終的に外に出て行くレスポンスを生成します。

3.5.1 アクション

コントローラは、エンド・ユーザがアドレスを指定して実行をリクエストできる最も基本的なユニットである アクション から構成されます。 コントローラは一つまたは複数のアクションを持つことが出来ます。

次の例は、view と create という二つのアクションを持つ post コントローラを示すものです。

```
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        return $this->render('view', [
            'model' => $model,
        ]);
    }

    public function actionCreate()
    {
```

[8] http://ja.wikipedia.org/wiki/Model_View_Controller

```
        $model = new Post;

        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('create', [
                'model' => $model,
            ]);
        }
    }
}
```

view アクション (actionView() メソッドで定義されます) において、コードは最初に、リクエストされたモデルの ID に従って モデル を読み出します。 モデルの読み出しが成功したときは、view という名前の ビュー を使ってモデルを表示します。 失敗したときは例外を投げます。

create アクション (actionCreate() メソッドで定義されます) においても、コードは似たようなものです。 最初に、リクエスト・データを使って モデル の新しいインスタンスにデータを投入することを試み、そして、モデルを保存することを試みます。 両方が成功したときは、新しく作成されたモデルの ID を使って view アクションにブラウザをリダイレクトします。 どちらかが失敗したときは、ユーザが必要なデータを入力できるようにするための create ビューを表示します。

3.5.2 ルート

エンド・ユーザは、いわゆる ルート によって、アクションを指定します。ルートは、次の部分からなる文字列です。

- モジュール ID: この部分は、コントローラがアプリケーションではない モジュール に属する場合にのみ存在します。

コントローラ ID ((#controller-ids): 同じアプリケーション (または、コントローラがモジュールに属する場合は、同じモジュール) に属する全てのコントローラの中から、コントローラを一意に特定する文字列。

- アクション ID: 同じコントローラに属する全てのアクションの中から、アクションを一意に特定する文字列。

ルートは次の形式を取ります。

```
ControllerID/ActionID
```

または、コントローラがモジュールに属する場合は、次の形式を取ります。

```
ModuleID/ControllerID/ActionID
```

ですから、ユーザが http://hostname/index.php?r=site/index という URL でリクエストをした場合は、 site コントローラの中の index アクションが実行されます。 ルートがどのようにしてアクションとして解決されるかについての詳細は、ルーティングと URL 生成 のセクションを参照してください。

3.5.3 コントローラを作成する

ウェブ・アプリケーション では、コントローラは yii\web\Controller またはその子クラスから派生させなければなりません。 同様に、コンソール・アプリケーション では、コントローラは yii\console\Controller またはその子クラスから派生させなければなりません。次のコードは site コントローラを定義するものです。

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}
```

コントローラ ID

通常、コントローラは特定のタイプのリソースに関するリクエストを処理するように設計されます。 この理由により、たいていは、処理するリソースのタイプを示す名詞をコントローラの ID として使います。 例えば、記事データを処理するコントローラの ID としては、article を使うことが出来ます。

デフォルトでは、コントローラ ID は、小文字の英字、数字、アンダースコア、ダッシュ、および、フォワード・スラッシュのみを含むべきものです。 例えば、article と post-comment はともに有効なコントローラ ID ですが、 article?、PostComment、admin\post はそうではありません。

コントローラ ID は、サブ・ディレクトリの接頭辞を含んでも構いません。 例えば、admin/article は、コントローラ名前空間 の下の admin サブ・ディレクトリにある article コントローラを表します。 サブ・ディレクトリの接頭辞として有効な文字は、小文字または大文字の英字、数字、アンダースコア、そして、フォワード・スラッシュです。 フォワード・スラッシュは、複数レベルのサブ・ディレクトリの区切り文字として使われます (例えば、panels/admin)。

コントローラ・クラスの命名規則

コントローラ・クラスの名前は下記の手順に従ってコントローラ ID から導出することが出来ます。

1. ハイフンで区切られた各単語の最初の文字を大文字に変える。 コントローラ ID がスラッシュを含む場合、この規則は ID の最後のスラッシュの後ろの部分にのみ適用されることに注意。

2. ハイフンを削除し、フォワード・スラッシュを全てバックワード・スラッシュに置き換える。

3. 接尾辞 `Controller` を追加する。

4. コントローラ名前空間 を頭に付ける。

以下は、コントローラ名前空間 がデフォルト値 `app\controllers` を取っていると仮定したときの、いくつかの例です。

- `article` は `app\controllers\ArticleController` になる。
- `post-comment` は `app\controllers\PostCommentController` になる。
- `admin/post-comment` は `app\controllers\admin\PostCommentController` になる。
- `adminPanels/post-comment` は `app\controllers\adminPanels\PostCommentController` になる。

コントローラ・クラスは オートロード可能 でなければなりません。 この理由により、上記の例の `aritcle` コントローラ・クラスは エイリアス が `@app/controllers/ArticleController.php` であるファイルに保存されるべきものとなります。 一方、`admin/post-comment` コントローラは `@app/controllers/admin/PostCommentController.php` というエイリアスのファイルに保存されるべきものとなります。

> 情報: 最後の例である `admin/post-comment` は、どうすれば コントローラ名前空間 のサブ・ディレクトリにコントローラを置くことが出来るかを示しています。 この方法は、コントローラをいくつかのカテゴリに分けて編成したい、けれども モジュール は使いたくない、という場合に役立ちます。

コントローラ・マップ

コントローラ・マップ を構成すると、上で述べたコントローラ ID とクラス名の制約を乗り越えることが出来ます。 これは、主として、クラス名に対する制御が及ばないサード・パーティのコントローラを使おうとする場合に有用です。

コントローラ・マップ は アプリケーションの構成情報 の中で、次のように構成することが出来ます。

```
[
    'controllerMap' => [
        // クラス名を使って "account" コントローラを宣言する
        'account' => 'app\controllers\UserController',

        // 構成情報配列を使って "article" コントローラを宣言する
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

デフォルト・コントローラ

全てのアプリケーションは、それぞれ、yii\base\Application::$defaultRoute プロパティによって指定されるデフォルト・コントローラを持ちます。 リクエストが ルート を指定していない場合、このプロパティによって指定されたルートが使われます。 ウェブ・アプリケーション では、この値は 'site' であり、一方、コンソール・アプリケーション では、help です。 従って、URL が http://hostname/index.php である場合は、site コントローラがリクエストを処理することになります。

次のように アプリケーションの構成情報 を構成して、デフォルト・コントローラを変更することが出来ます。

```
[
    'defaultRoute' => 'main',
]
```

3.5.4 アクションを作成する

アクションは、コントローラ・クラスの中にいわゆる アクション・メソッド を定義するだけで簡単に作成することが出来ます。 アクション・メソッドとは、action という語で始まる名前を持つ *public* メソッドのことです。 アクション・メソッドの返り値がエンド・ユーザに送信されるレスポンス・データを表します。次のコードは、index と hello-world という二つのアクションを定義するものです。

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

    public function actionHelloWorld()
    {
        return 'Hello World';
    }
}
```

アクション ID

アクションは、たいてい、あるリソースについて特定の操作を実行するように設計されます。 この理由により、アクション ID は、通常、view 、update などのような動詞になります。

デフォルトでは、アクション ID は、小文字の英字、数字、アンダースコア、そして、ハイフンのみを含むべきものです。 アクション ID の

中のハイフンは単語を分けるために使われます。 例えば、view、update2、comment-post は全て有効なアクション ID ですが、view?、Update はそうではありません。

アクションは二つの方法、すなわち、インライン・アクションまたはスタンドアロン・アクションとして作成することが出来ます。 インライン・アクションはコントローラ・クラスのメソッドとして定義されるものであり、 一方、スタンドアロン・アクションは yii\base\Action またはその子クラスを拡張するクラスです。 インライン・アクションは作成するのにより少ない労力を要するため、通常は、アクションを再利用する意図がない場合に推奨されます。 もう一方のスタンドアロン・アクションは、主として、さまざまなコントローラの中で使われることや、エクステンション として再配布されることを目的として作成されます。

インライン・アクション

インライン・アクションは、たった今説明したように、アクション・メソッドの形で定義されるアクションを指します。

アクション・メソッドの名前は、次の手順に従って、アクション ID から導出されます。

1. アクション ID に含まれる各単語の最初の文字を大文字に変換する。
2. ハイフンを削除する。
3. 接頭辞 action を付ける。

例えば、index は actionIndex となり、hello-world は actionHelloWorld となります。

> 補足: アクション・メソッドの名前は、大文字と小文字を区別します。 ActionIndex という名前のメソッドがあっても、それはアクション・メソッドとは見なされず、結果として、index アクションに対するリクエストは例外に帰結します。 アクション・メソッドが public でなければならない事にも注意してください。 private や protected なメソッドがインライン・アクションを定義することはありません。

インライン・アクションは作成するのにほとんど労力を要さないため、たいていのアクションはインライン・アクションとして定義されます。しかし、同じアクションを別の場所で再利用する計画を持っていたり、また、アクションを再配布したいと思っていたりする場合は、 アクションを スタンドアロン・アクション として定義することを検討すべきです。

スタンドアロン・アクション

スタンドアロン・アクションは、yii\base\Action またはその子クラスを拡張するアクション・クラスの形で定義されるものです。 例えば、Yii のリリースに yii\web\ViewAction と yii\web\ErrorAction が含まれていますが、 これらは両方ともスタンドアロン・アクションです。

スタンドアロン・アクションを使用するためには、下記のように、コントローラの yii\base\Controller::actions() メソッドをオーバーライドして、 アクション・マップ の中でスタンドアロン・アクションを宣言しなければなりません。

```
public function actions()
{
    return [
        // クラス名を使って "error" アクションを宣言する
        'error' => 'yii\web\ErrorAction',

        // 構成情報配列を使って "view" アクションを宣言する
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

ご覧のように、actions() メソッドは、キーがアクション ID であり、値が対応するアクションのクラス名または 構成情報 である配列を返さなければなりません。 インライン・アクションと違って、スタンドアロン・アクションのアクション ID は、actions() メソッドにおいて宣言される限りにおいて、任意の文字を含むことが出来ます。

スタンドアロン・アクション・クラスを作成するためには、yii\base\Action またはその子クラスを拡張して、run() という名前の public メソッドを実装しなければなりません。 run() メソッドの役割はアクション・メソッドの役割と同じです。例えば、

```
<?php
namespace app\components;

use yii\base\Action;

class HelloWorldAction extends Action
{
    public function run()
    {
        return "Hello World";
    }
}
```

アクションの結果

アクション・メソッド、または、スタンドアロン・アクションの run() メ

ソッドの返り値は、重要な意味を持ちます。 それは、対応するアクションの結果を表すものです。

返り値は、エンド・ユーザにレスポンスとして送信される レスポンス オブジェクトとすることが出来ます。

- ウェブ・アプリケーション では、返り値を yii\web\Response::$data に割り当てられる任意のデータとすることも出来ます。 このデータは、後に、レスポンス・ボディを表す文字列へと変換されます。
- コンソール・アプリケーション では、返り値をコマンド実行の 終了ステータス を示す整数とすることも出来ます。

これまでに示した例においては、アクションの結果はすべて文字列であり、エンド・ユーザに送信されるレスポンス・ボディとして扱われるものでした。 次の例では、アクションがレスポンス・オブジェクトを返すことによって、ユーザのブラウザを 新しい URL にリダイレクトすることが出来る様子が示されています (redirect() メソッドの返り値はレスポンス・オブジェクトです)。

```
public function actionForward()
{
    // ユーザのブラウザを http://example.com にリダイレクトする
    return $this->redirect('http://example.com');
}
```

アクション・パラメータ

インライン・アクションのアクション・メソッドと、スタンドアロン・アクションの run() メソッドは、アクション・パラメータ と呼ばれるパラメータを取ることが出来ます。 パラメータの値はリクエストから取得されます。 ウェブ・アプリケーション では、各アクション・パラメータの値は $_GET からパラメータ名をキーとして読み出されます。 コンソール・アプリケーション では、アクション・パラメータはコマンドライン引数に対応します。

次の例では、view アクション (インライン・アクションです) は、二つのパラメータ、すなわち、$id と $version を宣言しています。

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public function actionView($id, $version = null)
    {
        // ...
    }
}
```

アクション・パラメータには、次のように、さまざまなリクエストに応じて異なる値が投入されます。

- http://hostname/index.php?r=post/view&id=123: $id パラメータには '123' という値が入れられます。 一方、version というクエリ・パラメータは無いので、$version は null のままになります。
- http://hostname/index.php?r=post/view&id=123&version=2: $id および $version パラメータに、 それぞれ、'123' と '2' が入ります。
- http://hostname/index.php?r=post/view: 必須の $id パラメータがリクエストで提供されていないため、 yii\web\BadRequestHttpException 例外が投げられます。
- http://hostname/index.php?r=post/view&id[]=123: $id パラメータが予期しない配列値 ['123'] を受け取ろうとするため、 yii\web\BadRequestHttpException 例外が投げられます。

アクション・パラメータに配列値を受け取らせたい場合は、次のように、パラメータに array の型ヒントを付けなければなりません。

```
public function actionView(array $id, $version = null)
{
    // ...
}
```

このようにすると、リクエストが http://hostname/index.php?r=post/view&id[]=123 である場合は、$id パラメータは ['123'] という値を受け取ります。 リクエストが http://hostname/index.php?r=post/view&id=123 である場合も、スカラ値 '123' が自動的に配列に変換されるため、 $id パラメータは引き続き同じ配列値を受け取ります。

上記の例は主としてウェブ・アプリケーションでのアクション・パラメータの動作を示すものです。 コンソール・アプリケーションについては、コンソール・コマンド のセクションで詳細を参照してください。

デフォルト・アクション

すべてのコントローラは、それぞれ、yii\base\Controller::$defaultAction によって指定されるデフォルト・アクションを持ちます。 ルート がコントローラ ID のみを含む場合は、 指定されたコントローラのデフォルト・アクションがリクエストされたことを意味します。

デフォルトでは、デフォルト・アクションは index と設定されます。 このデフォルト値を変更したい場合は、以下のように、コントローラ・クラスでこのプロパティをオーバーライドするだけです。

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $defaultAction = 'home';

    public function actionHome()
    {
        return $this->render('home');
```

```
    }
}
```

3.5.5 コントローラのライフサイクル

リクエストを処理するときに、アプリケーション はリクエストされたルート に基いてコントローラを作成します。 そして、次に、コントローラはリクエストに応じるために以下のライフサイクルを経過します。

1. コントローラが作成され構成された後、`yii\base\Controller::init()` メソッドが呼ばれる。
2. コントローラは、リクエストされたアクション ID に基いて、アクション・オブジェクトを作成する。
   - アクション ID が指定されていないときは、デフォルト・アクション `ID` が使われる。
   - アクション ID が アクション・マップ の中に見つかった場合は、 スタンドアロン・アクションが作成される。
   - アクション ID に合致するアクション・メソッドが見つかった場合は、インライン・アクションが作成される。
   - 上記以外の場合は、`yii\base\InvalidRouteException` 例外が投げられる。
3. コントローラは、アプリケーション、(コントローラがモジュールに属する場合は) モジュール、 そしてコントローラの `beforeAction()` メソッドをこの順で呼び出す。
   - どれか一つの呼び出しが `false` を返した場合は、残りのまだ呼ばれていない `beforeAction()` メソッドはスキップされ、 アクションの実行はキャンセルされる。
   - デフォルトでは、それぞれの `beforeAction()` メソッドは、ハンドラをアタッチすることが可能な `beforeAction` イベントをトリガする。
4. コントローラがアクションを実行する。
   - アクション・パラメータが解析されて、リクエスト・データからデータが投入される。
5. コントローラは、コントローラ、(コントローラがモジュールに属する場合は) モジュール、 そしてアプリケーションの `afterAction()` メソッドをこの順で呼び出す。
   - デフォルトでは、それぞれの `afterAction()` メソッドは、ハンドラをアタッチすることが可能な `afterAction` イベントをトリガする。
6. アプリケーションはアクションの結果を受け取り、それを レスポンス に割り当てる。

3.5.6 ベスト・プラクティス

良く設計されたアプリケーションでは、コントローラはたいてい非常に軽いものになり、 それぞれのアクションは数行のコードしか含まないものになります。 あなたのコントローラが少々複雑になっている場合、そのことは、通常、コントローラをリファクタして、コードの一部を他のクラスに移動すべきことを示すものです。

いくつかのベスト・プラクティスを特に挙げるなら、コントローラは、

- リクエスト データにアクセスすることが出来ます。
- リクエスト・データを使って モデル や他のサービス・コンポーネントのメソッドを呼ぶことが出来ます。
- ビュー を使ってレスポンスを構成することが出来ます。
- リクエストされたデータの処理をするべきではありません - データは モデルのレイヤ において処理されるべきです。
- HTML を埋め込むなどの表示に関わるコードは避けるべきです - 表示は ビュー で行う方が良いです。

3.6 モデル

モデルは MVC[9] アーキテクチャの一部を成すものです。 これは、ビジネスのデータ、規則、ロジックを表現するオブジェクトです。

モデル・クラスは、`yii\base\Model` またはその子クラスを拡張することによって作成することが出来ます。 基底クラス `yii\base\Model` は、次のような数多くの有用な機能をサポートしています。

- 属性: ビジネス・データを表現します。通常のオブジェクト・プロパティや配列要素のようにして アクセスすることが出来ます。
- 属性のラベル: 属性の表示ラベルを指定します。
- 一括代入: 一回のステップで複数の属性にデータを投入することをサポートしています。
- 検証規則: 宣言された検証規則に基いて入力されたデータの有効性を保証します。
- データのエクスポート: カスタマイズ可能な形式でモデル・データを配列にエクスポートすることが出来ます。

`Model` クラスは、アクティブ・レコード のような、更に高度なモデルの基底クラスでもあります。 それらの高度なモデルについての詳細は、関連するドキュメントを参照してください。

> 情報: あなたのモデル・クラスの基底クラスとして `yii\base\Model` を使うことが要求されている訳ではありません。 しかしながら、Yii のコンポーネントの多くが `yii\base\Model` をサポートするように作られていますので、通常は `yii\base\Model` がモデルの基底クラスとして推奨されます。

[9] http://ja.wikipedia.org/wiki/Model_View_Controller

3.6.1 属性

モデルはビジネス・データを 属性 の形式で表現します。全ての属性はそれぞれパブリックにアクセス可能なモデルのプロパティと同様なものです。 `yii\base\Model::attributes()` メソッドが、モデルがどのような属性を持つかを指定します。

属性に対しては、通常のオブジェクト・プロパティにアクセスするのと同じようにして、アクセスすることが出来ます。

```
$model = new \app\models\ContactForm;

// "name" は ContactForm の属性
$model->name = 'example';
echo $model->name;
```

また、配列の要素にアクセスするようして、属性にアクセスすることも出来ます。 これは、`yii\base\Model` が ArrayAccess インタフェイス[10] と Traversable インタフェイス[11] をサポートしている恩恵です。

```
$model = new \app\models\ContactForm;

// 配列要素のように属性にアクセスする
$model['name'] = 'example';
echo $model['name'];

// モデルは foreach で中身をたどることが出来る
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

属性を定義する

あなたのモデルが `yii\base\Model` を直接に拡張するものである場合、デフォルトでは、全ての *static* でない *public* な メンバ変数は属性となります。 例えば、次に示す `ContactForm` モデルは四つの属性、すなわち、`name`、`email`、`subject`、そして、`body` を持ちます。 この `ContactForm` モデルは、HTML フォームから受け取る入力データを表現するために使われます。

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
```

[10] https://secure.php.net/manual/ja/class.arrayaccess.php
[11] https://secure.php.net/manual/ja/class.traversable.php

```
}
```

yii\base\Model::attributes() をオーバーライドして、属性を異なる方法で定義することが出来ます。 このメソッドはモデルが持つ属性の名前を返さなくてはなりません。 例えば、yii\db\ActiveRecord は、関連付けられたデータベース・テーブルのコラム名を属性の名前として返すことによって、属性を定義しています。 ただし、これと同時に、定義された属性に対して通常のオブジェクト・プロパティと同じようにアクセスすることが出来るように、 __get() や __set() などのマジック・メソッドをオーバーライドする必要があるかもしれないことに注意してください。

属性のラベル

属性の値を表示したり、入力してもらったりするときに、属性と関連付けられたラベルを表示する必要があることがよくあります。 例えば、firstName という名前の属性を考えたとき、入力フォームやエラー・メッセージのような箇所でエンド・ユーザに表示するときは、 もっとユーザ・フレンドリーな First Name というラベルを表示したいと思うでしょう。

yii\base\Model::getAttributeLabel() を呼ぶことによって属性のラベルを得ることが出来ます。例えば、

```
$model = new \app\models\ContactForm;

// "Name" を表示する
echo $model->getAttributeLabel('name');
```

デフォルトでは、属性のラベルは属性の名前から自動的に生成されます。 ラベルの生成は yii\base\Model::generateAttributeLabel() というメソッドによって行われます。 このメソッドは、キャメルケースの変数名を複数の単語に分割し、各単語の最初の文字を大文字にします。 例えば、username は Username となり、firstName は First Name となります。

自動的に生成されるラベルを使用したくない場合は、yii\base\Model::attributeLabels() をオーバーライドして、 属性のラベルを明示的に宣言することが出来ます。例えば、

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;

    public function attributeLabels()
    {
```

```
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
            'body' => 'Content',
        ];
    }
}
```

複数の言語をサポートするアプリケーションでは、属性のラベルを翻訳したいと思うでしょう。 これも、以下のように、attributeLabels() の中で行うことが出来ます。

```
public function attributeLabels()
{
    return [
        'name' => \Yii::t('app', 'Your name'),
        'email' => \Yii::t('app', 'Your email address'),
        'subject' => \Yii::t('app', 'Subject'),
        'body' => \Yii::t('app', 'Content'),
    ];
}
```

条件に従って属性のラベルを定義することも出来ます。 例えば、モデルが使用される シナリオ に基づいて、同じ属性に対して違うラベルを返すことことが出来ます。

> 情報: 厳密に言えば、属性のラベルは ビュー の一部を成すものです。 しかし、たいていの場合、モデルの中でラベルを宣言する方が便利が良く、結果としてクリーンで再利用可能なコードになります。

3.6.2 シナリオ

モデルはさまざまに異なる シナリオ で使用されます。 例えば、User モデルはユーザ・ログインの入力を収集するために使われますが、同時に、ユーザ登録の目的でも使われます。 異なるシナリオの下では、モデルが使用するビジネス・ルールとロジックも異なるものになり得ます。 例えば、email 属性はユーザ登録の際には必須とされるかも知れませんが、ログインの際にはそうではないでしょう。

モデルは yii\base\Model::$scenario プロパティを使って、自身が使われているシナリオを追跡します。 デフォルトでは、モデルは default という一つのシナリオだけをサポートします。 次のコードは、モデルのシナリオを設定する二つの方法を示すものです。

```
// シナリオをプロパティとして設定する
$model = new User;
$model->scenario = User::SCENARIO_LOGIN;

// シナリオを設定情報で設定する
$model = new User(['scenario' => User::SCENARIO_LOGIN]);
```

デフォルトでは、モデルによってサポートされるシナリオは、モデルで宣言されている 検証規則 によって決定されます。 しかし、次のように、yii\base\Model::scenarios() メソッドをオーバーライドして、 この振る舞いをカスタマイズすることが出来ます。

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['username', 'password'],
            self::SCENARIO_REGISTER => ['username', 'email', 'password'],
        ];
    }
}
```

> 情報: 上記の例と後続の例では、モデル・クラスは yii\db\ActiveRecord を拡張するものとなっています。 というのは、複数のシナリオを使用することは、通常は、アクティブ・レコード クラスで発生するからです。

seanarios() メソッドは、キーがシナリオの名前であり、値が対応するアクティブな属性 である配列を返します。 アクティブな属性とは、一括代入 することが出来て、検証 の対象になる属性です。 上記の例では、login シナリオにおいては username と password の属性がアクティブであり、 一方、register シナリオにおいては、username と password に加えて email もアクティブです。

scenarios() のデフォルトの実装は、検証規則の宣言メソッドである yii\base\Model::rules() に現れる全てのシナリオを返すものです。 scenarios() をオーバーライドするときに、デフォルトのシナリオに加えて新しいシナリオを導入したい場合は、 次のようなコードを書きます。

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
```

```
        $scenarios = parent::scenarios();
        $scenarios[self::SCENARIO_LOGIN] = ['username', 'password'];
        $scenarios[self::SCENARIO_REGISTER] = ['username', 'email', '
    password'];
        return $scenarios;
    }
}
```

シナリオの機能は、主として、検証 と 属性の一括代入 によって使用されます。 しかし、他の目的に使うことも可能です。例えば、現在のシナリオに基づいて異なる 属性のラベル を宣言することも出来ます。

3.6.3 検証規則

モデルのデータをエンド・ユーザから受け取ったときは、データを検証して、 それが一定の規則 (検証規則、あるいは、いわゆる ビジネス・ルール) を満たしていることを確認しなければなりません。 ContactForm モデルを例に挙げるなら、全ての属性が空ではなく、email 属性が有効なメール・アドレスを含んでいることを確認したいでしょう。 いずれかの属性の値が対応するビジネス・ルールを満たしていないときは、 ユーザがエラーを訂正するのを助ける適切なエラー・メッセージが表示されなければなりません。

受信したデータを検証するために、yii\base\Model::validate() を呼ぶことが出来ます。 このメソッドは、yii\base\Model::rules() で宣言された検証規則を使って、該当するすべての属性を検証します。 エラーが見つからなければ、メソッドは true を返します。 そうでなければ、yii\base\Model::$errors にエラーを保存して、false を返します。例えば、

```
$model = new \app\models\ContactForm;

// モデルの属性にユーザの入力を代入する
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // すべての入力値は有効である
} else {
    // 検証が失敗: $errors はエラー・メッセージを含む配列
    $errors = $model->errors;
}
```

モデルに関連付けられた検証規則を宣言するためには、yii\base\Model::rules() メソッドをオーバーライドして、 モデルの属性が満たすべき規則を返すようにします。 次の例は、ContactForm モデルのために宣言された検証規則を示します。

```
public function rules()
{
    return [
        // 、、、nameemailsubjectbody の属性が必須
```

```
            [['name', 'email', 'subject', 'body'], 'required'],

            // email 属性は、有効なメール・アドレスでなければならない
            ['email', 'email'],
        ];
    }
```

一つの規則は、一つまたは複数の属性を検証するために使うことが出来ます。 また、一つの属性は、一つまたは複数の規則によって検証することが出来ます。 検証規則をどのように宣言するかについての詳細は 入力を検証する のセクションを参照してください。

時として、特定の シナリオ にのみ適用される規則が必要になるでしょう。 そのためには、下記のように、規則に on プロパティを指定することが出来ます。

```
public function rules()
{
    return [
        // "register" シナリオでは、、、usernameemailpassword のすべてが必須
        [['username', 'email', 'password'], 'required', 'on' => self::
    SCENARIO_REGISTER],

        // "login" シナリオでは、username と password が必須
        [['username', 'password'], 'required', 'on' => self::SCENARIO_LOGIN
    ],
    ];
}
```

on プロパティを指定しない場合は、その規則は全てのシナリオに適用されることになります。 現在の シナリオ に適用可能な規則は アクティブな規則 と呼ばれます。

属性が検証されるのは、それが scenarios() の中でアクティブな属性であると宣言されており、かつ、 その属性が rules() の中で宣言されている一つまたは複数のアクティブな規則と結び付けられている場合であり、また、そのような場合だけです。

3.6.4 一括代入

一括代入は、一行のコードを書くだけで、ユーザの入力した複数のデータをモデルに投入できる便利な方法です。 一括代入は、入力されたデータを yii\base\Model::$attributes プロパティに直接に代入することによって、モデルの属性にデータを投入します。 次の二つのコード断片は等価であり、どちらもエンド・ユーザから送信されたフォームのデータを ContactForm モデルの属性に割り当てようとするものです。 明らかに、一括代入を使う前者の方が、後者よりも明快で間違いも起こりにくいでしょう。

```
$model = new \app\models\ContactForm;
$model->attributes = \Yii::$app->request->post('ContactForm');
```

```
$model = new \app\models\ContactForm;
$data = \Yii::$app->request->post('ContactForm', []);
$model->name = isset($data['name']) ? $data['name'] : null;
$model->email = isset($data['email']) ? $data['email'] : null;
$model->subject = isset($data['subject']) ? $data['subject'] : null;
$model->body = isset($data['body']) ? $data['body'] : null;
```

安全な属性

一括代入は、いわゆる 安全な属性、すなわち、yii\base\Model::scenarios() においてモデルの現在の シナリオ のためにリストされている属性に対してのみ適用されます。 例えば、User モデルが次のようなシナリオ宣言を持っている場合において、現在のシナリオが login であるときは、username と password のみが一括代入が可能です。 その他の属性はいっさい触れられません。

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password'],
        self::SCENARIO_REGISTER => ['username', 'email', 'password'],
    ];
}
```

> 情報: 一括代入が安全な属性に対してのみ適用されるのは、エンド・ユーザの入力データがどの属性を修正することが出来るか、 ということを制御する必要があるからです。 例えば、User モデルに、ユーザに割り当てられた権限を決定する permission という属性がある場合、 この属性を修正できるのは、管理者がバックエンドのインタフェイスを通じてする時だけに制限したいでしょう。

yii\base\Model::scenarios() のデフォルトの実装は yii\base\Model::rules() に現われる全てのシナリオと属性を返すものです。 従って、このメソッドをオーバーライドしない場合は、アクティブな検証規則のどれかに出現する限り、 その属性は安全である、ということになります。

このため、実際に検証することなく属性を安全であると宣言できるように、 safe というエイリアスを与えられた特別なバリデータが提供されています。 例えば、次の規則は title と description の両方が安全な属性であると宣言しています。

```
public function rules()
{
    return [
        [['title', 'description'], 'safe'],
    ];
}
```

安全でない属性

上記で説明したように、yii\base\Model::scenarios() メソッドは二つの目的を持っています。 すなわち、どの属性が検証されるべきかを決めることと、どの属性が安全であるかを決めることです。 めったにない場合として、属性を検証する必要はあるが、安全であるという印は付けたくない、ということがあります。 そういう時は、下の例の secret 属性のように、名前の前に感嘆符 ! を付けて scenarios() の中で宣言することが出来ます。

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password', '!secret'],
    ];
}
```

このモデルが login シナリオにある場合、三つの属性は全て検証されます。 しかし、username と password の属性だけが一括代入が可能です。secret 属性に入力値を割り当てるためには、下記のように明示的に代入を実行する必要があります。

```
$model->secret = $secret;
```

同じ事が rules() メソッドの中でも出来ます。

```
public function rules()
{
    return [
        [['username', 'password', '!secret'], 'required', 'on' => 'login']
    ];
}
```

この場合、username、password そして secret の属性が必須項目とされますが、secret は明示的に代入される必要があります。

3.6.5 データのエクスポート

モデルを他の形式にエクスポートする必要が生じることはよくあります。例えば、モデルのコレクションを JSON や Excel 形式に変換したい場合があるでしょう。 このエクスポートのプロセスは二つの独立したステップに分割することが出来ます。

- モデルが配列に変換され、
- 配列が目的の形式に変換される。

あなたは最初のステップだけに注力することが出来ます。 と言うのは、第二のステップは汎用的なデータ・フォーマッタ、例えば yii\web\JsonResponseFormatter によって達成できるからです。

モデルを配列に変換する最も簡単な方法は、yii\base\Model::$attributes プロパティを使うことです。 例えば、

```
$post = \app\models\Post::findOne(100);
$array = $post->attributes;
```

デフォルトでは、yii\base\Model::$attributes プロパティは yii\base\Model::attributes() で宣言されている 全て の属性の値を返します。

モデルを配列に変換するためのもっと柔軟で強力な方法は、yii\base\Model::toArray() メソッドを使うことです。 このメソッドのデフォルトの動作は yii\base\Model::$attributes のそれと同じものです。 しかしながら、このメソッドを使うと、どのデータ項目 (フィールド と呼ばれます) を結果の配列に入れるか、そして、その項目にどのような書式を適用するかを選ぶことが出来ます。 実際、レスポンス形式の設定 で説明されているように、RESTful ウェブサービスの開発においては、 これがモデルをエクスポートするデフォルトの方法となっています。

フィールド

フィールドとは、単に、モデルの yii\base\Model::toArray() メソッドを呼ぶことによって取得される配列に含まれる、 名前付きの要素のことです。

デフォルトでは、フィールドの名前は属性の名前と等しいものになります。 しかし、このデフォルトの動作は、fields() および/または extraFields() メソッドをオーバーライドして、変更することが出来ます。 どちらのメソッドも、フィールド定義のリストを返すものです。 fields() によって定義されるフィールドは、デフォルト・フィールドです。すなわち、toArray() はデフォルトでこれらのフィールドを返す、ということを意味します。 extraFields() メソッドは、$expand パラメータによって指定する限りにおいて toArray() によって返される、追加のフィールドを定義するものです。 例として、次のコードは、fields() で定義された全てのフィールドと、(extraFields() で定義されていれば) prettyName と fullAddress フィールドを返すものです。

```
$array = $model->toArray([], ['prettyName', 'fullAddress']);
```

fields() をオーバーライドして、フィールドを追加、削除、リネーム、再定義することが出来ます。 fields() の返り値は配列でなければなりません。配列のキーはフィールド名であり、配列の値は対応するフィールド定義です。 フィールドの定義には、プロパティ/属性 の名前か、または、対応するフィールドの値を返す無名関数を使うことが出来ます。フィールド名がそれを定義する属性名と同一であるという特殊な場合においては、配列のキーを省略することが出来ます。 例えば、

```
// 明示的に全てのフィールドをリストする方法。(API の後方互換性を保つため
    に) DB テーブルや
// モデル属性の変更がフィールドの変更を引き起こさないことを保証したい場合に適し
    ている。
public function fields()
{
    return [
        // フィールド名が属性名と同じ
        'id',
```

```
        // フィールド名は "email"、対応する属性名は" "email_address"
        'email' => 'email_address',

        // フィールド名は "name"、その値は" PHP コールバックで定義
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// いくつかのフィールドを除外する方法。親の実装を継承しつつ、公開すべきでない
    フィールドは
// 除外したいときに適している。
public function fields()
{
    $fields = parent::fields();

    // 公開すべきでない情報を含むフィールドを削除する
    unset($fields['auth_key'], $fields['password_hash'], $fields['
    password_reset_token']);

    return $fields;
}
```

> 警告: デフォルトではモデルの全ての属性がエクスポートされる配列に含まれるため、データを精査して、 公開すべきでない情報が含まれていないことを確認しなければなりません。 そういう情報がある場合は、`fields()` をオーバーライドして、除外しなければなりません。 上記の例では、`auth_key`、`password_hash` および `password_reset_token` を除外しています。

3.6.6 ベスト・プラクティス

モデルは、ビジネスのデータ、規則、ロジックを表わす中心的なオブジェクトです。 モデルは、たいてい、さまざまな場所で再利用される必要があります。 良く設計されたアプリケーションでは、通常、モデルはコントローラ よりもはるかに太ったものになります。

要約すると、モデルは、

- ビジネス・データを表現する属性を含むことが出来ます。
- データの有効性と整合性を保証する検証規則を含むことが出来ます。
- ビジネス・ロジックを実装するメソッドを含むことが出来ます。
- リクエスト、セッション、または他の環境データに直接アクセスするべきではありません。 これらのデータは、コントローラ によってモデルに注入されるべきです。
- HTML を埋め込むなどの表示用のコードは避けるべきです - 表示はビュー で行う方が良いです。

- あまりに多くの シナリオ を一つのモデルで持つことは避けましょう。

大規模で複雑なシステムを開発するときには、たいてい、上記の最後にあげた推奨事項を考慮するのが良いでしょう。 そういうシステムでは、モデルは数多くの場所で使用され、それに従って、数多くの規則セットやビジネス・ロジックを含むため、 非常に太ったものになり得ます。 コードの一ヶ所に触れるだけで数ヶ所の違った場所に影響が及ぶため、ついには、モデルのコードの保守が悪夢になってしまうこともよくあります。 モデルのコードの保守性を高めるためには、以下の戦略をとることが出来ます。

- 異なる アプリケーション または モジュール によって共有される一連の基底モデル・クラスを定義します。 これらのモデル・クラスは、すべてで共通に使用される最小限の規則セットとロジックのみを含むべきです。
- モデルを使用するそれぞれの アプリケーション または モジュール において、 対応する基底モデル・クラスから拡張した具体的なモデル・クラスを定義します。 この具体的なモデル・クラスが、そのアプリケーションやモジュールに固有の規則やロジックを含むべきです。

例えば、アドバンスト・プロジェクト・テンプレート[12] の中で、基底モデル・クラス `common\models\Post` を定義することが出来ます。 次に、フロントエンド・アプリケーションにおいては、`common\models\Post` から拡張した具体的なモデル・クラス `frontend\models\Post` を定義して使います。 また、バックエンド・アプリケーションにおいても、同様に、`backend\models\Post` を定義します。 この戦略を取ると、`frontend\models\Post` の中のコードはフロントエンド・アプリケーション固有のものであると保証することが出来ます。 そして、フロントエンドのコードにどのような変更を加えても、バックエンド・アプリケーションを壊すかもしれないと心配する必要がなくなります。

3.7 ビュー

ビューは MVC[13] アーキテクチャの一部を成すものです。 ビューはエンド・ユーザにデータを表示することに責任を持つコードです。 ウェブ・アプリケーションにおいては、ビューは、通常、ビュー・テンプレート の形式、すなわち、 主として HTML コードと表示目的の PHP コードを含む PHP スクリプト・ファイルとして作成されます。 そして、ビュー・テンプレートを管理する ビュー アプリケーション・コンポーネント が、ビューの構築とレンダリングを助けるためによく使われるメソッドを提供します。 なお、簡潔さを重視して、ビュー・テンプレート

[12] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/README.md

[13] http://ja.wikipedia.org/wiki/Model_View_Controller

またはビュー・テンプレート・ファイルを単にビューと呼ぶことがよくあります。

3.7.1 ビューを作成する

前述のように、ビューは HTML と PHP コードが混ざった単なる PHP スクリプトです。 次に示すのは、ログイン・フォームを表示するビューです。 ご覧のように、PHP コードがタイトルやフォームなど動的なコンテントを生成するのに使われ、HTML コードがそれらを編成して表示可能な HTML ページを作っています。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model app\models\LoginForm */

$this->title = 'ログイン';
?>
<h1><?= Html::encode($this->title) ?></h1>

<p次の項目を入力してログインしてください>:</p>

<?php $form = ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('ログイン') ?>
<?php ActiveForm::end(); ?>
```

ビューの中では、このビュー・テンプレートを管理しレンダリングしている ビュー・コンポーネント を参照する `$this` にアクセスすることが出来ます。

`$this` 以外に、上記の例の `$model` のように、事前に定義される変数をビューの中に置くことが出来ます。 このような変数は、ビューのレンダリング をトリガする コントローラ などのオブジェクトによってビューに プッシュ されるデータを表します。

> ヒント: 上の例では、事前に定義される変数は、IDE に認識されるように、ビューの先頭のコメント・ブロックの中にリストされています。 これは、ビューにドキュメントを付けるのにも良い方法です。

セキュリティ

HTML ページを生成するビューを作成するときは、エンド・ユーザから受け取るデータを表示する前にエンコード および/または フィルタする

ことが重要です。 そうしなければ、あなたのアプリケーションは クロス・サイト・スクリプティング[14] 攻撃を こうむるおそれがあります。

プレーン・テキストを表示するためには、まず yii\helpers\Html::encode() を呼んでエンコードします。 例えば、次のコードはユーザの名前を表示する前にエンコードしています。

```
<?php
use yii\helpers\Html;
?>

<div class="username">
    <?= Html::encode($user->name) ?>
</div>
```

HTML コンテントを表示するためには、yii\helpers\HtmlPurifier を使って、最初にコンテントをフィルタします。 例えば、次のコードは、投稿のコンテントを表示する前にフィルタしています。

```
<?php
use yii\helpers\HtmlPurifier;
?>

<div class="post">
    <?= HtmlPurifier::process($post->text) ?>
</div>
```

> ヒント: HTMLPurifier は、出力を安全なものにすることにおいては素晴らしい仕事をしますが、速くはありません。 アプリケーションが高いパフォーマンスを要求する場合は、フィルター結果を キャッシュ することを考慮すべきです。

ビューを編成する

コントローラ や モデル と同じように、ビューを編成するための規約があります。.

- コントローラによって表示されるビューは、デフォルトでは、ディレクトリ @app/views/ControllerID の下に置かれるべきものです。 ここで、ControllerID は コントローラ ID を指します。 例えば、コントローラ・クラスが PostController である場合、ディレクトリは @app/views/post となります。 PostCommentController の場合は、ディレクトリは @app/views/post-comment です。 また、コントローラがモジュールに属する場合は、ディレクトリは モジュール・ディレクトリ の下の views/ControllerID です。

[14] http://ja.wikipedia.org/wiki/%E3%82%AF%E3%83%AD%E3%82%B9%E3%82%B5%E3%82%A4%E3%83%88%E3%82%B9%E3%82%AF%E3%83%AA%E3%83%97%E3%83%86%E3%82%A3%E3%83%B3%E3%82%B0

- ウィジェット で表示されるビューは、デフォルトでは、WidgetPath/views ディレクトリの下に置かれるべきものです。 ここで、WidgetPath は、ウィジェットのクラス・ファイルを含んでいるディレクトリを指します。
- 他のオブジェクトによって表示されるビューについても、ウィジェットの場合と同じ規約に従うことが推奨されます。

これらのデフォルトのビュー・ディレクトリは、コントローラやウィジェットの yii\base\ViewContextInterface::getViewPath() メソッドをオーバーライドすることでカスタマイズすることが可能です。

3.7.2 ビューをレンダリングする

コントローラ の中でも、ウィジェット の中でも、または、その他のどんな場所でも、ビューをレンダリングするメソッドを呼ぶことによってビューをレンダリングすることが出来ます。 これらのメソッドは、下記に示されるような類似のシグニチャを共有します。

```
/**
 * @param string $view ビュー名またはファイル・パス 実際のレンダリング・メソッドに依存する()
 * @param array $params ビューに引き渡されるデータ
 * @return string レンダリングの結果
 */
methodName($view, $params = [])
```

コントローラでのレンダリング

コントローラ の中では、ビューをレンダリングするために次のコントローラ・メソッドを呼ぶことが出来ます。

- render(): 名前付きビュー をレンダリングし、 その結果に レイアウト を適用する。
- renderPartial(): 名前付きビュー をレイアウトなしでレンダリングする。
- renderAjax(): 名前付きビュー をレイアウトなしでレンダリングし、登録されている全ての JS/CSS スクリプトおよびファイルを注入する。 通常、AJAX ウェブ・リクエストに対するレスポンスにおいて使用される。
- renderFile(): ビュー・ファイルのパスまたは エイリアス の形式で指定されたビューをレンダリングする。
- renderContent(): 静的な文字列をレンダリングして、現在適用可能な レイアウトに埋め込む。 このメソッドは バージョン 2.0.1 以降で使用可能。

例えば、

```
namespace app\controllers;

use Yii;
```

```
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        // "view" という名前のビューをレンダリングし、レイアウトを適用する
        return $this->render('view', [
            'model' => $model,
        ]);
    }
}
```

ウィジェットでのレンダリング

ウィジェット の中では、ビューをレンダリングするために、次のウィジェット・メソッドを使用することが出来ます。

- render(): 名前付きビュー をレンダリングする。
- renderFile(): ビュー・ファイルのパスまたは エイリアス の形式で指定されたビューをレンダリングする。

例えば、

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class ListWidget extends Widget
{
    public $items = [];

    public function run()
    {
        // "list" という名前のビューをレンダリングする
        return $this->render('list', [
            'items' => $this->items,
        ]);
    }
}
```

ビューでのレンダリング

ビュー・コンポーネント によって提供される下記のメソッドのどれかを使うと、ビューの中で、別のビューをレンダリングすることが出来ま

す。

- render(): 名前付きビュー をレンダリングする。
- renderAjax(): 名前付きビュー をレンダリングし、登録されている全ての JS/CSS スクリプトおよびファイルを注入する。 通常、AJAX ウェブリクエストに対するレスポンスにおいて使用される。
- renderFile(): ビュー・ファイルのパスまたは エイリアス の形式で指定されたビューをレンダリングする。

例えば、ビューの中の次のコードは、現在レンダリングされているビューと同じディレクトリにある _overview.php というビュー・ファイルをレンダリングします。 ビューでは $this が ビュー・コンポーネントを参照することを思い出してください。

```
<?= $this->render('_overview') ?>
```

他の場所でのレンダリング

場所がどこであれ、Yii::$app->view という式によって ビュー・アプリケーション・コンポーネントにアクセスすることが出来ますから、 前述の ビュー・コンポーネント・メソッドを使ってビューをレンダリングすることが出来ます。例えば、

```
// ビュー・ファイル "@app/views/site/license.php" を表示
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

名前付きビュー

ビューをレンダリングするとき、ビューを指定するのには、ビューの名前か、ビュー・ファイルのパス/エイリアスか、どちらかを使うことが出来ます。 たいていの場合は、より簡潔で柔軟な前者を使います。名前を使って指定されるビューを 名前付きビュー と呼びます。

ビューの名前は、以下の規則に従って、対応するビュー・ファイルのパスに解決されます。

- ビュー名はファイル拡張子を省略することが出来ます。その場合、.php が拡張子として使われます。 例えば、about というビュー名は about.php というファイル名に対応します。
- ビュー名が二つのスラッシュ (//) で始まる場合は、対応するビュー・ファイルのパスは @app/views/ViewName となります。 つまり、ビュー・ファイルは アプリケーションのビュー・パス の下で探されます。 例えば、//site/about は @app/views/site/about.php へと解決されます。
- ビュー名が一つのスラッシュ (/) で始まる場合は、ビュー・ファイルのパスは、ビュー名の前に、現在アクティブな モジュール の ビュー・パス を置くことによって形成されます。 アクティブなモジュールが無い場合は、@app/views/ViewName が使用されます。 例え

ば、/user/create は、現在アクティブなモジュールが user である場合は、@app/modules/user/views/user/create.php へと解決されます。 アクティブなモジュールが無い場合は、ビュー・ファイルのパスは @app/views/user/create.php となります。

- ビューが コンテキスト を伴ってレンダリングされ、そのコンテキストが yii\base\ViewContextInterface を実装している場合は、 ビュー・ファイルのパスは、コンテキストの ビュー・パス をビュー名の前に置くことによって形成されます。 これは、主として、コントローラとウィジェットの中でレンダリングされるビューに当てはまります。 例えば、コンテキストが SiteController コントローラである場合、about は @app/views/site/about.php へと解決されます。
- あるビューが別のビューの中でレンダリングされる場合は、後者のビュー・ファイルを含んでいるディレクトリが前者のビュー名の前に置かれて、 実際のビュー・ファイルのパスが形成されます。 例えば、item は、@app/views/post/index.php というビューの中でレンダリングされる場合、@app/views/post/item へと解決されます。

上記の規則によって、コントローラ app\controllers\PostController の中で $this->render('view') を呼ぶと、 実際には、ビュー・ファイル @app/views/post/view.php がレンダリングされ、一方、そのビューの中で $this->render('_overview') を呼ぶと、 ビュー・ファイル @app/views/post/_overview.php がレンダリングされることになります。

ビューの中でデータにアクセスする

ビューの中でデータにアクセスするためのアプローチが二つあります。「プッシュ」と「プル」です。

ビューをレンダリングするメソッドに二番目のパラメータとしてデータを渡すのが「プッシュ」のアプローチです。 データは、「名前-値」のペアの配列として表わされなければなりません。 ビューがレンダリングされるときに、PHP の extract() 関数がこの配列に対して呼び出され、ビューの中で使う変数が抽出されます。 例えば、次のコードはコントローラの中でビューをレンダリングしていますが、report ビューに二つの変数、 すなわち、$foo = 1 と $bar = 2 をプッシュしています。

```
echo $this->render('report', [
    'foo' => 1,
    'bar' => 2,
]);
```

「プル」のアプローチは、ビュー・コンポーネント またはビューからアクセス出来るその他のオブジェクト (例えば Yii::$app) から積極的にデータを読み出すものです。 下記のコード例のように、ビューの中では $this->context という式でコントローラ・オブジェクトを取得することが出来ます。 その結果、report ビューの中で、コントローラの全てのプ

ロパティやメソッドにアクセスすることが出来ます。 次の例ではコントローラ ID にアクセスしています。

```
The controller ID is: <?= $this->context->id ?>
```

通常は「プッシュ」アプローチが、ビューでデータにアクセスする方法として推奨されます。なぜなら、ビューのコンテキスト・オブジェクトに対する依存がより少ないからです。 その短所は、常にデータ配列を手作業で作成する必要がある、ということです。 ビューが共有されてさまざまな場所でレンダリングされる場合、その作業が面倒くさくなり、また、間違いも生じやすくなります。

ビューの間でデータを共有する

ビュー・コンポーネント が提供する params プロパティを使うと、ビューの間でデータを共有することが出来ます。

例えば、about というビューで、次のようなコードを使って、 パン屑リストの現在の区分を指定することが出来ます。

```
$this->params['breadcrumbs'][] = 'About Us';
```

そして、レイアウト ファイル (これも一つのビューです) の中で、params によって渡されたデータを使って、 パン屑リストを表示することが出来ます。

```
<?= yii\widgets\Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ? $this->params['
    breadcrumbs'] : [],
]) ?>
```

3.7.3 レイアウト

レイアウトは、複数のビューの共通部分をあらわす特殊なタイプのビューです。 例えば、たいていのウェブ・アプリケーションでは、ページは共通のヘッダとフッタを持っています。 すべてのビューで同じヘッダとフッタを繰り返すことも出来ますが、もっと良い方法は、 そういうことはレイアウトの中で一度だけして、コンテント・ビューのレンダリング結果をレイアウトの中の適切な場所に埋め込むことです。

レイアウトを作成する

レイアウトもまたビューですので、通常のビューと同様な方法で作成することが出来ます。 デフォルトでは、レイアウトは @app/views/layouts ディレクトリに保存されます。 モジュール の中で使用されるレイアウトについては、モジュール・ディレクトリ の下の views/layouts ディレクトリに保存されるべきものとなります。 デフォルトのレイアウト・ディレクトリは、アプリケーションまたはモジュールの yii\base\Module::$layoutPath プロパティを構成することでカスタマイズすることが出来ます。

次の例は、レイアウトがどのようなものであるかを示すものです。説明のために、レイアウトの中のコードを大幅に単純化していることに注意してください。 実際には、ヘッドのタグやメイン・メニューなど、もっと多くのコンテントを追加する必要があるでしょう。

```
<?php
use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $content string */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <header>My Company</header>
    <?= $content ?>
    <footer>&copy; 2014 by My Company</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

ご覧のように、レイアウトはすべてのページに共通な HTML タグを生成しています。 `<body>` セクションの中でレイアウトが `$content` という変数をエコーしていますが、これは、コンテント・ビューのレンダリング結果を表すものであり、 `yii\base\Controller::render()` が呼ばれるときに、レイアウトにプッシュされるものです。

上記のコードに示されているように、たいていのレイアウトは次に挙げるメソッドを呼び出さなければなりません。 これらのメソッドは、主としてレンダリングの過程に関するイベントをトリガするもので、 他の場所で登録されたスクリプトやタグが、メソッドが呼ばれた場所に正しく注入されるようにするためのものです。

- `beginPage()`: このメソッドがレイアウトの冒頭で呼ばれなければなりません。 これは、ページの開始を示す `EVENT_BEGIN_PAGE` イベントをトリガします。
- `endPage()`: このメソッドがレイアウトの末尾で呼ばれなければなりません。 これは、ページの終了を示す `EVENT_END_PAGE` イベントをトリガします。
- `head()`: このメソッドが HTML ページの `<head>` セクションの中で呼ばれなければなりません。 このメソッドは、ページのレンダリングが完了したときに、登録された head の HTML コード (リンク・

タグ、メタ・タグなど) に置き換えられるプレースホルダを生成します。
- beginBody(): このメソッドが <body> セクションの冒頭で呼ばれなければなりません。 このメソッドは EVENT_BEGIN_BODY イベントをトリガし、 body の開始位置をターゲットとする登録された HTML コード (JavaScript など) によって置き換えられるプレースホルダを生成します。
- endBody(): このメソッドが <body> セクションの末尾で呼ばれるなければなりません。 このメソッドは EVENT_END_BODY イベントをトリガし、 body の終了位置をターゲットとする登録された HTML コード (JavaScript など) によって置き換えられるプレースホルダを生成します。

レイアウトでデータにアクセスする

レイアウトの中では、事前定義された二つの変数、すなわち、$this と $content にアクセスすることが出来ます。 前者は、通常のビューにおいてと同じく、ビュー コンポーネントを参照します。 一方、後者は、コントローラの中で render() メソッドを呼ぶことによってレンダリングされる、コンテント・ビューのレンダリング結果を含むものです。

レイアウトの中でその他のデータにアクセスする必要があるときは、ビューの中でデータにアクセスする の項で説明されている「プル」の方法を使う必要があります。 コンテント・ビューからレイアウトにデータを渡す必要があるときは、ビューの間でデータを共有する の項で説明されている方法を使うことが出来ます。

レイアウトを使う

コントローラでのレンダリング の項で説明されているように、 コントローラの中で render() メソッドを呼んでビューをレンダリングすると、レンダリング結果にレイアウトが適用されます。 デフォルトでは、@app/views/layouts/main.php というレイアウトが使用されます。

yii\base\Application::$layout または yii\base\Controller::$layout のどちらかを構成することによって、異なるレイアウトを使うことが出来ます。 前者は全てのコントローラによって使用されるレイアウトを決定するものですが、後者は個々のコントローラについて前者をオーバーライドするものです。 例えば、次のコードは、post コントローラがビューをレンダリングするときに @app/views/layouts/post.php をレイアウトとして使うようにするものです。 その他のコントローラは、layout プロパティに触れられていないと仮定すると、 引き続きデフォルトの @app/views/layouts/main.php をレイアウトとして使います。

```
namespace app\controllers;

use yii\web\Controller;
```

```
class PostController extends Controller
{
    public $layout = 'post';

    // ...
}
```

モジュールに属するコントローラについては、モジュールの layout プロパティを構成して、 モジュール内のコントローラに特定のレイアウトを使用することも出来ます。

layout プロパティは異なるレベル (コントローラ、モジュール、アプリケーション) で構成されうるものですので、 Yii は舞台裏で二つのステップを踏んで、特定のコントローラで実際に使われるレイアウト・ファイルが何であるかを決定します。

最初のステップで、Yii はレイアウトの値とコンテキスト・モジュールを決定します。

- コントローラの yii\base\Controller::$layout プロパティが null でないときは、それをレイアウトの値として使い、 コントローラのモジュール をコンテキスト・モジュールとして使う。
- layout が null のときは、コントローラの祖先となっている全てのモジュール (アプリケーション自体も含む) を探して、 layout プロパティが null でない最初のモジュールを見つける。 見つかったモジュールとその layout の値をコンテキスト・モジュールと選ばれたレイアウトの値とする。 そのようなモジュールが見つからなかったときは、レイアウトは適用されないということを意味する。

第二のステップでは、最初のステップで決定されたレイアウトの値とコンテキスト・モジュールに従って、実際のレイアウト・ファイルを決定します。 レイアウトの値は下記のいずれかであり得ます。

- パス・エイリアス (例えば、@app/views/layouts/main)。
- 絶対パス (例えば、/main): すなわち、スラッシュで始まるレイアウトの値の場合。 実際のレイアウトファイルはアプリケーションのレイアウト・パス (デフォルトでは @app/views/layouts) の下で探される。
- 相対パス (例えば、main): 実際のレイアウト・ファイルはコンテキスト・モジュールの レイアウト・パス (デフォルトでは モジュール・ディレクトリ の下の views/layouts ディレクトリ) の下で探される。
- 真偽値 false: レイアウトは適用されない。

レイアウトの値がファイル拡張子を含んでいない場合は、デフォルト値である .php を使います。

入れ子のレイアウト

ときとして、あるレイアウトの中に別のレイアウトを入れたい場合があるでしょう。 例えば、ウェブ・サイトの別々のセクションにおいて、違うレイアウトを使いたいけれども、 それらのレイアウトは全て、全体と

しての HTML5 ページ構造を生成する同一の基本レイアウトを共有している、という場合です。 この目的を達することは、次のように、子レイアウトの中で beginContent() と endContent() を呼ぶことで可能になります。

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>

... 子レイアウトのコンテントをここに ...

<?php $this->endContent(); ?>
```

上のコードが示すように、子レイアウトのコンテントは beginContent() と endContent() によって囲まれなければなりません。 beginContent() に渡されるパラメータは、親レイアウトが何であるかを指定するものです。 レイアウトのファイルまたはエイリアスのどちらかを使うことが出来ます。

上記のアプローチを使って、2レベル以上のレイアウトを入れ子にすることも出来ます。

ブロックを使う

ブロックを使うと、ある場所でビューコンテントを定義して、別の場所でそれを表示することが可能になります。ブロックはたいていはレイアウトと一緒に使われます。 例えば、ブロックをコンテント・ビューで定義して、それをレイアウトで表示する、ということが出来ます。

beginBlock() と endBlock() を呼んでブロックを定義します。 すると、そのブロックを $view->blocks[$blockID] によってアクセス出来るようになります。 ここで $blockID は、定義したときにブロックに割り当てたユニークな ID を指します。

次の例は、どのようにブロックを使えば、レイアウトの特定の部分をコンテント・ビューでカスタマイズすることが出来るかを示すものです。

最初に、コンテント・ビューで、一つまたは複数のブロックを定義します。

```
...

<?php $this->beginBlock('block1'); ?>

... block1 のコンテント ...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('block3'); ?>

... block3 のコンテント ...
```

```
<?php $this->endBlock(); ?>
```

次に、レイアウト・ビューで、得ることが出来ればブロックをレンダリングし、 ブロックが定義されていないときは何らかのデフォルトのコンテントを表示します。

```
...
<?php if (isset($this->blocks['block1'])): ?>
    <?= $this->blocks['block1'] ?>
<?php else: ?>
    ... block1 のデフォルトのコンテント ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>
    <?= $this->blocks['block2'] ?>
<?php else: ?>
    ... block2 のデフォルトのコンテント ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block3'])): ?>
    <?= $this->blocks['block3'] ?>
<?php else: ?>
    ... block3 のデフォルトのコンテント ...
<?php endif; ?>
...
```

3.7.4 ビュー・コンポーネントを使う

ビュー・コンポーネント はビューに関連する多くの機能を提供します。ビュー・コンポーネントは、`yii\base\View` またはその子クラスの個別のインスタンスを作成することによっても取得できますが、 たいていの場合は、`view` アプリケーション・コンポーネントを主として使うことになるでしょう。 このコンポーネントは アプリケーションの構成情報 の中で、次のようにして構成することが出来ます。

```
[
    // ...
    'components' => [
        'view' => [
            'class' => 'app\components\View',
        ],
        // ...
    ],
]
```

ビュー・コンポーネントは、次に挙げるビュー関連の有用な機能を提供します。それぞれについては、独立のセクションで更に詳細に説明されます。

- テーマ: ウェブ・サイトのテーマを開発し変更することを可能にします。
- フラグメント・キャッシュ: ウェブ・ページの中の断片をキャッシュすることを可能にします。
- クライアント・スクリプトの取り扱い: CSS と JavaScript の登録とレンダリングをサポートします。
- アセット・バンドルの取り扱い: アセット・バンドル の登録とレンダリングをサポートします。
- 代替のテンプレート・エンジン: Twig[15]、Smarty[16] など、 他のテンプレート・エンジンを使用することを可能にします。

次に挙げるマイナーではあっても有用な諸機能は、ウェブ・ページを開発するときに頻繁に使用するでしょう。

ページタイトルを設定する

どんなウェブ・ページにもタイトルが無ければなりません。通常、タイトル・タグは layout の中で表示されます。 しかし、実際においては、多くの場合、タイトルはレイアウトではなくコンテント・ビューで決められます。 この問題を解決するために、`yii\web\View` は、タイトル情報をコンテント・ビューからレイアウトに渡すための `title` プロパティを提供しています。

この機能を利用するためには、全てのコンテント・ビューにおいて、次のようにタイトルを設定します。

```
<?php
$this->title = 'My page title';
?>
```

そして、レイアウト・ビューで、`<head>` セクションに次のコードを忘れずに書くようにします。

```
<title><?= Html::encode($this->title) ?></title>
```

メタ・タグを登録する

ウェブ・ページは、通常、いろいろな関係者によって必要とされるさまざまなメタ・タグを生成する必要があります。 ページ・タイトルと同じように、メタ・タグは `<head>` セクションに出現して、通常はレイアウトの中で生成されます。

どのようなメタ・タグを生成するかをコンテント・ビューの中で指定したい場合は、下記のように、 `yii\web\View::registerMetaTag()` をコンテント・ビューで呼ぶことが出来ます。

```
<?php
```

[15] http://twig.sensiolabs.org/
[16] http://www.smarty.net/

```
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework,
    php']);
?>
```

上記のコードは、ビュー・コンポーネントによって “keywords” メタ・タグを登録するものです。 登録されたメタ・タグは、レイアウトがレンダリングを完了した後でレンダリングされます。 すなわち、レイアウトの中で `yii\web\View::head()` を呼び出した場所に、次の HTML コードが生成されて挿入されます。

```
<meta name="keywords" content="yii, framework, php">
```

`yii\web\View::registerMetaTag()` を複数回呼び出した場合は、メタ・タグが同じものか否かに関係なく、 複数のメタ・タグが登録されることに注意してください。

ある型のメタ・タグのインスタンスが一つだけになることを保証したい場合は、このメソッドを呼ぶときに第二のパラメータとしてキーを指定することが出来ます。 例えば、次のコードでは、二つの “description” メタ・タグを登録していますが、二番目のものだけがレンダリングされることになります。

```
$this->registerMetaTag(['name' => 'description', 'content' => '俺が Yii で
    作ったクールなウェブ・サイトだぜい!!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => '面白いアライ
    グマに関するウェブ・サイトです。'], 'description');
```

リンク・タグを登録する

メタ・タグ と同じように、リンク・タグも多くの場合において有用なものです。例えば、favicon をカスタマイズしたり、RSS フィードを指し示したり、OpenID を別のサーバに委任したり、等々。 リンク・タグも、`yii\web\View::registerLinkTag()` を使って、メタ・タグと同じような方法で取り扱うことが出来ます。 例えば、コンテント・ビューにおいて、次のようにしてリンク・タグを登録することが出来ます。

```
$this->registerLinkTag([
    'title' => 'Yii ライブ・ニューズ',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'http://www.yiiframework.com/rss.xml/',
]);
```

上記のコードは、次の結果になります。

```
<link title="Yii ライブ・ニューズ" rel="alternate" type="application/rss+xml
    " href="http://www.yiiframework.com/rss.xml/">
```

`registerMetaTag()` と同じように、`registerLinkTag()` を呼ぶときにキーを指定すると、 同じリンク・タグを繰り返して生成するのを避けることが出来ます。

3.7.5 ビューのイベント

ビュー・コンポーネント はビューをレンダリングする過程においていくつかのイベントをトリガします。 これらのイベントに反応することによって、ビューにコンテントを注入したり、エンド・ユーザに送信される前にレンダリング結果を加工したりすることが出来ます。

- EVENT_BEFORE_RENDER: コントローラでファイルをレンダリングする前にトリガされます。 このイベントのハンドラは、yii\base\ViewEvent::$isValid を false にセットして、レンダリングのプロセスをキャンセルすることが出来ます。
- EVENT_AFTER_RENDER: ファイルのレンダリングの後、yii\base\View::afterRender() を呼ぶことによってトリガされます。 このイベントのハンドラは、レンダリング結果をプロパティ yii\base\ViewEvent::$output を通じて取得して、 それを修正してレンダリング結果を変更することが出来ます。
- EVENT_BEGIN_PAGE: レイアウトの中で yii\base\View::beginPage() を呼ぶことによってトリガされます。
- EVENT_END_PAGE: レイアウトの中で yii\base\View::endPage() を呼ぶことによってトリガされます。
- EVENT_BEGIN_BODY: レイアウトの中で yii\web\View::beginBody() を呼ぶことによってトリガされます。
- EVENT_END_BODY: レイアウトの中で yii\web\View::endBody() を呼ぶことによってトリガされます。

例えば、次のコードはページの body の最後に現在の日付を注入するものです。

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {
    echo date('Y-m-d');
});
```

3.7.6 静的なページをレンダリングする

静的なページというのは、主たるコンテントのほとんどが静的なもので、 コントローラからプッシュされる動的なデータにアクセスする必要がないページを指します。

静的なページは、そのコードをビューに置き、そして、コントローラで次のようなコードを使うと表示することが出来ます。

```
public function actionAbout()
{
    return $this->render('about');
}
```

ウェブ・サイトが多くの静的なページを含んでいる場合、同じようなコードを何度も繰り返すのは非常に面倒くさいでしょう。 この問題を解決するために、yii\web\ViewAction というスタンドアロン・アクションをコントローラに導入することが出来ます。 例えば、

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

このようにすると、ディレクトリ @app/views/site/pages の下に about という名前のビューを作成したときに、 次の URL によってこのビューを表示することが出来るようになります。

```
http://localhost/index.php?r=site%2Fpage&view=about
```

view という GET パラメータが、どのビューがリクエストされているかを yii\web\ViewAction に教えます。 そこで、アクションはこのビューをディレクトリ @app/views/site/pages の下で探します。 yii\web\ViewAction::$viewPrefix を構成して、ビューを探すディレクトリを変更することが出来ます。

3.7.7 ベスト・プラクティス

ビューはエンド・ユーザが望む形式でモデルを表現することに対して責任を持ちます。一般的に、ビューは

- 主として表示目的のコードを含むべきです。例えば、HTML、または、データをたどって書式化してレンダリングする簡単な PHP コードなど。
- DB クエリを実行するコードは含むべきではありません。そのようなコードはモデルの中で実行されるべきです。
- $_GET や $_POST のようなリクエスト・データに直接アクセスするべきではありません。それはコントローラの仕事です。 リクエスト・データが必要な場合は、コントローラからビューにプッシュされるべきです。
- モデルのプロパティを読み出すことが出来ます。しかし、それを修正するべきではありません。

ビューを管理しやすいものにするために、複雑すぎるビューや、冗長なコードをあまりに多く含むビューを作ることは避けましょう。 この目的を達するために、次のテクニックを使うことが出来ます。

- 共通の表示セクション (ページのヘッダやフッタなど) を表すためにレイアウト を使う。

- 複雑なビューはいくつかの小さなビューに分割する。既に説明したレンダリングのメソッドを使えば、 小さなビューをレンダリングして大きなビューを組み上げることが出来る。
- ビューの構成要素として ウィジェット を使う。
- ビューでデータを変換し書式化するためのヘルパ・クラスを作成して使う。

3.8 モジュール

モジュールは、モデル、ビュー、コントローラ、およびその他の支援コンポーネントから構成される自己充足的なソフトウェアのユニットです。 モジュールが アプリケーション にインストールされている場合、エンド・ユーザはモジュールのコントローラにアクセスする事が出来ます。 これらのことを理由として、モジュールは小さなアプリケーションと見なされることがよくあります。 しかし、モジュールは単独では配備できず、アプリケーションの中に存在しなければならないという点で アプリケーション とは異なります。

3.8.1 モジュールを作成する

モジュールは、モジュールの ベース・パス と呼ばれるディレクトリとして編成されます。 このディレクトリの中に、ちょうどアプリケーションの場合と同じように、controllers、models、views のようなサブ・ディレクトリが存在して、コントローラ、モデル、ビュー、その他のコードを収納しています。 次の例は、モジュール内の中身を示すものです。

```
forum/
    Module.php                  モジュール・クラス・ファイル
    controllers/                コントローラ・クラス・ファイルを含む
        DefaultController.php   デフォルトのコントローラ・クラス・ファイル
    models/                     モデル・クラス・ファイルを含む
    views/                      コントローラのビューとレイアウトのファイルを
    含む
        layouts/                レイアウトのビュー・ファイルを含む
        default/                DefaultController のためのビュー・ファイルを
    含む
            index.php           index ビュー・ファイル
```

モジュール・クラス

全てのモジュールは yii\base\Module から拡張したユニークなモジュール・クラスを持たなければなりません。 モジュール・クラスは、モジュールの ベース・パス 直下に配置されて オートロード可能 になっていなければなりません。 モジュールがアクセスされたとき、対応するモジュール・クラスの単一のインスタンスが作成されます。 アプリケーションのインスタンス と同じように、モジュールのインスタンスは、 モ

ジュール内のコードがデータとコンポーネントを共有するために使用されます。

次のコードは、モジュール・クラスがどのようなものかを示す例です。

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';
        // ... 他の初期化コード ...
    }
}
```

`init` メソッドがモジュールのプロパティを初期化するためのコードをたくさん含む場合は、 それを 構成情報 の形で保存し、`init()` の中で次のコードを使って読み出すことも可能です。

```
public function init()
{
    parent::init();
    // config.php からロードした構成情報でモジュールを初期化する
    \Yii::configure($this, require __DIR__ . '/config.php');
}
```

ここで、構成情報ファイル `config.php` は、アプリケーションの構成情報の場合と同じように、 次のような内容を含むことが出来ます。

```
<?php
return [
    'components' => [
        // コンポーネントの構成情報のリスト
    ],
    'params' => [
        // パラメータのリスト
    ],
];
```

モジュール内のコントローラ

モジュールの中でコントローラを作成するときは、コントローラ・クラスをモジュール・クラスの名前空間の `controllers` サブ名前空間に置くことが規約です。 このことは、同時に、コントローラのクラス・ファイルをモジュールの ベース・パス 内の `controllers` ディレクトリに置くべきことをも意味します。 例えば、前の項で示された `forum` モジュールの中で `post` コントローラを作成するためには、 次のようにしてコントローラを宣言しなければなりません。

```
namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}
```

コントローラ・クラスの名前空間は、yii\base\Module::$controllerNamespace プロパティを構成してカスタマイズすることが出来ます。 いくつかのコントローラがこの名前空間の外にある場合でも、yii\base\Module::$controllerMap プロパティを構成することによって、それらをアクセス可能にすることが出来ます。 これは、アプリケーションでのコントローラ・マップ の場合と同様です。

モジュール内のビュー

モジュール内のビューは、モジュールの ベース・パス 内の views ディレクトリに置かれなくてはなりません。 モジュール内のコントローラによってレンダリングされるビューは、ディレクトリ views/ControllerID の下に置きます。 ここで、ControllerID は コントローラ ID を指します。例えば、コントローラ・クラスが PostController である場合、ディレクトリはモジュールの ベース・パス の中の views/post となります。

モジュールは、そのモジュールのコントローラによってレンダリングされるビューに適用される レイアウト を指定することが出来ます。 レイアウトは、デフォルトでは views/layouts ディレクトリに置かれなければならず、また、 yii\base\Module::$layout プロパティがレイアウトの名前を指すように構成しなければなりません。 layout プロパティを構成しない場合は、アプリケーションのレイアウトが代りに使用されます。

モジュール内のコンソールコマンド

コンソール モードで使用する事が出来るコマンドをモジュール内で宣言することも可能です。

あなたのコマンドがコマンド・ライン・ユーティリティから見えるようにするためには、Yii がコンソール・モードで実行されたときに yii\base\Module::$controllerNamespace を変更して、コマンドの名前空間を指し示すようにする必要があります。

それを達成する一つの方法は、モジュールの init() メソッドの中で Yii アプリケーションのインスタンスの型を調べるという方法です。

```
public function init()
{
    parent::init();
    if (Yii::$app instanceof \yii\console\Application) {
        $this->controllerNamespace = 'app\modules\forum\commands';
    }
```

```
}
```

このようにすれば、コマンドラインから次のルートを使ってあなたのコマンドを使用する事が出来るようになります。

```
yii <module_id>/<command>/<sub_command>
```

3.8.2 モジュールを使う

アプリケーションの中でモジュールを使うためには、アプリケーションの modules プロパティのリストにそのモジュールを載せてアプリケーションを構成するだけで大丈夫です。 アプリケーションの構成情報 の中の次のコードは、forum モジュールを使うようにするものです。

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... モジュールのその他の構成情報 ...
        ],
    ],
]
```

> 情報: モジュール内のコマンドを使用するためには、 モジュールを コンソール・アプリケーション設定 にも含める必要があります。

modules プロパティは、モジュールの構成情報の配列を取ります。 各配列のキーは、アプリケーションの全てのモジュールの中でそのモジュールを特定するためのユニークな モジュール *ID* を表します。 そして、対応する配列の値は、そのモジュールを作成するための 構成情報 です。

ルート

アプリケーションの中のコントローラをアクセスするのと同じように、ルート がモジュールの中のコントローラを指し示すために使われます。 モジュール内のコントローラのルートは、モジュール ID で始まり、コントローラ ID、 アクション ID と続くものでなければなりません。 例えば、アプリケーションが forum という名前のモジュールを使用している場合、forum/post/index というルートは、 forum モジュール内の post コントローラの index アクションを表します。 ルートがモジュール ID だけを含む場合は、yii\base\Module::$defaultRoute プロパティ (デフォルト値は default です) が、どのコントローラ・アクションが使用されるべきかを決定します。 これは、forum というルートは forum モジュール内の default コントローラを表すという意味です。

モジュールのための URL マネージャの規則は yii\web\UrlManager::parseRequest() が起動される前に追加されなくてはなりません。 す

なわち、モジュールの `init()` の中で規則を追加しても動作しない、ということです。なぜなら、モジュールの初期化はルートの処理が済んだ後になるからです。 従って、URL 規則は ブートストラップ段階 で追加される必要があります。 また、モジュールの URL 規則を `yii\web\GroupUrlRule` に入れるのも良い方法です。

モジュールが バージョン管理された API で使用される場合は、その URL 規則はアプリケーション構成情報の `urlManager` のセクションに直接に追加されなければなりません。

モジュールにアクセスする

モジュール内において、モジュール ID や、モジュールのパラメータ、モジュールのコンポーネントなどにアクセスするために、モジュール・クラス のインスタンスを取得する必要があることがよくあります。 次の文を使ってそうすることが出来ます。

```
$module = MyModuleClass::getInstance();
```

ここで `MyModuleClass` は、当該モジュール・クラスの名前を指すものです。 `getInstance()` メソッドは、現在リクエストされているモジュール・クラスのインスタンスを返します。 モジュールがリクエストされていない場合は、このメソッドは `null` を返します。モジュール・クラスの新しいインスタンスを手動で作成しようとしてはいけないことに注意してください。 手動で作成したインスタンスは、リクエストに対するレスポンスとして Yii によって作成されたインスタンスとは別のものになります。

> 情報: モジュールを開発するとき、モジュールが固定の ID を使うと仮定してはいけません。 なぜなら、モジュールは、アプリケーションや他のモジュールの中で使うときに、任意の ID と結び付けることが出来るからです。 モジュール ID を取得するためには、上記の方法を使って最初にモジュールのインスタンスを取得し、 そして `$module->id` によって ID を取得しなければなりません。

モジュールのインスタンスにアクセスするためには、次の二つの方法を使うことも出来ます。

```
// ID が "forum" である子モジュールを取得する
$module = \Yii::$app->getModule('forum');

// 現在リクエストされているコントローラが属するモジュールを取得する
$module = \Yii::$app->controller->module;
```

最初の方法は、モジュール ID を知っている時しか役に立ちません。一方、第二の方法は、 リクエストされているコントローラについて知っている場合に使うのに最適な方法です。

いったんモジュールのインスタンスをとらえれば、モジュールに登録されたパラメータやコンポーネントにアクセスすることが可能になります。例えば、

```
$maxPostCount = $module->params['maxPostCount'];
```

モジュールをブートストラップする

いくつかのモジュールは、全てのリクエストで毎回走らせる必要があります。yii\debug\Module・モジュールがその一例です。 そうするためには、そのようなモジュールをアプリケーションの bootstrap プロパティのリストに挙げます。

例えば、次のアプリケーションの構成情報は、debug モジュールが常にロードされることを保証するものです。

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

3.8.3 入れ子のモジュール

モジュールはレベルの制限無く入れ子にすることが出来ます。つまり、モジュールは別のモジュールを含むことが出来、その含まれたモジュールもさらに別のモジュールを含むことが出来ます。 含む側を 親モジュール、含まれる側を 子モジュール と呼びます。 子モジュールは、親モジュールの modules プロパティの中で宣言されなければなりません。例えば、

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->modules = [
            'admin' => [
                // ここはもっと短い名前空間の使用を考慮すべきです
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

入れ子にされたモジュールの中にあるコントローラのルートは、全ての祖先のモジュールの ID を含まなければなりません。 例えば、forum/admin/dashboard/index というルートは、forum モジュールの子モジュールである

admin モジュールの dashboard コントローラの index アクションを表します。

> 情報: getModule() メソッドは、親モジュールに直接属する子モジュールだけを返します。 yii\base\Application::$loadedModules プロパティがロードされた全てのモジュールのリストを保持しています。 このリストには、直接の子と孫以下の両方のモジュールが含まれ、クラス名によってインデックスされています。

3.8.4 モジュール内からコンポーネントにアクセスする

バージョン 2.0.13 以降、モジュールは ツリー走査 をサポートしています。 これによって、モジュールの開発者は(アプリケーション)コンポーネントを参照するのに、 サービス・ロケータとして自身のモジュールを使うことが出来るようになりました。 これが意味することは、Yii::$app->get('db') よりも $module->get('db') を使う方が良い、ということです。モジュールのユーザは、異なるコンポーネント(構成)が要求される場合は、モジュールで使用する特定のコンポーネントを指定することが出来ます。

例えば、次のような部分的なアプリケーション構成が可能です。

```
'components' => [
    'db' => [
        'tablePrefix' => 'main_',
        'class' => Connection::class,
        'enableQueryCache' => false
    ],
],
'modules' => [
    'mymodule' => [
        'components' => [
            'db' => [
                'tablePrefix' => 'module_',
                'class' => Connection::class
            ],
        ],
    ],
],
```

アプリケーションのデータベース・テーブルは main_ という接頭辞を持つ一方で、モジュールのデータベース・テーブルは module_ という接頭辞を持ちます。 上記の構成はマージされないということに注意して下さい。例えば、モジュールのコンポーネントではクエリ・キャッシュはデフォルト値に従って有効なままになります。

3.8.5 ベスト・プラクティス

モジュールは、それぞれ密接に関係する一連の機能を含む数個のグループに分割できるような、 規模の大きなアプリケーションに最も適してい

ます。 そのような機能グループをそれぞれモジュールとして、特定の個人やチームによって開発することが出来ます。

モジュールは、また、機能グループ・レベルでコードを再利用するための良い方法でもあります。 ある種のよく使われる機能、例えばユーザ管理やコメント管理などは、全て、将来のプロジェクトで容易に再利用できるように、 モジュールの形式で開発することが出来ます。

3.9 フィルタ

フィルタは、コントローラ・アクション の前 および/または 後に走るオブジェクトです。 例えば、アクセス・コントロール・フィルタはアクションの前に走って、アクションが特定のエンド・ユーザだけにアクセスを許可するものであることを保証します。 また、コンテント圧縮フィルタはアクションの後に走って、レスポンスのコンテントをエンド・ユーザに送出する前に圧縮します。

一つのフィルタは、前フィルタ (アクションの 前 に適用されるフィルタのロジック) および/または 後フィルタ (アクションの 後 に適用されるロジック) から構成することが出来ます。

3.9.1 フィルタを使用する

フィルタは、本質的には特別な種類の ビヘイビア です。 したがって、フィルタを使うことは ビヘイビアを使う ことと同じです。 下記のように、behaviors() メソッドをオーバーライドすることによって、コントローラの中でフィルタを宣言することが出来ます。

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index', 'view'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

デフォルトでは、コントローラ・クラスの中で宣言されたフィルタは、そのコントローラの 全て のアクションに適用されます。 しかし、only プロパティを構成することによって、 フィルタがどのアクションに適用されるべきかを明示的に指定することも出来ます。 上記の例では、HttpCache フィルタは、index と view のアクションに対してのみ適用されています。 また、except プロパティを構成して、いくつかのアクションをフィルタされないように除外することも可能です。

コントローラのほかに、モジュール または アプリケーション でもフィルタを宣言することが出来ます。 そのようにした場合、only と except のプロパティを上で説明したように構成しない限り、 そのフィルタは、モジュールまたはアプリケーションに属する 全て のコントローラ・アクションに適用されます。

> 補足: モジュールやアプリケーションでフィルタを宣言する場合、only と except のプロパティでは、 アクション ID ではなく、ルート を使わなければなりません。 なぜなら、モジュールやアプリケーションのスコープでは、アクション ID だけでは完全にアクションを指定することが出来ないからです。

一つのアクションに複数のフィルタが構成されている場合、フィルタは下記で説明されている規則に従って適用されます。

- 前フィルタ
  - アプリケーションで宣言されたフィルタを behaviors() にリストされた順に適用する。
  - モジュールで宣言されたフィルタを behaviors() にリストされた順に適用する。
  - コントローラで宣言されたフィルタを behaviors() にリストされた順に適用する。
  - フィルタのどれかがアクションをキャンセルすると、 そのフィルタの後のフィルタ (前フィルタと後フィルタの両方) は適用されない。
- 前フィルタを通過したら、アクションを走らせる。
- 後フィルタ
  - コントローラで宣言されたフィルタを behaviors() にリストされた逆順で適用する。
  - モジュールで宣言されたフィルタを behaviors() にリストされた逆順で適用する。
  - アプリケーションで宣言されたフィルタを behaviors() にリストされた逆順で適用する。

3.9.2 フィルタを作成する

新しいアクション・フィルタを作成するためには、yii\base\ActionFilter を拡張して、 beforeAction() および/または afterAction() メソッドをオーバーライドします。 前者はアクションが走る前に実行され、後者は走った後に実行されます。 beforeAction() の返り値が、アクションが実行されるべきか否かを決定します。 返り値が false である場合、このフィルタの後に続くフィルタはスキップされ、アクションは実行を中止されます。

次の例は、アクションの実行時間をログに記録するフィルタを示すものです。

```
namespace app\components;

use Yii;
use yii\base\ActionFilter;

class ActionTimeFilter extends ActionFilter
{
    private $_startTime;

    public function beforeAction($action)
    {
        $this->_startTime = microtime(true);
        return parent::beforeAction($action);
    }

    public function afterAction($action, $result)
    {
        $time = microtime(true) - $this->_startTime;
        Yii::debug("アクション '{$action->uniqueId}' は $time 秒を消費。");
        return parent::afterAction($action, $result);
    }
}
```

3.9.3 コアのフィルタ

Yii はよく使われる一連のフィルタを提供しており、それらは、主として yii\filters 名前空間の下にあります。 以下では、それらのフィルタを簡単に紹介します。

AccessControl

AccessControl は、一組の 規則 に基づいて、シンプルなアクセス・コントロールを提供するものです。 具体的に言うと、アクションが実行される前に、AccessControl はリストされた規則を調べて、 現在のコンテキスト変数 (例えば、ユーザの IP アドレスや、ユーザのログイン状態など) に最初に合致するものを見つけます。 そして、合致した規則によって、リクエストされたアクションの実行を許可するか拒否するかを決定します。 合致する規則がなかった場合は、アクセスは拒否されます。

次の例は、認証されたユーザに対しては create と update のアクションへのアクセスを許可し、 その他のすべてのユーザにはこれら二つのアクションに対するアクセスを拒否する仕方を示すものです。

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['create', 'update'],
```

```
            'rules' => [
                // 認証されたユーザに許可する
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // その他はすべてデフォルトにより拒否される
            ],
        ],
    ];
}
```

アクセス・コントロール一般についての詳細は 権限 のセクションを参照してください。

認証メソッド・フィルタ

認証メソッド・フィルタは、HTTP Basic 認証[17]、 OAuth 2[18] など、様々なメソッドを使ってユーザを認証するために使われるものです。 これらのフィルタ・クラスはすべて `yii\filters\auth` 名前空間の下にあります。

次の例は、`yii\filters\auth\HttpBasicAuth` の使い方を示すもので、HTTP Basic 認証に基づくアクセス・トークンを使ってユーザを認証しています。 これを動作させるためには、あなたの ユーザ・アイデンティティ・クラス が `findIdentityByAccessToken()` メソッドを実装していなければならないことに注意してください。

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::className(),
        ],
    ];
}
```

認証メソッド・フィルタは RESTful API を実装するときに使われるのが通例です。 詳細については、RESTful の 認証 のセクションを参照してください。

ContentNegotiator

ContentNegotiator は、レスポンス形式のネゴシエーションとアプリケーション言語のネゴシエーションをサポートします。 このフィルタは `GET` パラメータと `Accept` HTTP ヘッダを調べることによって、レスポンス形式 および/または 言語を決定しようとします。

[17] http://ja.wikipedia.org/wiki/Basic%E8%AA%8D%E8%A8%BC
[18] http://oauth.net/2/

次の例では、ContentNegotiator はレスポンス形式として JSON と XML をサポートし、 (合衆国の)英語とドイツ語を言語としてサポートするように構成されています。

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

public function behaviors()
{
    return [
        [
            'class' => ContentNegotiator::className(),
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ];
}
```

レスポンス形式と言語は アプリケーションのライフサイクル のもっと早い段階で決定される必要があることがよくあります。 このため、ContentNegotiator はフィルタの他に、ブートストラップ・コンポーネント としても使うことができるように設計されています。 例えば、次のように、ContentNegotiator を アプリケーションの構成情報 の中で構成することが出来ます。

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

[
    'bootstrap' => [
        [
            'class' => ContentNegotiator::className(),
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ],
];
```

> 情報: 望ましいコンテント・タイプと言語がリクエストから決定できない場合は、 `formats` および `languages` に挙げられている最初の形式と言語が使用されます。

HttpCache

HttpCache は `Last-Modified` および `Etag` の HTTP ヘッダを利用して、クライアント・サイドのキャッシュを実装するものです。 例えば、

```
use yii\filters\HttpCache;

public function behaviors()
{
    return [
        [
            'class' => HttpCache::className(),
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

HttpCache に関する詳細は HTTP キャッシュ のセクションを参照してください。

PageCache

PageCache はサーバ・サイドにおけるページ全体のキャッシュを実装するものです。 次の例では、PageCache が `index` アクションに適用されて、最大 60 秒間、または、`post` テーブルのエントリ数が変化するまでの間、ページ全体をキャッシュしています。 さらに、このページ・キャッシュは、選択されたアプリケーションの言語に従って、違うバージョンのページを保存するようにしています。

```
use yii\filters\PageCache;
use yii\caching\DbDependency;

public function behaviors()
{
    return [
        'pageCache' => [
            'class' => PageCache::className(),
            'only' => ['index'],
            'duration' => 60,
            'dependency' => [
                'class' => DbDependency::className(),
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
            'variations' => [
                \Yii::$app->language,
            ]
        ],
    ];
}
```

PageCache の使用に関する詳細は ページ・キャッシュ のセクションを参照してください。

RateLimiter

RateLimiter は リーキー・バケット・アルゴリズム[19] に基づいてレート制限のアルゴリズムを実装するものです。 主として RESTful API を実装するときに使用されます。 このフィルタの使用に関する詳細は レート制限 のセクションを参照してください。

VerbFilter

VerbFilter は、HTTP リクエスト・メソッド (HTTP 動詞) がリクエストされたアクションによって許可されているかどうかをチェックするものです。 許可されていない場合は、HTTP 405 例外を投げます。 次の例では、VerbFilter が宣言されて、CRUD アクションに対して許可されるメソッドの典型的なセットを指定しています。

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'index'  => ['get'],
                'view'   => ['get'],
                'create' => ['get', 'post'],
                'update' => ['get', 'put', 'post'],
                'delete' => ['post', 'delete'],
            ],
        ],
    ];
}
```

Cors

クロス・オリジン・リソース共有 CORS[20] とは、ウェブ・ページにおいて、さまざまなリソース (例えば、フォントや JavaScript など) を、それを生成するドメイン以外のドメインからリクエストすることを可能にするメカニズムです。 特に言えば、JavaScript の AJAX 呼出しが使用することが出来る XMLHttpRequest メカニズムです。 このような「クロス・ドメイン」のリクエストは、このメカニズムに拠らなければ、 同一生成元のセキュリティ・ポリシーによって、ウェブ・ブラウザから禁止される

[19] http://ja.wikipedia.org/wiki/%E3%83%AA%E3%83%BC%E3%82%AD%E3%83%BC%E3%83%90%E3%82%B1%E3%83%83%E3%83%88

[20] https://developer.mozilla.org/ja/docs/HTTP_access_control

はずのものです。 CORS は、ブラウザとサーバが交信して、クロス・ドメインのリクエストを許可するか否かを決定する方法を定義するものです。

Cors フィルタ は、CORS ヘッダが常に送信されることを保証するために、 Authentication / Authorization のフィルタよりも前に定義されなければなりません。

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
        ],
    ], parent::behaviors());
}
```

あなたの API の yii\rest\ActiveController クラスに CORS フィルタを追加したい場合は、 REST コントローラ のセクションも参照して下さい。

Cors のフィルタリングは $cors プロパティを使ってチューニングすることが出来ます。

- cors['Origin']: 許可される生成元を定義するのに使われる配列。['*'] (すべて) または ['http://www.myserver.net', 'http://www.myotherserver.com'] などが設定可能。デフォルトは ['*']。
- cors['Access-Control-Request-Method']: 許可される HTTP 動詞の配列。たとえば、['GET', 'OPTIONS', 'HEAD']。デフォルトは ['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']。
- cors['Access-Control-Request-Headers']: 許可されるヘッダの配列。全てのヘッダを意味する ['*'] または特定のヘッダを示す ['X-Request-With'] が設定可能。デフォルトは ['*']。
- cors['Access-Control-Allow-Credentials']: 現在のリクエストをクレデンシャルを使ってすることが出来るかどうかを定義。true、false または null (設定なし) が設定可能。デフォルトは null。
- cors['Access-Control-Max-Age']: プリフライト・リクエストの寿命を定義。デフォルトは 86400。

次の例は、生成元 http://www.myserver.net に対する GET、HEAD および OPTIONS のメソッドによる CORS を許可するものです。

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
```

```
            'cors' => [
                'Origin' => ['http://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'
    ],
            ],
        ],
    ], parent::behaviors());
}
```

デフォルトのパラメータをアクション単位でオーバーライドして CORS ヘッダをチューニングすることも可能です。 例えば、`login` アクションに `Access-Control-Allow-Credentials` を追加することは、次のようにすれば出来ます。

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
            'cors' => [
                'Origin' => ['http://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'
    ],
            ],
            'actions' => [
                'login' => [
                    'Access-Control-Allow-Credentials' => true,
                ]
            ]
        ],
    ], parent::behaviors());
}
```

3.10 ウィジェット

ウィジェットは、ビュー で使用される再利用可能な構成ブロックで、複雑かつ構成可能なユーザ・インタフェイス要素をオブジェクト指向の流儀で作成するためのものです。 例えば、日付選択ウィジェットを使うと、入力として日付を選択することを可能にする素敵なデイト・ピッカーを生成することが出来ます。 このとき、あなたがしなければならないことは、次のようなコードをビューに挿入することだけです:

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>
```

数多くのウィジェットが Yii にバンドルされています。 例えば、アクティブ・フォーム や、メニュー、jQuery UI ウィジェット[21]、Twitter Bootstrap ウィジェット[22] などです。 下記では、ウィジェットに関する基本的な知識の手引きをします。 特定のウィジェットの使い方について学ぶ必要がある場合は、クラス API ドキュメントを参照してください。

3.10.1 ウィジェットを使う

ウィジェットは主として ビュー で使われます。 `yii\base\Widget::widget()` メソッドを呼んで、ビューでウィジェットを使います。 このメソッドは、ウィジェットを初期化するための 構成情報 配列を受け取り、ウィジェットのレンダリング結果を返します。 例えば、下記のコードは、日本語を使い、入力を `$model` の `from_date` 属性に保存するように構成された日付選択ウィジェットを挿入するものです。

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ja',
    'dateFormat' => 'php:Y-m-d',
]) ?>
```

ウィジェットの中には、コンテントのブロックを受け取ることが出来るものもあります。 その場合、コンテントのブロックは `yii\base\Widget::begin()` と `yii\base\Widget::end()` の呼び出しで囲むようにしなければなりません。 例えば、次のコードは `yii\widgets\ActiveForm` ウィジェットを使ってログイン・フォームを生成するものです。 このウィジェットは、`begin()` と `end()` が呼ばれる場所で、それぞれ、開始と終了の `<form>` タグを生成します。 その間に置かれたものは全てそのままレンダリングされます。

```
<?php
use yii\widgets\ActiveForm;
use yii\helpers\Html;
?>

<?php $form = ActiveForm::begin(['id' => 'login-form']); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <?= Html::submitButton('ログイン') ?>
    </div>
```

[21] https://www.yiiframework.com/extension/yiisoft/yii2-jui
[22] https://www.yiiframework.com/extension/yiisoft/yii2-bootstrap

```
<?php ActiveForm::end(); ?>
```

yii\base\Widget::widget() がウィジェットのレンダリング結果を返すのとは違って、yii\base\Widget::begin() メソッドがウィジェットのインスタンスを返すことに注意してください。 返されたウィジェットのインスタンスを使って、ウィジェットのコンテントを構築することが出来ます。

> 補足: いくつかのウィジェットは、yii\base\Widget::end() が呼ばれるときに囲んだコンテンツを調整するため、 出力バッファリング[23] を使用します。 この理由から、yii\base\Widget::begin() と yii\base\Widget::end() の呼び出しは、同じビュー・ファイルの中で発生するものと想定されています。 この規則に従わない場合は、予期しない出力結果が生じ得ます。

グローバルなデフォルトを構成する

あるタイプのウィジェットのグローバルなデフォルトを DI コンテナによって構成することが出来ます。

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

詳細については 依存注入コンテナのガイドの “実際の使いかた” のセクション を参照してください。

3.10.2 ウィジェットを作成する

ウィジェットは必要に従って二つの異なる方法で作成することが出来ます。

1: widget() メソッドを利用する

ウィジェットを作成するためには、yii\base\Widget を拡張して、yii\base\Widget::init() および/または yii\base\Widget::run() メソッドをオーバーライドします。 通常、init() メソッドはウィジェットのプロパティを初期化するコードを含むべきものであり、run() メソッドはウィジェットのレンダリング結果を生成するコードを含むべきものです。 レンダリング結果は、直接に “echo” しても、run() の返り値として文字列として返しても構いません。

次の例では、HelloWidget が message プロパティとして割り当てられたコンテントを HTML エンコードして表示します。 プロパティがセットされていない場合は、デフォルトとして “Hello World” を表示します。

[23] https://secure.php.net/manual/ja/book.outcontrol.php

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public $message;

    public function init()
    {
        parent::init();
        if ($this->message === null) {
            $this->message = 'Hello World';
        }
    }

    public function run()
    {
        return Html::encode($this->message);
    }
}
```

このウィジェットを使うために必要なことは、次のコードをビューに挿入するだけのことです。

```
<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'おはよう']) ?>
```

ウィジェットが大きなかたまりのコンテントをレンダーする必要がある場合もあります。 コンテントを `run()` の中に埋め込むことも出来ますが、もっと良い方法は、コンテントを view に置き、 `yii\base\Widget::render()` を呼んでレンダーする方法です。例えば、

```
public function run()
{
    return $this->render('hello');
}
```

2: `begin()` と `end()` のメソッドを利用する

これは上記のものと少し異なるだけのものです。 下記は `HelloWidget` の変種で、`begin()` と `end()` の間に包まれたコンテントを受け取り、 それをHTML エンコードして表示するものです。

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
```

```
{
    public function init()
    {
        parent::init();
        ob_start();
    }

    public function run()
    {
        $content = ob_get_clean();
        return Html::encode($content);
    }
}
```

ご覧のように、init() の中で PHP の出力バッファが開始され、init() と run() の呼び出しの間の全ての出力がキャプチャされ、 run() の中で処理されて返されます。

> 情報: yii\base\Widget::begin() を呼ぶと、ウィジェットの新しいインスタンスが作成され、 ウィジェットのコンストラクタの最後で init() メソッドが呼ばれます。 yii\base\Widget::end() を呼ぶと、run() メソッドが呼ばれて、その返り値が end() によって echo されます。

次のコードは、この HelloWidget の新しい変種をどのように使うかを示すものです:

```
<?php
use app\components\HelloWidget;
?>
<?php HelloWidget::begin(); ?>

    一つまたはそれ以上の <strong>HTML</strong> <preタグ></pre> を含みうるサンプル・コンテント

    このコンテントが大きくなりすぎる場合は、サブ・ビューを使います。

    例えば、

    <?php echo $this->render('viewfile'); // 注意: この render() メソッドは \yii\base\View クラスのもの。コードのこの部分はビュー・ファイルの中にあり、Widget クラス・ファイルの中にはない ?>

<?php HelloWidget::end(); ?>
```

デフォルトでは、ウィジェット用のビューは WidgetPath/views ディレクトリの中のファイルに保存すべきものです。 ここで WidgetPath はウィジェットのクラス・ファイルを含むディレクトリを指します。したがって、上記の例では、ウィジェット・クラスが @app/components に配置されていると仮定すると、@app/components/views/hello.php というビュー・ファイルがレンダリングされることになります。 yii\base

`\Widget::getViewPath()` メソッドをオーバーライドして、ウィジェットのビュー・ファイルを含むディレクトリをカスタマイズすることが出来ます。

3.10.3 ベスト・プラクティス

ウィジェットはビューのコードを再利用するためのオブジェクト指向の方法です。

ウィジェットを作成するときでも、MVC パターンに従うべきです。一般的に言うと、ロジックはウィジェット・クラスに保持し、表現はビュー に保持すべきです。

ウィジェットは自己完結的に設計されるべきです。 言い換えると、ウィジェットを使うときに、他に何もしないでもビューに挿入することが出来るようにすべきです。 この要求は、ウィジェットがCSS、JavaScript、画像などの外部リソースを必要とする場合は、扱いにくい問題になり得ます。 幸いなことに、Yii はこの問題を解決するのに利用することが出来る アセット・バンドル のサポートを提供しています。

ウィジェットがビュー・コードだけを含む場合は、ビュー と非常に似たものになります。 実際のところ、この場合、両者の唯一の違いは、ウィジェットが再配布可能なクラスである一方で、 ビューはアプリケーション内に保持することが望ましい素の PHP スクリプトである、というぐらいの事です。

3.11 アセット

Yii では、アセットは、ウェブ・ページで参照できるファイルを意味します。アセットは CSS ファイルであったり、JavaScript ファイルであったり、画像やビデオのファイルであったりします。 アセットはウェブでアクセス可能なディレクトリに配置され、ウェブ・サーバによって直接に提供されます。

たいていの場合、アセットはプログラム的に管理する方が望ましいものです。 例えば、ページの中で `yii\jui\DatePicker` ウィジェットを使うとき、このウィジェットは必要な CSS と JavaScript のファイルを自動的にインクルードします。 あなたに対して、手作業で必要なファイルを探してインクルードするように要求したりはしません。 そして、このウィジェットを新しいバージョンにアップグレードしたときは、自動的に新しいバージョンのアセット・ファイルが使用されるようになります。 このチュートリアルでは、Yii によって提供される強力なアセット管理機能について説明します。

3.11.1 アセット・バンドル

Yii はアセットを アセット・バンドル を単位として管理します。アセット・バンドルは、簡単に言えば、 あるディレクトリの下に集められた一

群のアセットです。 ビュー の中でアセット・バンドルを登録すると、バンドルの中の CSS や JavaScript のファイルがレンダリングされるウェブ・ページに挿入されます。

3.11.2 アセット・バンドルを定義する

アセット・バンドルは yii\web\AssetBundle から拡張された PHP クラスとして定義されます。 バンドルの名前は、対応する PHP クラスの完全修飾名 (先頭のバック・スラッシュを除く) です。 アセット・バンドル・クラスは オートロード可能 でなければなりません。 アセット・バンドル・クラスは、通常、アセットがどこに置かれているか、バンドルがどういう CSS や JavaScript のファイルを含んでいるか、そして、バンドルが他のバンドルにどのように依存しているかを定義します。

以下のコードは ベーシック・プロジェクト・テンプレート によって使用されているメインのアセット・バンドルを定義するものです。

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
        ['css/print.css', 'media' => 'print'],
    ];
    public $js = [
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

上の AppAsset クラスは、アセット・ファイルが @webroot ディレクトリの下に配置されており、それが URL @web に対応することを定義しています。バンドルは一つだけ CSS ファイル css/site.css を含み、JavaScript ファイルは含みません。 バンドルは、他の二つのバンドル、すなわち yii\web\YiiAsset と yii\bootstrap\BootstrapAsset に依存しています。以下、yii\web\AssetBundle のプロパティに関して、更に詳細に説明します。

- sourcePath: このバンドルのアセット・ファイルを含むルート・ディレクトリを指定します。 ルート・ディレクトリがウェブ・アクセス可能でない場合に、このプロパティをセットしなければなりません。 そうでない場合は、代りに、basePath と baseUrl のプロパ

ティをセットしなければなりません。 パス・エイリアス をここで使うことが出来ます。

- basePath: このバンドルのアセット・ファイルを含むウェブ・アクセス可能なディレクトリを指定します。 sourcePath プロパティをセットした場合は、アセット・マネージャ がバンドルに含まれるファイルをウェブ・アクセス可能なディレクトリに発行して、 その結果、このプロパティを上書きします。 アセット・ファイルが既にウェブ・アクセス可能なディレクトリにあり、アセットの発行が必要でない場合に、このプロパティをセットしなければなりません。 パス・エイリアス をここで使うことが出来ます。
- baseUrl: basePath ディレクトリに対応する URL を指定します。 basePath と同じように、sourcePath プロパティをセットした場合は、 アセット・マネージャ がアセットを発行して、その結果、このプロパティを上書きします。 パス・エイリアス をここで使うことが出来ます。
- css: このバンドルに含まれている CSS ファイルをリストする配列です。 ディレクトリの区切りとしてフォワード・スラッシュ "/" だけを使わなければならないことに注意してください。 それぞれのファイルは、個別に、パス文字列、または、パス文字列と属性のタグと値を一緒に含む配列によって指定することが出来ます。
- js: このバンドルに含まれる JavaScript ファイルをリストする配列です。 この配列の形式は、css の配列の形式と同じです。 それぞれの JavaScript ファイルは、以下の二つの形式のどちらかによって指定することが出来ます。
  - ローカルの JavaScript ファイルを表す相対パス (例えば js/main.js)。 実際のファイルのパスは、この相対パスの前に yii\web\AssetManager::$basePath を付けることによって決定されます。 また、実際の URL は、この相対パスの前に yii\web\AssetManager::$baseUrl を付けることによって決定されます。
  - 外部の JavaScript ファイルを表す絶対 URL。 例えば、http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js や //ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js など。
- depends: このバンドルが依存しているアセット・バンドルの名前をリストする配列です (バンドルの依存関係については、すぐ後で説明します)。
- jsOptions: このバンドルにある 全て の JavaScript ファイルについて、 それを登録するときに呼ばれる yii\web\View::registerJsFile() メソッドに渡されるオプションを指定します。
- cssOptions: このバンドルにある 全て の CSS ファイルについて、それを登録するときに呼ばれる yii\web\View::registerCssFile() メソッドに渡されるオプションを指定します。
- publishOptions: ソースのアセット・ファイルをウェブ・ディレクトリに発行するときに呼ばれる yii\web\AssetManager::publish()

メソッドに渡されるオプションを指定します。 これは sourcePath プロパティを指定した場合にだけ使用されます。

アセットの配置場所

アセットは、配置場所を基準にして、次のように分類することが出来ます。

- ソース・アセット: アセット・ファイルは、ウェブ経由で直接にアクセスすることが出来ない PHP ソース・コードと一緒に配置されています。 ページの中でソース・アセットを使用するためには、ウェブ・ディレクトリにコピーして、いわゆる発行されたアセットに変換しなければなりません。 このプロセスは、すぐ後で詳しく説明しますが、アセット発行 と呼ばれます。
- 発行されたアセット: アセット・ファイルはウェブ・ディレクトリに配置されており、したがってウェブ経由で直接にアクセスすることが出来ます。
- 外部アセット: アセット・ファイルは、あなたのウェブ・アプリケーションをホストしているのとは別のウェブ・サーバ上に 配置されています。

アセット・バンドル・クラスを定義するときに、sourcePath プロパティを指定した場合は、 相対パスを使ってリストに挙げられたアセットは全てソース・アセットであると見なされます。 このプロパティを指定しなかった場合は、アセットは発行されたアセットであることになります (したがって、basePath と baseUrl を指定して、アセットがどこに配置されているかを Yii に知らせなければなりません)。

アプリケーションに属するアセットは、不要なアセット発行プロセスを避けるために、ウェブ・ディレクトリに置くことが推奨されます。 前述の例において AppAsset が sourcePath ではなく basePath を指定しているのは、これが理由です。

エクステンション の場合は、 アセットがソース・コードと一緒にウェブからアクセス出来ないディレクトリに配置されているため、 アセット・バンドル・クラスを定義するときには sourcePath プロパティを指定しなければなりません。

> 補足: @webroot/assets を ソース・パス として使ってはいけません。 このディレクトリは、デフォルトでは、アセット・マネージャ がソースの配置場所から発行されたアセット・ファイルを保存する場所として使われます。 このディレクトリの中のファイルはすべて一時的なものと見なされており、 削除されることがあります。

アセットの依存関係

ウェブ・ページに複数の CSS や JavaScript ファイルをインクルードするときは、オーバーライドの問題を避けるために、一定の順序に従わなけ

ればなりません。 例えば、ウェブ・ページで jQuery UI ウィジェットを使おうとするときは、jQuery JavaScript ファイルが jQuery UI JavaScript ファイルより前にインクルードされることを保証しなければなりません。 このような順序付けをアセット間の依存関係と呼びます。

アセットの依存関係は、主として、`yii\web\AssetBundle::$depends` プロパティによって指定されます。 `AppAsset` の例では、このアセット・バンドルは他の二つのアセット・バンドル、すなわち、`yii\web\YiiAsset` と `yii\bootstrap\BootstrapAsset` に依存しています。 このことは、`AppAsset` の CSS と JavaScript ファイルが、依存している二つのアセット・バンドルにあるファイルの 後に インクルードされることを意味します。

アセットの依存関係は中継されます。つまり、バンドル A が B に依存し、B が C に依存していると、A は C にも依存していることになります。

アセットのオプション

`cssOptions` および `jsOptions` のプロパティを指定して、 CSS と JavaScript ファイルがページにインクルードされる方法をカスタマイズすることが出来ます。 これらのプロパティの値は、ビュー が CSS と JavaScript ファイルをインクルードするために、`yii\web\View::registerCssFile()` および `yii\web\View::registerJsFile()` メソッドを呼ぶときに、それぞれ、オプションとして引き渡されます。

> 補足: バンドル・クラスでセットしたオプションは、バンドルの中の 全て の CSS/JavaScript ファイルに適用されます。 いろいろなファイルに別々のオプションを使用したい場合は、上述した [[yii\web\AssetBundle::css|css] の形式を使うか、 または、別々のアセット・バンドルを作成して、個々のバンドルの中では、一組のオプションを使うようにしなければなりません。

例えば、IE9 以下のブラウザに対して CSS ファイルを条件的にインクルードするために、次のオプションを使うことが出来ます。

```
public $cssOptions = ['condition' => 'lte IE9'];
```

こうすると、バンドルの中の CSS ファイルは下記の HTML タグを使ってインクルードされるようになります。

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

生成された CSS のリンクタグを `<noscript>` の中に包むためには、次のように `cssOptions` を構成することが出来ます。

```
public $cssOptions = ['noscript' => true];
```

JavaScript ファイルをページの head セクションにインクルードするためには、次のオプションを使います (デフォルトでは、JavaScript ファイルは body セクションの最後にインクルードされます)。

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

デフォルトでは、アセット・バンドルが発行されるときは、`yii\web\AssetBundle::$sourcePath` で指定されたディレクトリの中にある全てのコンテントが発行されます。 `publishOptions` プロパティを構成することによって、この振る舞いをカスタマイズすることが出来ます。 例えば、`yii\web\AssetBundle::$sourcePath` の一個または数個のサブ・ディレクトリだけを発行するために、アセット・バンドル・クラスの中で下記のようにすることが出来ます。

```
<?php
namespace app\assets;

use yii\web\AssetBundle;

class FontAwesomeAsset extends AssetBundle
{
    public $sourcePath = '@bower/font-awesome';
    public $css = [
        'css/font-awesome.min.css',
    ];
    public $publishOptions = [
        'only' => [
            'fonts/*',
            'css/*',
        ]
    ];
}
```

上記の例は、"fontawesome" パッケージ[24] のためのアセット・バンドルを定義するものです。 発行オプション `only` を指定して、`fonts` と `css` サブ・ディレクトリだけが発行されるようにしています。

Bower と NPM のアセットのインストール

ほとんどの JavaScript/CSS パッケージは、Bower[25] および/または NPM[26] によって管理されています。 PHP の世界には PHP の依存を管理する Composer がありますが、PHP のパッケージと全く同じように `composer.json` を使って Bower のパッケージも NPM のパッケージもロードすることが可能です。

このことを達成するために Composer の構成を少し修正しなければなりません。二つの方法があります。

---

[24] http://fontawesome.io/
[25] http://bower.io/
[26] https://www.npmjs.org/

asset-packagist レポジトリを使う　この方法は NPM または Bower のパッケージを必要とするプロジェクトの大半の要求を満たすことが出来ます。

> 補足: 2.0.13 以降、ベーシック・アプリケーション・テンプレートとアドバンスト・アプリケーション・テンプレートはともに、 デフォルトで asset-packagist を使うように前もって構成されていますので、このセクションは読み飛ばすことが出来ます。

プロジェクトの `composer.json` に、下記を追加します。

```
"repositories": [
    {
        "type": "composer",
        "url": "https://asset-packagist.org"
    }
]
```

アプリケーション構成情報 の `@npm` と `@bower` の エイリアス を修正します。

```
$config = [
    ...
    'aliases' => [
        '@bower' => '@vendor/bower-asset',
        '@npm'   => '@vendor/npm-asset',
    ],
    ...
];
```

asset-packagist がどのようにして動作するのかは、asset-packagist.org のサイト[27] に説明があります。

fxp/composer-asset-plugin を使う　asset-packagist と異なって、composer-asset-plugin はアプリケーション構成の変更を少しも要求しません。 その代りに、次のコマンドを実行して特別な Composer プラグインをグローバルにインストールすることが要求されます。

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
```

このコマンドによって composer asset plugin[28] がグローバルにインストールされ、 Bower と NPM パッケージの依存関係を Composer によって管理することが出来るようになります。 プラグインがインストールされた後は、あなたのコンピュータ上の全てのプロジェクトが `composer.json` を通じて Bower と NPM パッケージをサポートすることが出来ます。

Yii を使ってこれらのアセットを発行したい場合は、プロジェクトの `composer.json` に下記を追加して、 インストールされるパッケージが配置されるディレクトリを調整します。

[27] https://asset-packagist.org
[28] https://github.com/francoispluchino/composer-asset-plugin/

```
"config": {
    "fxp-asset": {
        "installer-paths": {
            "npm-asset-library": "vendor/npm",
            "bower-asset-library": "vendor/bower"
        }
    }
}
```

補足: `fxp/composer-asset-plugin` は、asset-packagist に比べて、`composer update` コマンドを著しく遅くします。

Composer で Bower と NPM をサポートできるように構成した後は:

1. アプリケーションまたはエクステンションの `composer.json` ファイルを修正して、パッケージを `require` のエントリに入れます。 ライブラリを参照するのに、`bower-asset/PackageName` (Bower パッケージ) または `npm-asset/PackageName` (NPM パッケージ) を使わなければなりません。

2. `composer update` を実行します。

3. アセット・バンドル・クラスを作成して、アプリケーションまたはエクステンションで使う予定の JavaScript/CSS ファイルをリストに挙げます。 `sourcePath` プロパティは、`@bower/PackageName` または `@npm/PackageName` としなければなりません。 これは、Composer が Bower または NPM パッケージを、このエイリアスに対応するディレクトリにインストールするためです。

   補足: パッケージの中には、全ての配布ファイルをサブ・ディレクトリに置くものがあります。 その場合には、そのサブ・ディレクトリを `sourcePath` の値として指定しなければなりません。 例えば、`yii\web\JqueryAsset` は `@bower/jquery` ではなく `@bower/jquery/dist` を使います。

3.11.3 アセット・バンドルを使う

アセット・バンドルを使うためには、`yii\web\AssetBundle::register()` メソッドを呼んでアセット・バンドルを ビュー に登録します。 例えば、次のようにしてビュー・テンプレートの中でアセット・バンドルを登録することが出来ます。

```
use app\assets\AppAsset;
AppAsset::register($this);  // $this はビュー・オブジェクトを表す
```

> 情報: yii\web\AssetBundle::register() メソッドは、basePath や baseUrl など、 発行されたアセットに関する情報を含むアセット・バンドル・オブジェクトを返します。

他の場所でアセット・バンドルを登録しようとするときは、必要とされるビュー・オブジェクトを提供しなければなりません。 例えば、ウィジェット・クラスの中でアセット・バンドルを登録するためには、$this->view によってビュー・オブジェクトを取得することが出来ます。

アセット・バンドルがビューに登録されるとき、舞台裏では、依存している全てのアセット・バンドルが Yii によって登録されます。 そして、アセット・バンドルがウェブからはアクセス出来ないディレクトリに配置されている場合は、 アセット・バンドルがウェブ・ディレクトリに発行されます。 その後、ビューがページをレンダリングするときに、登録されたバンドルのリストに挙げられている CSS と JavaScript ファイルのための <link> タグと <script> タグが生成されます。 これらのタグの順序は、登録されたバンドル間の依存関係、および、yii\web\AssetBundle::$css と yii\web\AssetBundle::$js のプロパティのリストに挙げられたアセットの順序によって決定されます。

動的なアセット・バンドル

アセット・バンドルは、通常の PHP クラスですので、内部のパラメータを動的に調整する追加のロジックを持つことが出来ます。 例えば、洗練された JavaScript ライブラリには、国際化の機能を、サポートする言語ごとに独立したソース・ファイルにパッケージして提供しているものもあります。 その場合、ライブラリの翻訳を動作させるためには、特定の '.js' ファイルをページに追加しなければなりません。 このことを yii\web\AssetBundle::init() メソッドをオーバーライドすることによって実現することが出来ます。

```
namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class SophisticatedAssetBundle extends AssetBundle
{
    public $sourcePath = '/path/to/sophisticated/src';
    public $js = [
        'sophisticated.js' // 常に使用されるファイル
    ];

    public function init()
    {
        parent::init();
        $this->js[] = 'i18n/' . Yii::$app->language . '.js'; // 動的に追加されるファイル
    }
```

```
}
```

個々のアセット・バンドルは、 yii\web\AssetBundle::register() によって返されるインスタンスによって調整することも出来ます。 例えば、

```
use app\assets\SophisticatedAssetBundle;
use Yii;

$bundle = SophisticatedAssetBundle::register(Yii::$app->view);
$bundle->js[] = 'i18n/' . Yii::$app->language . '.js'; // 動的に追加される
    ファイル
```

> 補足: アセット・バンドルの動的な調整はサポートされてはいますが、推奨はされません。 予期しない副作用を引き起こしやすいので、可能であれば避けるべきです。

アセット・バンドルをカスタマイズする

Yii は、yii\web\AssetManager によって実装されている assetManager という名前のアプリケーション・コンポーネントを通じてアセット・バンドルを管理します。 yii\web\AssetManager::$bundles プロパティを構成することによって、アセット・バンドルの振る舞いをカスタマイズすることが出来ます。 例えば、デフォルトの yii\web\JqueryAsset アセット・バンドルはインストールされた jQuery の Bower パッケージにある jquery.js ファイルを使用します。 あなたは、可用性とパフォーマンスを向上させるために、Google によってホストされたバージョンを使いたいと思うかも知れません。 次のように、アプリケーションの構成情報で assetManager を構成することによって、それが達成できます。

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                    'sourcePath' => null,   // バンドルを発行しない
                    'js' => [
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery
    .min.js',
                    ]
                ],
            ],
        ],
    ],
];
```

複数のアセット・バンドルも同様に yii\web\AssetManager::$bundles によって構成することが出来ます。 配列のキーは、アセット・バンドルのクラス名 (最初のバック・スラッシュを除く) とし、 配列の値は、対応する 構成情報配列 とします。

ヒント: アセット・バンドルの中で使うアセットを条件的に選択することが出来ます。 次の例は、開発環境では jquery.js を使い、そうでなければ jquery.min.js を使う方法を示すものです。

```
'yii\web\JqueryAsset' => [
    'js' => [
        YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
    ]
],
```

無効にしたいアセット・バンドルの名前に false を結びつけることによって、一つまたは複数のアセット・バンドルを無効にすることが出来ます。 無効にされたアセット・バンドルをビューに登録した場合は、依存するバンドルは一つも登録されません。 また、ビューがページをレンダリングするときも、バンドルの中のアセットは一つもインクルードされません。 例えば、yii\web\JqueryAsset を無効化するために、次の構成情報を使用することが出来ます。

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => false,
            ],
        ],
    ],
];
```

yii\web\AssetManager::$bundles を false にセットすることによって、全て のバンドルを無効にすることも出来ます。

yii\web\AssetManager::$bundles によってなされたカスタマイズはアセット・バンドルの生成時、すなわち、オブジェクトのコンストラクタの段階で適用される、 ということを心に留めてください。 従って、yii\web\AssetManager::$bundles のレベルで設定されたマッピングは、それ以後にバンドルのオブジェクトに対してなされる修正によって上書きされます。 具体的に言えば、yii\web\AssetBundle::init() メソッドの中での修正や、登録されたバンドル・オブジェクトに対する修正は、AssetManager の構成よりも優先されます。 以下に、yii\web\AssetManager::$bundles によって設定されたマッピングが効力を持たない例を示します。

```
// プログラムのソース・コード

namespace app\assets;

use yii\web\AssetBundle;
use Yii;
```

```
class LanguageAssetBundle extends AssetBundle
{
    // ...

    public function init()
    {
        parent::init();
        $this->baseUrl = '@web/i18n/' . Yii::$app->language; // AssetManager
    ‘ では処理出来な
    い!
    }
}
// ...

$bundle = \app\assets\LargeFileAssetBundle::register(Yii::$app->view);
$bundle->baseUrl = YII_DEBUG ? '@web/large-files': '@web/large-files/
    minified'; // AssetManager‘ では処理出来な
    い!


// アプリケーション構成

return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'app\assets\LanguageAssetBundle' => [
                    'baseUrl' => 'http://some.cdn.com/files/i18n/en' // 効力
    を持たない!
                ],
                'app\assets\LargeFileAssetBundle' => [
                    'baseUrl' => 'http://some.cdn.com/files/large-files' //
    効力を持たな
    い!
                ],
            ],
        ],
    ],
];
```

アセット・マッピング

時として、複数のアセット・バンドルで使われている 正しくない/互換でない アセット・ファイル・パスを「修正」したい場合があります。 例えば、バンドル A がバージョン 1.11.1 の `jquery.min.js` を使い、バンドル B がバージョン 2.1.1 の `jquery.js` を使っているような場合です。 それぞれのバンドルをカスタマイズすることで問題を修正することも出来ますが、それよりも簡単な方法は、アセット・マップ 機能を使って、正しくないアセットを望ましいアセットに割り付けることです。 そうするため

には、以下のように yii\web\AssetManager::$assetMap プロパティを構成します。

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' => '//ajax.googleapis.com/ajax/libs/jquery
    /2.1.1/jquery.min.js',
            ],
        ],
    ],
];
```

assetMap のキーは修正したいアセットの名前であり、値は望ましいアセットのパスです。 アセット・バンドルをビューに登録するとき、css と js の配列に含まれるすべてのアセット・ファイルの相対パスがこのマップと突き合わせて調べられます。 キーのどれかがアセット・ファイルのパス (利用できる場合は、yii\web\AssetBundle::$sourcePath が前置されます) の最後の部分と一致した場合は、対応する値によってアセットが置き換えられ、ビューに登録されます。 例えば、my/path/to/jquery.js というアセット・ファイルは jquery.js というキーにマッチします。

> 補足: 相対パスを使って指定されたアセットだけがアセット・マッピングの対象になります。 そして、置き換える側のアセットのパスは、絶対 URL であるか、yii\web\AssetManager::$basePath からの相対パスであるかの、どちらかでなければなりません。

アセット発行

既に述べたように、アセット・バンドルがウェブからアクセス出来ないディレクトリに配置されている場合は、 バンドルがビューに登録されるときに、アセットがウェブ・ディレクトリにコピーされます。 このプロセスは アセット発行 と呼ばれ、アセット・マネージャ によって自動的に実行されます。

デフォルトでは、アセットが発行されるディレクトリは @webroot/assets であり、@web/assets という URL に対応するものです。 この場所は、basePath と baseUrl のプロパティを構成してカスタマイズすることが出来ます。

ファイルをコピーすることでアセットを発行する代りに、OS とウェブ・サーバが許容するなら、シンボリック・リンクを使うことを考慮しても良いでしょう。 この機能は linkAssets を true にセットすることで有効にすることが出来ます。

```
return [
    // ...
    'components' => [
```

```
        'assetManager' => [
            'linkAssets' => true,
        ],
    ],
];
```

上記の構成によって、アセット・マネージャはアセット・バンドルを発行するときにソース・パスへのシンボリック・リンクを作成するようになります。 この方がファイルのコピーより速く、また、 発行されたアセットが常に最新であることを保証することも出来ます。

キャッシュの廃棄

運用モードで動作しているウェブ・アプリケーションでは、アセットなどの静的なリソースに対する HTTP キャッシュを有効にするのが通例です。 この慣行の難点は、アセットを修正して運用サーバに配備したときに、ユーザのクライアントが HTTP キャッシュのせいで古いバージョンを使い続けるおそれが常にあるという点です。 この難点を克服するために、キャッシュ廃棄機能を使うことが出来ます。 これはバージョン 2.0.3 で導入された機能で、`yii\web\AssetManager` を下記のように構成することで有効になります。

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'appendTimestamp' => true,
        ],
    ],
];
```

このようにすると、全ての発行されたアセットの URL の末尾に最終更新日時のタイムスタンプが追加されます。 例えば、`yii.js` に対する URL は `/assets/5515a87c/yii.js?v=1423448645"` のようなものになります。 パラメータ `v` が `yii.js` ファイルの最終更新日時のタイムスタンプを表しています。 これで、あなたがアセットを修正したときには、その URL も変更され、クライアントに最新バージョンのアセットを強制的に取得させることが出来ます。

3.11.4 よく使われるアセット・バンドル

コアの Yii コードは多くのアセット・バンドルを定義しています。 その中で、下記のバンドルはよく使われるものであり、あなたのアプリケーションやエクステンションのコードでも参照することが出来るものです。

- `yii\web\YiiAsset`: 主として `yii.js` ファイルをインクルードするためのバンドルです。 このファイルはモジュール化された JavaScript のコードを編成するメカニズムを実装しています。 また、`data-method` と `data-confirm` の属性に対する特別なサポートや、その他の

有用な機能を提供します。 yii.js に関する詳細な情報は クライアント・スクリプトのセクション にあります。

- yii\web\JqueryAsset: jQuery の bower パッケージから jquery.js ファイルをインクルードします。
- yii\bootstrap\BootstrapAsset: Twitter Bootstrap フレームワークから CSS ファイルをインクルードします。
- yii\bootstrap\BootstrapPluginAsset: Bootstrap JavaScript プラグインをサポートするために、 Twitter Bootstrap フレームワークから JavaScript ファイルをインクルードします。
- yii\jui\JuiAsset: jQuery UI ライブラリから CSS と JavaScript のファイルをインクルードします。

あなたのコードが、jQuery や jQuery UI または Bootstrap に依存する場合は、自分自身のバージョンを作るのではなく、これらの定義済みのアセット・バンドルを使用すべきです。 これらのバンドルのデフォルトの設定があなたの必要を満たさない時は、アセット・バンドルをカスタマイズする の項で説明したように、それをカスタマイズすることが出来ます。

3.11.5 アセット変換

直接に CSS および/または JavaScript のコードを書く代りに、何らかの拡張構文を使って書いたものを特別なツールを使って CSS/JavaScript に変換する、ということを開発者はしばしば行います。 例えば、CSS コードのためには、LESS[29] や SCSS[30] を使うことが出来ます。 また、JavaScript のためには、TypeScript[31] を使うことが出来ます。

拡張構文を使ったアセット・ファイルをアセット・バンドルの中の css と js のリストに挙げることが出来ます。例えば、

```
class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.less',
    ];
    public $js = [
        'js/site.ts',
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

[29] http://lesscss.org/
[30] http://sass-lang.com/
[31] http://www.typescriptlang.org/

このようなアセット・バンドルをビューに登録すると、アセット・マネージャ が自動的にプリ・プロセッサ・ツールを走らせて、 認識できた拡張構文のアセットを CSS/JavaScript に変換します。 最終的にビューがページをレンダリングするときには、ビューは元の拡張構文のアセットではなく、 変換後の CSS/JavaScript ファイルをページにインクルードします。

Yii はファイル名の拡張子を使って、アセットが使っている拡張構文を識別します。 デフォルトでは、下記の構文とファイル名拡張子を認識します。

- LESS[32]: `.less`
- SCSS[33]: `.scss`
- Stylus[34]: `.styl`
- CoffeeScript[35]: `.coffee`
- TypeScript[36]: `.ts`

Yii はインストールされたプリ・プロセッサ・ツールに頼ってアセットを変換します。 例えば、LESS[37] を使うためには、`lessc` プリ・プロセッサ・コマンドをインストールしなければなりません。

下記のように `yii\web\AssetManager::$converter` を構成することで、 プリ・プロセッサ・コマンドとサポートされる拡張構文をカスタマイズすることが出来ます。

```
return [
    'components' => [
        'assetManager' => [
            'converter' => [
                'class' => 'yii\web\AssetConverter',
                'commands' => [
                    'less' => ['css', 'lessc {from} {to} --no-color'],
                    'ts' => ['js', 'tsc --out {to} {from}'],
                ],
            ],
        ],
    ],
];
```

上記においては、サポートされる拡張構文が `yii\web\AssetConverter::$commands` プロパティによって定義されています。 配列のキーはファイルの拡張子 (先頭のドットは省く) であり、 配列の値は結果として作られるアセット・ファイルの拡張子とアセット変換を実行するためのコマンドです。 コマンドの中の `{from}` と `{to}` のトークンは、ソースのアセット・ファイルのパスとターゲットのアセット・ファイルのパスに置き換えられます。

[32] http://lesscss.org/
[33] http://sass-lang.com/
[34] http://learnboost.github.io/stylus/
[35] http://coffeescript.org/
[36] http://www.typescriptlang.org/
[37] http://lesscss.org/

> 情報: 上記で説明した方法の他にも、拡張構文のアセットを扱う方法はあります。 例えば、grunt[38] のようなビルド・ツールを使って、拡張構文のアセットをモニターし、 自動的に変換することが出来ます。 この場合は、元のファイルではなく、結果として作られる CSS/JavaScript ファイルをアセット・バンドルのリストに挙げなければなりません。

3.11.6 アセットを結合して圧縮する

ウェブ・ページは数多くの CSS および/または JavaScript ファイルをインクルードすることがあり得ます。 HTTP リクエストの数とこれらのファイルの全体としてのダウンロード・サイズを削減するためによく用いられる方法は、複数の CSS/JavaScript ファイルを結合して圧縮し、一つまたはごく少数のファイルにまとめることです。 そして、ウェブ・ページでは元のファイルをインクルードする代りに、圧縮されたファイルをインクルードする訳です。

> 情報: アセットの結合と圧縮は、通常はアプリケーションが本番モードにある場合に必要になります。 開発モードにおいては、たいていは元の CSS/JavaScript ファイルを使う方がデバッグのために好都合です。

次に、既存のアプリケーション・コードを修正する必要なしに、 アセット・ファイルを結合して圧縮する方法を紹介します。

1. アプリケーションの中で、結合して圧縮する予定のアセット・バンドルを全て探し出す。
2. これらのバンドルを一個か数個のグループにまとめる。どのバンドルも一つのグループにしか属することが出来ないことに注意。
3. 各グループの CSS ファイルを一つのファイルに結合/圧縮する。JavaScript ファイルに対しても同様にこれを行う。
4. 各グループに対して新しいアセット・バンドルを定義する。
   - `css` と `js` のプロパティに、 それぞれ、結合された CSS ファイルと JavaScript ファイルをセットする。
   - 各グループに属する元のアセット・バンドルをカスタマイズして、`css` と `js` のプロパティを空にし、`depends` プロパティにグループのために作られた新しいバンドルを指定する。

この方法を使うと、ビューでアセット・バンドルを登録したときに、 元のバンドルが属するグループのための新しいアセット・バンドルが自動的に登録されるようになります。 そして、結果として、結合/圧縮されたアセット・ファイルが、元のファイルの代りに、ページにインクルードされます。

[38] http://gruntjs.com/

一例

上記の方法をさらに説明するために一つの例を挙げましょう。

あなたのアプリケーションが二つのページ、X と Y を持つと仮定します。ページ X はアセット・バンドル A、B、C を使用し、ページ Y はアセット・バンドル B、C、D を使用します。

これらのアセット・バンドルを分割する方法は二つあります。一つは単一のグループを使用して全てのアセット・バンドルを含める方法です。 もう一つは、A をグループ X に入れ、D をグループ Y に入れ、そして、B と C をグループ S に入れる方法です。 どちらが良いでしょう? 場合によります。 最初の方法の利点は、二つのページが同一の結合された CSS と JavaScript のファイルを共有するため、HTTP キャッシュの効果が高くなることです。 その一方で、単一のグループが全てのバンドルを含むために、結合された CSS と JavaScript のファイルはより大きくなり、従って最初のファイル転送時間はより長くなります。 この例では話を簡単にするために、最初の方法、すなわち、全てのバンドルを含む単一のグループを使用することにします。

> 情報: アセット・バンドルをグループに分けることは些細な仕事ではありません。通常、そのためには、いろいろなページのさまざまなアセットの現実世界での転送量を分析することが必要になります。 とりあえず、最初は、簡単にするために、単一のグループから始めて良いでしょう。

既存のツール (例えば Closure Compiler[39] や YUI Compressor[40]) を使って、 全てのバンドルにある CSS と JavaScript のファイルを結合して圧縮します。 ファイルは、バンドル間の依存関係を満たす順序に従って結合しなければならないことに注意してください。 例えば、バンドル A が B に依存し、B が C と D の両方に依存している場合は、アセット・ファイルの結合順は、 最初に C と D、次に B、そして最後に A としなければなりません。

結合と圧縮が完了すると、一つの CSS ファイルと一つの JavaScript ファイルが得られます。 それらは、`all-xyz.css` および `all-xyz.js` と命名されたとしましょう。 ここで `xyz` は、ファイル名をユニークにして HTTP キャッシュの問題を避けるために使用されるタイムスタンプまたはハッシュを表します。

いよいよ最終ステップです。 アプリケーションの構成情報の中で、アセット・マネージャ を次のように構成します。

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => [
                'all' => [
```

[39] https://developers.google.com/closure/compiler/
[40] https://github.com/yui/yuicompressor/

```
                    'class' => 'yii\web\AssetBundle',
                    'basePath' => '@webroot/assets',
                    'baseUrl' => '@web/assets',
                    'css' => ['all-xyz.css'],
                    'js' => ['all-xyz.js'],
                ],
                'A' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'B' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'C' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'D' => ['css' => [], 'js' => [], 'depends' => ['all']],
            ],
        ],
    ],
];
```

アセット・バンドルをカスタマイズする の項で説明したように、上記の構成によって元のバンドルは全てデフォルトの振る舞いを変更されます。 具体的にいえば、バンドル A、B、C、D は、もはやアセット・ファイルを一つも持っていません。 この4つは、それぞれ、結合された `all-xyz.css` と `all-xyz.js` ファイルを持つ `all` バンドルに依存するようになりました。 結果として、ページ X では、バンドル A、B、C から元のソース・ファイルをインクルードする代りに、これら二つの結合されたファイルだけがインクルードされます。 同じことはページ Y でも起ります。

最後にもう一つ、上記の方法をさらにスムーズに運用するためのトリックがあります。 アプリケーションの構成情報ファイルを直接修正する代りに、バンドルのカスタマイズ用の配列を独立したファイルに置いて、条件によってそのファイルをアプリケーションの構成情報にインクルードすることが出来ます。 例えば、

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => require __DIR__ . '/' . (YII_ENV_PROD ? 'assets-
    prod.php' : 'assets-dev.php'),
        ],
    ],
];
```

つまり、アセット・バンドルの構成情報配列は、本番モードのものは `assets-prod.php` に保存し、 開発モードのものは `assets-dev.php` に保存するという訳です。

> 補足: このアセット結合のメカニズムは、登録されるアセット・バンドルのプロパティをオーバーライドできるという `yii\web\AssetManager::$bundles` の機能に基づいています。 しかし、既に上で述べたように、この機能は、`yii\web\AssetBundle::init()` メソッドの中やバンドルが登録された後で実行されるアセット・バンドルの修正をカバーしていま

せん。 そのような動的なバンドルの使用は、アセット結合をする際には避けなければなりません。

asset コマンドを使う

Yii は、たった今説明した方法を自動化するための asset という名前のコンソール・コマンドを提供しています。

このコマンドを使うためには、最初に構成情報ファイルを作成して、どのアセット・バンドルが結合されるか、 そして、それらがどのようにグループ化されるかを記述しなければなりません。 asset/template サブ・コマンドを使って最初にテンプレートを生成し、それをあなたの要求に合うように修正することが出来ます。

```
yii asset/template assets.php
```

上記のコマンドは、カレント・ディレクトリに assets.php というファイルを生成します。このファイルの内容は以下のようなものです。

```
<?php
/**
 * "yii asset" コンソール・コマンドのための構成情報ファイル
 * コンソール環境では、'@webroot' や '@web' のように、存在しないパス・エイリアスがあり得ることに注意してください。
 * これらの欠落したパス・エイリアスは手作業で定義してください。
 */
return [
    // JavaScript ファイルの圧縮のためのコマンドコールバックを調整。/
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file {to}',
    // CSS ファイルの圧縮のためのコマンドコールバックを調整。/
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o {to}',
    // 圧縮後にアセットのソースを削除するかどうか。
    'deleteSource' => false,
    // 圧縮するアセット・バンドルのリスト。
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // 圧縮出力用のアセット・バンドル。
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // アセット・マネージャの構成情報
    'assetManager' => [
    ],
```

```
];
```

このファイルを修正して、どのバンドルを結合するつもりであるかを bundles オプションで指定しなければなりません。 targets オプションでは、バンドルがどのようにグループに分割されるかを指定しなければなりません。 既に述べたように、一つまたは複数のグループを定義することが出来ます。

> 補足: パス・エイリアス @webroot および @web はコンソール・アプリケーションでは利用できませんので、 これらは構成情報の中で明示的に定義しなければなりません。

JavaScript ファイルは結合され、圧縮されて js/all-{hash}.js に保存されます。ここで {hash} は、 結果として作られたファイルのハッシュで置き換えられるものです。

jsCompressor と cssCompressor のオプションは、JavaScript と CSS の結合/圧縮を実行するコンソール・コマンドまたは PHP コールバックを指定するものです。 デフォルトでは、Yii は JavaScript ファイルの結合に Closure Compiler[41] を使い、 CSS ファイルの結合に YUI Compressor[42] を使用します。 あなたの好みのツールを使うためには、手作業でツールをインストールしたり、オプションを修正したりしなければなりません。

この構成情報ファイルを使い、asset コマンドを走らせて、アセット・ファイルを結合して圧縮し、 同時に、新しいアセット・バンドルの構成情報ファイル assets-prod.php を生成することが出来ます。

```
yii asset assets.php config/assets-prod.php
```

直前の項で説明したように、 この生成された構成情報ファイルをアプリケーションの構成情報にインクルードすることが出来ます。

> 補足: アプリケーションのアセット・バンドルを yii\web\AssetManager::$bundles または yii\web\AssetManager::$assetMap によってカスタマイズしており、 そのカスタマイズを圧縮のソース・ファイルにも適用したい場合は、それらのオプションを asset コマンドの構成ファイルの assetManager のセクションに含めなければいけません。

> 補足: 圧縮のソースを指定する場合は、パラメータが動的に (例えば init() メソッドの中や登録後に) 修正されるアセット・バンドルを避けなければなりません。 なぜなら、パラメータの動的な修正は、圧縮後は正しく働かない可能性があるからです。

> 情報: asset コマンドを使うことは、アセットの結合・圧縮のプロセスを自動化する唯一の選択肢ではありません。 優秀なタスク実行ツールである grunt[43] を使っても、同じ目的を達す

[41] https://developers.google.com/closure/compiler/
[42] https://github.com/yui/yuicompressor/
[43] http://gruntjs.com/

ることが出来ます。

アセット・バンドルをグループ化する

直前の項において、全てのアセット・バンドルを一つに結合して、アプリケーションで参照されるアセット・ファイルに対する HTTP リクエストを最小化する方法を説明しました。 現実には、それが常に望ましいわけではありません。 例えば、あなたのアプリケーションがフロントエンドとバックエンドを持っており、 それぞれが異なる一群の JavaScript と CSS ファイルを使う場合を想像してください。 この場合、両方の側の全てのアセット・バンドルを一つに結合するのは合理的ではありません。何故なら、フロントエンドのためのアセット・バンドルはバックエンドでは使用されませんから、フロントエンドのページがリクエストされているときにバックエンドのアセットを送信するのはネットワーク帯域の浪費です。

上記の問題を解決するために、アセット・バンドルをグループ化して、グループごとにアセット・バンドルを結合することが出来ます。 下記の構成情報は、アセット・バンドルをグループ化する方法を示すものです。

```
return [
    ...
    // 出力されるバンドルをグループ化する
    'targets' => [
        'allShared' => [
            'js' => 'js/all-shared-{hash}.js',
            'css' => 'css/all-shared-{hash}.css',
            'depends' => [
                // バックエンドとフロントエンドで共有される全てのアセットを含める
                'yii\web\YiiAsset',
                'app\assets\SharedAsset',
            ],
        ],
        'allBackEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [
                // バックエンドだけのアセットを含める
                'app\assets\AdminAsset'
            ],
        ],
        'allFrontEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [], // 残りの全てのアセットを含める
        ],
    ],
    ...
];
```

ご覧のように、アセット・バンドルは三つのグループ、すなわち、allShared 、allBackEnd および allFrontEnd に分けられています。 そして、それぞれが適切な一群のアセット・バンドルに依存しています。例えば、allBackEnd は app\assets\AdminAsset に依存しています。 この構成情報によって asset コマンドを実行すると、上記の定義に従ってアセット・バンドルが結合されます。

> 情報: ターゲット・バンドルのうちの一つについて depends の構成を空のままにしておくことが出来ます。 そのようにすると、他のターゲット・バンドルが依存しないために残された全てのアセット・バンドルが、このターゲット・バンドルに含まれるようになります。

3.12 エクステンション

エクステンションは、Yii のアプリケーションで使われることに限定して設計され、そのまますぐに使える機能を提供する再配布可能なソフトウェア・パッケージです。 例えば、yiisoft/yii2-debug[44] エクステンションは、あなたのアプリケーションにおいて、全てのページの末尾に便利なデバッグ・ツールバーを追加して、 ページが生成される過程をより容易に把握できるように手助けしてくれます。 エクステンションを使うと、あなたの開発プロセスを加速することが出来ます。 また、あなたのコードをエクステンションとしてパッケージ化すると、あなたの優れた仕事を他の人たちと共有することが出来ます。

> 情報: 「エクステンション」という用語は Yii に限定されたソフトウェア・パッケージを指すものとして使用します。 Yii がなくても使用できる汎用のソフトウェア・パッケージを指すためには、「パッケージ」または「ライブラリ」という用語を使うことにします。

3.12.1 エクステンションを使う

エクステンションを使うためには、先ずはそれをインストールする必要があります。 ほとんどのエクステンションは Composer[45] のパッケージとして配布されていて、次の二つの簡単なステップをふめばインストールすることが出来ます。

1. アプリケーションの composer.json ファイルを修正して、どのエクステンション (Composer パッケージ) をインストールしたいかを指定する。

[44] https://github.com/yiisoft/yii2-debug
[45] https://getcomposer.org/

2. `composer install` コマンドを走らせて指定したエクステンションをインストールする。

Composer[46] を持っていない場合は、それをインストールする必要があることに注意してください。

デフォルトでは、Composer はオープン・ソース Composer パッケージの最大のレポジトリである Packagist[47] に登録されたパッケージをインストールします。 エクステンションは Packagist で探すことが出来ます。また、自分自身のレポジトリを作成[48] して、それを使うように Composer を構成することも出来ます。 これは、あなたがプライベートなエクステンションを開発していて、それを自分のプロジェクト間でのみ共有したい場合に役に立つ方法です。

Composer によってインストールされるエクステンションは `BasePath/vendor` ディレクトリに保存されます。 ここで `BasePath` は、アプリケーションの ベース・パス を指します。 Composer は依存関係を管理するものですから、あるパッケージをインストールするときには、それが依存する全てのパッケージも同時にインストールします。

例えば、`yiisoft/yii2-imagine` エクステンションをインストールするためには、あなたの `composer.json` を次のように修正します。

```
{
    // ...

    "require": {
        // ... 他の依存パッケージ

        "yiisoft/yii2-imagine": "*"
    }
}
```

インストール完了後には、`BasePath/vendor` の下に `yiisoft/yii2-imagine` ディレクトリが作られている筈です。 それと同時に、`imagine/imagine` という別のディレクトリも作られて、依存するパッケージがそこにインストールされている筈です。

> 情報: `yiisoft/yii2-imagine` は Yii 開発チームによって開発され保守されるコア・エクステンションの一つです。 全てのコア・エクステンションは Packagist[49] でホストされ、`yiisoft/yii2-xyz` のように名付けられます。 ここで `xyz` はエクステンションによってさまざまに変ります。

これであなたはインストールされたエクステンションをあなたのアプリケーションの一部であるかのように使うことが出来ます。 次の

[46] https://getcomposer.org/
[47] https://packagist.org/
[48] https://getcomposer.org/doc/05-repositories.md#repository
[49] https://packagist.org/

例は、yiisoft/yii2-imagine エクステンションによって提供される yii\imagine\Image クラスをどのようにして使うことが出来るかを示すものです。

```
use Yii;
use yii\imagine\Image;

// サムネール画像を生成する
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
    ->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>
    50]);
```

> 情報: エクステンションのクラスは Yii クラス・オートローダによってオートロードされます。

エクステンションを手作業でインストールする

あまり無いことですが、いくつかまたは全てのエクステンションを Composer に頼らずに手作業でインストールしたい場合があるかもしれません。 そうするためには、次のようにしなければなりません。

1. エクステンションのアーカイブ・ファイルをダウンロードして、vendor ディレクトリに解凍する。
2. もし有れば、エクステンションによって提供されているクラス・オートローダをインストールする。
3. 指示に従って、依存するエクステンションを全てダウンロードしインストールする。

エクステンションがクラス・オートローダを持っていなくても、PSR-4 標準[50] に従っている場合は、Yii によって提供されているクラス・オートローダを使ってエクステンションのクラスをオートロードすることが出来ます。 必要なことは、エクステンションのルート・ディレクトリのための ルート・エイリアス を宣言することだけです。 例えば、エクステンションを vendor/mycompany/myext というディレクトリにインストールしたと仮定します。 そして、エクステンションのクラスは myext 名前空間の下にあるとします。 その場合、アプリケーションの構成情報に下記のコードを含めます。

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

[50] http://www.php-fig.org/psr/psr-4/

3.12.2 エクステンションを作成する

あなたの優れたコードを他の人々と共有する必要があると感じたときは、エクステンションを作成することを考慮するのが良いでしょう。 エクステンションは、ヘルパ・クラス、ウィジェット、モジュールなど、どのようなコードでも含むことが出来ます。

エクステンションは、Composer パッケージ[51] の形式で作成することが推奨されます。 そうすれば、直前の項で説明したように、いっそう容易に他のユーザによってインストールされ、使用されることが出来ます。

以下は、エクステンションを Composer のパッケージとして作成するために踏む基本的なステップです。

1. エクステンションのためのプロジェクトを作成して、github.com[52] などの VCS レポジトリ上でホストします。 エクステンションに関する開発と保守の作業はこのレポジトリ上でしなければなりません。

2. プロジェクトのルート・ディレクトリに、Composer によって要求される `composer.json` という名前のファイルを作成します。 詳細については、次の項を参照してください。

3. エクステンションを Packagist[53] などの Composer レポジトリに登録します。 そうすると、他のユーザがエクステンションを見つけて Composer を使ってインストールすることが出来るようになります。

`composer.json`

全ての Composer パッケージは、ルート・ディレクトリに `composer.json` というファイルを持たなければなりません。このファイルはパッケージに関するメタデータを含むものです。 このファイルに関する完全な仕様は Composer Manual[54] に記載されています。 次の例は、`yiisoft/yii2-imagine` エクステンションのための `composer.json` ファイルを示すものです。

```
{
    // パッケージ名
    "name": "yiisoft/yii2-imagine",

    // パッケージタイプ
    "type": "yii2-extension",
```

[51] https://getcomposer.org/
[52] https://github.com
[53] https://packagist.org/
[54] https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup

```
    "description": "The Imagine integration for the Yii framework",
    "keywords": ["yii2", "imagine", "image", "helper"],
    "license": "BSD-3-Clause",
    "support": {
        "issues": "https://github.com/yiisoft/yii2/issues?labels=ext%3
    Aimagine",
        "forum": "http://www.yiiframework.com/forum/",
        "wiki": "http://www.yiiframework.com/wiki/",
        "irc": "irc://irc.freenode.net/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "authors": [
        {
            "name": "Antonio Ramirez",
            "email": "amigo.cobos@gmail.com"
        }
    ],

    // 依存パッケージ
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // クラスのオートロードの仕様
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

パッケージ名　全ての Composer パッケージは、他の全てパッケージと異なる一意に特定できる名前を持たなければなりません。 パッケージ名の形式は `vendorName/projectName` です。 例えば、`yiisoft/yii2-imagine` というパッケージ名の中では、ベンダー名とプロジェクト名は、それぞれ、`yiisoft` と `yii2-imagine` です。

ベンダー名として `yiisoft` を使ってはいけません。これは Yii のコア・コードに使うために予約されています。

プロジェクト名には、Yii 2 エクステンションを表す `yii2-` を前置することを推奨します。例えば、`myname/yii2-mywidget` です。 このようにすると、ユーザはパッケージが Yii 2 エクステンションであることをより容易に知ることが出来ます。

パッケージ・タイプ　パッケージがインストールされたときに Yii のエクステンションとして認識されるように、エクステンションのパッケージ・タイプを `yii2-extension` と指定することは重要なことです。

ユーザが `composer install` を走らせてエクステンションをインストールすると、 `vendor/yiisoft/extensions.php` というファイルが自動的に更新

されて、新しいエクステンションに関する情報を含むようになります。Yii のアプリケーションは、このファイルによって、どんなエクステンションがインストールされているかを知ることが出来ます (その情報には、`yii\base\Application::$extensions` を通じてアクセスすることが出来ます)。

依存パッケージ あなたのエクステンションは Yii に依存します (当然ですね)。ですから、`composer.json` の `require` エントリのリストにそれ (`yiisoft/yii2`) を挙げなければなりません。 あなたのエクステンションがその他のエクステンションやサード・パーティのライブラリに依存する場合は、それらもリストに挙げなければなりません。 それぞれの依存パッケージについて、適切なバージョン制約 (例えば `1.*` や `@stable`) を指定することも忘れてはなりません。 あなたのエクステンションを安定バージョンとしてリリースする場合は、安定した依存パッケージを使ってください。

たいていの JavaScript/CSS パッケージは、Composer ではなく、Bower[55] および/または NPM[56] を使って管理されています。 Yii は Composer アセット・プラグイン[57] を使って、この種のパッケージを Composer によって管理することを可能にしています。 あなたのエクステンションが Bower パッケージに依存している場合でも、次のように、 `composer.json` に依存パッケージをリストアップすることが簡単に出来ます。

```
{
    // 依存パッケージ
    "require": {
        "bower-asset/jquery": ">=1.11.*"
    }
}
```

上記のコードは、エクステンションが `jquery` Bower パッケージに依存することを述べています。 一般に、`composer.json` の中で Bower パッケージを指すためには `bower-asset/PackageName` を使うことが出来ます。 そして、NPM パッケージを指すためには `npm-asset/PackageName` を使うことが出来ます。 Composer が Bower または NPM のパッケージをインストールする場合は、デフォルトでは、それぞれ、`@vendor/bower/PackageName` および `@vendor/npm/Packages` というディレクトリの下にパッケージの内容がインストールされます。 この二つのディレクトリは、`@bower/PackageName` および `@npm/PackageName` という短いエイリアスを使って参照することも可能です。

アセット管理に関する詳細については、アセット のセクションを参照してください。

[55] http://bower.io/
[56] https://www.npmjs.org/
[57] https://github.com/francoispluchino/composer-asset-plugin

クラスのオートロード　エクステンションのクラスが Yii のクラス・オートローダまたは Composer のクラス・オートローダによってオートロードされるように、 下記に示すように、`composer.json` ファイルの `autoload` エントリを指定しなければなりません。

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

一つまたは複数のルート名前空間と、それに対応するファイル・パスをリストに挙げることが出来ます。

エクステンションがアプリケーションにインストールされると、Yii は列挙されたルート名前空間の一つ一つに対して、 名前空間に対応するディレクトリを指す エイリアス を作成します。 例えば、上記の `autoload` の宣言は、`@yii/imagine` という名前のエイリアスに対応することになります。

推奨されるプラクティス

エクステンションは他の人々によって使われることを意図したものですから、多くの場合、追加の開発努力が必要になります。 以下に、高品質のエクステンションを作成するときによく用いられ、また推奨されるプラクティスのいくつかを紹介します。

名前空間　名前の衝突を避けて、エクステンションの中のクラスをオートロード可能にするために、名前空間を使うべきであり、 エクステンションの中のクラスには PSR-4 標準[58] または PSR-0 標準[59] に従った名前を付けるべきです。

あなたのクラスの名前空間は `vendorName\extensionName` で始まるべきです。 ここで `extensionName` は、`yii2-` という接頭辞を含むべきでないことを除けば、パッケージ名におけるプロジェクト名と同じものです。 例えば、`yiisoft/yii2-imagine` エクステンションでは、`yii\imagine` をエクステンションのクラスの名前空間として使っています。

`yii`、`yii2` または `yiisoft` をベンダー名として使ってはいけません。これらの名前は、Yii のコア・コードに使うために予約されています。

ブートストラップ・クラス　場合によっては、アプリケーションが ブートストラップ の段階にある間に、エクステンションに何らかのコードを

[58] http://www.php-fig.org/psr/psr-4/
[59] http://www.php-fig.org/psr/psr-0/

実行させたい場合があるでしょう。 例えば、エクステンションをアプリケーションの beginRequest イベントに反応させて、何らかの環境設定を調整したいことがあります。 エクステンションのユーザに対して、エクステンションの中にあるイベント・ハンドラを beginRequest イベントに明示的にアタッチするように指示することも出来ますが、より良い方法は、それを自動的に行うことです。

この目的を達するためには、yii\base\BootstrapInterface を実装する、いわゆる ブートストラップ・クラス を作成します。 例えば、

```
namespace myname\mywidget;

use yii\base\BootstrapInterface;
use yii\base\Application;

class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // ここで何かをする
        });
    }
}
```

そして、次のように、このクラスを composer.json ファイルのリストに挙げます。

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

このエクステンションがアプリケーションにインストールされると、すべてのリクエストのブートストラップの過程において、 毎回、Yii が自動的にブートストラップ・クラスのインスタンスを作成し、 その bootstrap() メソッドを呼びます。

データベースを扱う　あなたのエクステンションはデータベースにアクセスする必要があるかも知れません。エクステンションを使うアプリケーションが常に Yii::$db を DB 接続として使用すると仮定してはいけません。 その代りに、DB アクセスを必要とするクラスのために db プロパティを宣言すべきです。 このプロパティによって、エクステンションのユーザは、エクステンションにどの DB 接続を使わせるかをカスタマイズすることが出来るようになります。 その一例として、yii\caching\DbCache クラスを参照して、それがどのように db プロパティを宣言して使っているかを見ることが出来ます。

あなたのエクステンションが特定の DB テーブルを作成したり、DB スキーマを変更したりする必要がある場合は、次のようにするべきです。

- DB スキーマを操作するために、平文の SQL ファイルを使うのではなく、マイグレーション を提供する。
- マイグレーションがさまざまな DBMS に適用可能なものになるように試みる。
- マイグレーションの中では アクティブ・レコード の使用を避ける。

アセットを使う　あなたのエクステンションがウィジェットかモジュールである場合は、動作するために何らかの アセット が必要である可能性が高いでしょう。 例えば、モジュールは、画像、JavaScript、そして CSS を含むページを表示することがあるでしょう。 アプリケーションにインストールされるときに、エクステンションの全てのファイルは同じディレクトリの下に配置されますが、そのディレクトリはウェブからはアクセス出来ないものです。 そのため、次のどちらかの方法を使って、アセット・ファイルをウェブから直接アクセス出来るようにしなければなりません。

- アセット・ファイルをウェブからアクセス出来る特定のフォルダに手作業でコピーするように、エクステンションのユーザに要求する。
- アセット・バンドル を宣言し、アセット発行メカニズムに頼って、アセット・バンドルにリストされているファイルをウェブからアクセス出来るフォルダに自動的にコピーする。

あなたのエクステンションが他の人々にとって一層使いやすいものになるように、第二の方法をとることを推奨します。 アセットの取り扱い一般に関する詳細は アセット のセクションを参照してください。

国際化と地域化　あなたのエクステンションは、さまざまな言語をサポートするアプリケーションによって使われるかもしれません。 従って、あなたのエクステンションがエンド・ユーザにコンテントを表示するものである場合は、それを 国際化 するように努めるべきです。具体的には、

- エクステンションがエンド・ユーザに向けたメッセージを表示する場合は、翻訳することが出来るようにメッセージを `Yii::t()` で囲むべきです。 開発者に向けられたメッセージ (内部的な例外のメッセージなど) は翻訳される必要はありません。
- エクステンションが数値や日付などを表示する場合は、 `yii\i18n\Formatter` を適切な書式化の規則とともに使って書式設定すべきです。

詳細については、国際化 のセクションを参照してください。

テスト　あなたは、あなたのエクステンションが他の人々に問題をもたらすことなく完璧に動作することを望むでしょう。 この目的を達するためには、あなたのエクステンションを公開する前にテストすべきです。

手作業のテストに頼るのではなく、あなたのエクステンションのコードをカバーするさまざまなテスト・ケースを作成することが推奨されます。 あなたのエクステンションの新しいバージョンを公開する前には、毎回、それらのテスト・ケースを走らせるだけで、全てが良い状態にあることを確認することが出来ます。 Yii はテストのサポートを提供しており、それよって、単体テスト、機能テスト、受入テストを書くことが一層簡単に出来るようになっています。 詳細については、テスト のセクションを参照してください。

バージョン管理　エクステンションのリリースごとにバージョン番号 (例えば `1.0.1`) を付けるべきです。 どのようなバージョン番号を付けるべきかを決定するときは、セマンティック・バージョニング[60] のプラクティスに従うことを推奨します。

リリース(公開)　他の人々にあなたのエクステンションを知ってもらうためには、それをリリース(公開)する必要があります。

エクステンションをリリースするのが初めての場合は、Packagist[61] などの Composer レポジトリにエクステンションを登録するべきです。その後は、あなたがしなければならない仕事は、エクステンションの VCS レポジトリでリリース・タグ (例えば `v1.0.1`) を作成することと、Composer レポジトリに新しいリリースについて通知するだけのことになります。 そうすれば、人々が新しいリリースを見出すことが出来るようになり、Composer レポジトリを通じてエクステンションをインストールしたりアップデートしたりするようになります。

エクステンションのリリースには、コード・ファイル以外に、人々があなたのエクステンションについて知ったり、 エクステンションを使ったりするのを助けるために、下記のものを含めることを考慮すべきです。

- パッケージのルート・ディレクトリに readme ファイル: あなたのエクステンションが何をするものか、 そして、どのようにインストールして使うものかを説明するものです。 Markdown[62] 形式で書いて、`readme.md` という名前にすることを推奨します。
- パッケージのルート・ディレクトリに changelog ファイル: それぞれのリリースで何が変ったかを一覧表示するものです。 このファイルは Markdown 形式で書いて `changelog.md` と名付けることが出来ます。

[60] http://semver.org
[61] https://packagist.org/
[62] http://daringfireball.net/projects/markdown/

- パッケージのルート・ディレクトリに upgrade ファイル: エクステンションの古いリリースからのアップグレード方法について説明するものです。 このファイルは Markdown 形式で書いて `upgrade.md` と名付けることが出来ます。
- チュートリアル、デモ、スクリーン・ショットなど: あなたのエクステンションが readme ファイルでは十分にカバーできないほど多くの機能を提供するものである場合は、 これらが必要になります。
- API ドキュメント: あなたのコードは、他の人々が読んで理解することがより一層容易に出来るように、十分な解説を含むべきです。 BaseObject のクラス・ファイル[63] を参照すると、 コードに解説を加える方法を学ぶことが出来ます。

  情報: コードのコメントを Markdown 形式で書くことが出来ます。 `yiisoft/yii2-apidoc` エクステンションが、コードのコメントに基づいて綺麗な API ドキュメントを生成するツールを提供しています。

  情報: これは要求ではありませんが、あなたのエクステンションも一定のコーディング・スタイルを守るのが良いと思います。 コア・フレームワーク・コード・スタイル[64] を参照してください。

3.12.3 コア・エクステンション

Yii は下記のコア・エクステンション (または “公式エクステンション”[65]) を提供しています。 これらは Yii 開発チームによって開発され保守されているものです。 全て Packagist[66] に登録され、エクステンションを使うの項で説明したように、簡単にインストールすることが出来ます。

- yiisoft/yii2-apidoc[67]: 拡張可能で高性能な API ドキュメント生成機能を提供します。 コア・フレームワークの API ドキュメントを生成するためにも使われています。
- yiisoft/yii2-authclient[68]: Facebook OAuth2 クライアント、GitHub OAuth2 クライアントなど、よく使われる一連の auth クライアントを提供します。
- yiisoft/yii2-bootstrap[69]: Bootstrap[70] のコンポーネントとプラグインをカプセル化した一連のウィジェットを提供します。

---

[63] https://github.com/yiisoft/yii2/blob/master/framework/base/BaseObject.php
[64] https://github.com/yiisoft/yii2/wiki/Core-framework-code-style
[65] https://www.yiiframework.com/extensions/official
[66] https://packagist.org/
[67] https://github.com/yiisoft/yii2-apidoc
[68] https://github.com/yiisoft/yii2-authclient
[69] https://github.com/yiisoft/yii2-bootstrap
[70] http://getbootstrap.com/

- yiisoft/yii2-codeception[71] (非推奨): Codeception[72] に基づくテストのサポートを提供します。
- yiisoft/yii2-debug[73]: Yii アプリケーションのデバッグのサポートを提供します。 このエクステンションが使われると、全てのページの末尾にデバッガ・ツールバーが表示されます。 このエクステンションは、より詳細なデバッグ情報を表示する一連のスタンドアロン・ページも提供します。
- yiisoft/yii2-elasticsearch[74]: Elasticsearch[75] の使用に対するサポートを提供します。 基本的なクエリ/サーチのサポートを含むだけでなく、Elasticsearch にアクティブ・レコードを保存することを可能にする アクティブ・レコード パターンをも実装しています。
- yiisoft/yii2-faker[76]: ダミー・データを作る Faker[77] を使うためのサポートを提供します。
- yiisoft/yii2-gii[78]: 拡張性が非常に高いウェブ・ベースのコード・ジェネレータを提供します。 これを使って、モデル、フォーム、モジュール、CRUD などを迅速に生成することが出来ます。
- yiisoft/yii2-httpclient[79]: HTTP クライアントを提供します。
- yiisoft/yii2-imagine[80]: Imagine[81] に基づいて、使われることの多い画像操作機能を提供します。
- yiisoft/yii2-jui[82]: JQuery UI[83] のインタラクションとウィジェットをカプセル化した一連のウィジェットを提供します。
- yiisoft/yii2-mongodb[84]: MongoDB[85] の使用に対するサポートを提供します。 基本的なクエリ、アクティブ・レコード、マイグレーション、キャッシュ、コード生成などの機能を含みます。
- yiisoft/yii2-queue[86]: キューによるタスクの非同期実行のサポートを提供します。 データベース、Redis、RabbitMQ、AMQP、Beanstalk および Gearman によるキューをサポートしています。
- yiisoft/yii2-redis[87]: redis[88] の使用に対するサポートを提供します。

[71] https://github.com/yiisoft/yii2-codeception
[72] http://codeception.com/
[73] https://github.com/yiisoft/yii2-debug
[74] https://github.com/yiisoft/yii2-elasticsearch
[75] http://www.elasticsearch.org/
[76] https://github.com/yiisoft/yii2-faker
[77] https://github.com/fzaninotto/Faker
[78] https://github.com/yiisoft/yii2-gii
[79] https://github.com/yiisoft/yii2-httpclient
[80] https://github.com/yiisoft/yii2-imagine
[81] http://imagine.readthedocs.org/
[82] https://github.com/yiisoft/yii2-jui
[83] http://jqueryui.com/
[84] https://github.com/yiisoft/yii2-mongodb
[85] http://www.mongodb.org/
[86] https://www.yiiframework.com/extension/yiisoft/yii2-queue
[87] https://github.com/yiisoft/yii2-redis
[88] http://redis.io/

基本的なクエリ、アクティブ・レコード、キャッシュなどの機能を含みます。
- yiisoft/yii2-shell[89]: psysh[90] に基づくイタラクティブなシェルを提供します。
- yiisoft/yii2-smarty[91]: Smarty[92] に基づいたテンプレート・エンジンを提供します。
- yiisoft/yii2-sphinx[93]: Sphinx[94] の使用に対するサポートを提供します。 基本的なクエリ、アクティブ・レコード、コード生成などの機能を含みます。
- yiisoft/yii2-swiftmailer[95]: swiftmailer[96] に基づいたメール送信機能を提供します。
- yiisoft/yii2-twig[97]: Twig[98] に基づいたテンプレート・エンジンを提供します。

下記の公式エクステンションは Yii 2.1 以上のためのものです。 これらは、Yii 2.0 ではコア・フレームワークに含まれていますので、インストールする必要はありません。.

- yiisoft/yii2-captcha[99]: CAPTCHA を提供します。
- yiisoft/yii2-jquery[100]: jQuery[101] のサポートを提供します。
- yiisoft/yii2-maskedinput[102]: jQuery Input Mask plugin[103] に基づいて、マスクト・インプットを提供します。
- yiisoft/yii2-mssql[104]: MSSQL[105] を使うためのサポートを提供します。
- yiisoft/yii2-oracle[106]: Oracle[107] を使うためのサポートを提供します。
- yiisoft/yii2-rest[108]: REST API に対するサポートを提供します。

[89] https://www.yiiframework.com/extension/yiisoft/yii2-shell
[90] http://psysh.org/
[91] https://github.com/yiisoft/yii2-smarty
[92] http://www.smarty.net/
[93] https://github.com/yiisoft/yii2-sphinx
[94] http://sphinxsearch.com
[95] https://github.com/yiisoft/yii2-swiftmailer
[96] http://swiftmailer.org/
[97] https://github.com/yiisoft/yii2-twig
[98] http://twig.sensiolabs.org/
[99] https://www.yiiframework.com/extension/yiisoft/yii2-captcha
[100] https://www.yiiframework.com/extension/yiisoft/yii2-jquery
[101] https://jquery.com/
[102] https://www.yiiframework.com/extension/yiisoft/yii2-maskedinput
[103] http://robinherbots.github.io/Inputmask/
[104] https://www.yiiframework.com/extension/yiisoft/yii2-mssql
[105] https://www.microsoft.com/sql-server/
[106] https://www.yiiframework.com/extension/yiisoft/yii2-oracle
[107] https://www.oracle.com/
[108] https://www.yiiframework.com/extension/yiisoft/yii2-rest

Chapter 4

リクエストの処理

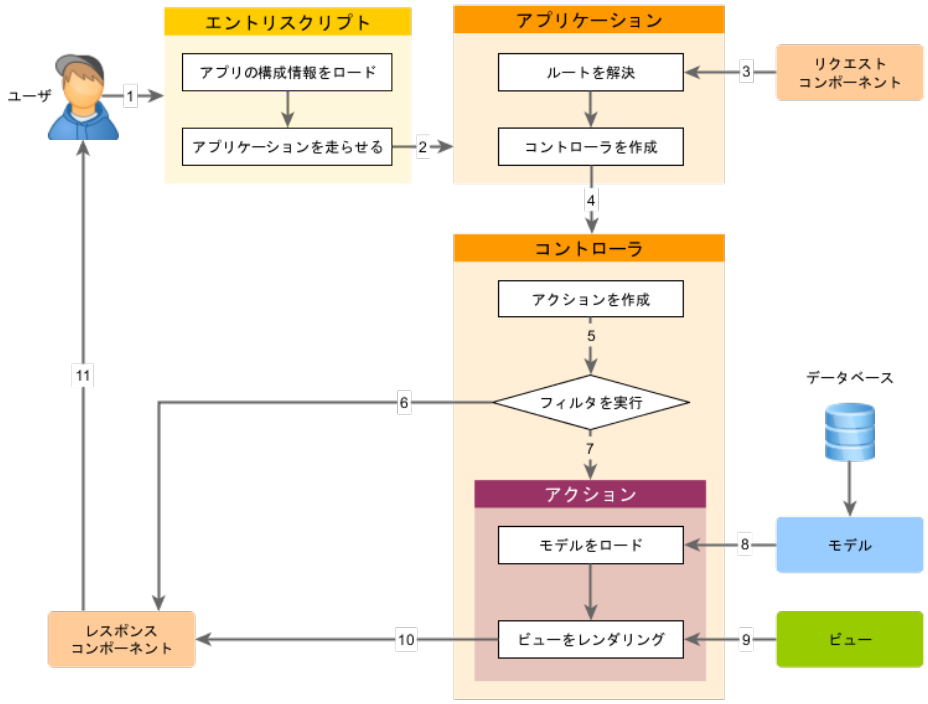
4.1 概要

Yii のアプリケーションがリクエストを処理するときは、毎回、同じようなワーク・フローになります。

1. ユーザが エントリ・スクリプト `web/index.php` にリクエストをします。
2. エントリ・スクリプトは、アプリケーションの 構成情報 をロードして、 リクエストを処理するための アプリケーション のインスタンスを作成します。
3. アプリケーションは、リクエスト アプリケーション・コンポーネントの助けを借りて、 リクエストされた ルート を解決します。
4. アプリケーションはリクエストを処理するための コントローラ のインスタンスを作成します。
5. コントローラは アクション のインスタンスを作成して、アクションのためのフィルタを実行します。
6. フィルタのどれかが失敗すると、アクションはキャンセルされます。
7. すべてのフィルタを無事に通ったら、アクションが実行されます。
8. アクションはデータモデルを、おそらくはデータベースから、ロードします。
9. アクションはデータ・モデルをビューに提供して、ビューをレンダリングします。
10. レンダリングの結果は レスポンス アプリケーション・コンポーネントに返されます。

11. レスポンス・コンポーネントがレンダリングの結果をユーザのブラウザに送信します。

次の図は、アプリケーションがどのようにしてリクエストを処理するかを示すものです。



このセクションでは、これらのステップのいくつかについて、どのように動作するかを詳細に説明します。

4.2 ブートストラップ

ブートストラップとは、アプリケーションが、入ってくるリクエストの解決と処理を開始する前に、環境を準備する過程を指すものです。 ブートストラップは二つの場所、すなわち、エントリ・スクリプト と アプリケーションで行われます。

エントリ・スクリプト では、さまざまなライブラリのためのクラス・オートローダが登録されます。 この中には、Composer の `autoload.php` によるオートローダと、Yii の `Yii` クラス・ファイルによるオートローダが含まれます。 エントリ・スクリプトは、次に、アプリケーションの 構成情報 をロードして、 アプリケーション のインスタンスを作成します。

アプリケーションのコンストラクタでは、次のようなブートストラップの仕事が行われます。

1. `preInit()` が呼ばれます。 このメソッドは、いくつかの優先度の高

いアプリケーション・プロパティ、例えば basePath などを構成します。

2. エラー・ハンドラ を登録します。
3. 与えられたアプリケーションの構成情報を使って、アプリケーションのプロパティを初期化します。
4. init() が呼ばれます。 そして init() が bootstrap() を呼んで、ブートストラップ・コンポーネントを走らせます。
   - エクステンション・マニフェスト・ファイル vendor/yiisoft/extensions.php をインクルードします。
   - エクステンションによって宣言された ブートストラップ・コンポーネント を作成して実行します。
   - アプリケーションの bootstrap プロパティ に宣言されている アプリケーション・コンポーネント および/または モジュール を作成して実行します。

ブートストラップの仕事は 全て のリクエストを処理する前に、毎回しなければなりませんので、 この過程を軽いものに保って可能な限り最適化することは非常に重要なことです。

あまりに多くのブートストラップ・コンポーネントを登録しないように努めてください。 ブートストラップ・コンポーネントが必要になるのは、リクエスト処理のライフサイクル全体に関与する必要がある場合だけです。 例えば、モジュールが追加の URL 解析規則を登録する必要がある場合は、モジュールを bootstrap プロパティ のリストに挙げなければなりません。 なぜなら、URL 規則を使ってリクエストが解決される前に、新しい URL 規則を有効にしなければならないからです。

本番運用モードにおいては、PHP OPCache[1] や APC[2] など、バイトコード・キャッシュを有効にして、 PHP ファイルをインクルードして解析するのに要する時間を最小化してください。

大規模なアプリケーションには、多数の小さな構成情報ファイルに分割された、非常に複雑なアプリケーション 構成情報 を持つものがあります。 そのような場合には、構成情報配列全体をキャッシュするという方法を考慮して下さい。 エントリ・スクリプトでアプリケーションのインスタンスを作成する前に構成情報をロードするときには、 配列全体を直接にキャッシュからロードするのです。

4.3 ルーティングと URL 生成

Yii のアプリケーションがリクエストされた URL の処理を開始するときに、最初に実行するステップは URL を解析して ルート にすることで

[1] https://secure.php.net/manual/ja/book.opcache.php
[2] https://secure.php.net/manual/ja/book.apc.php

す。 次に、リクエストを処理するために、このルートを使って、対応する コントローラ・アクション のインスタンスが作成されます。 このプロセスの全体が ルーティング と呼ばれます。

ルーティングの逆のプロセスが *URL* 生成 と呼ばれます。 これは、与えられたルートとそれに結び付けられたクエリ・パラメータから URL を生成するものです。 生成された URL が後でリクエストされたときには、ルーティングのプロセスがその URL を解決して元のルートとクエリ・パラメータに戻すことが出来ます。

ルーティングと URL 生成について主たる役割を果たすのが `urlManager` アプリケーション・コンポーネント として登録されている `URL` マネージャ です。 `URL` マネージャ は、入ってくるリクエストをルートとそれに結び付けられたクエリ・パラメータとして解析するための `parseRequest()` メソッドと、 与えられたルートとそれに結び付けられたクエリ・パラメータから URL を生成するための `createUrl()` メソッドを提供します。

アプリケーション構成情報の `urlManager` コンポーネントを構成することによって、既存のアプリケーション・コードを修正することなく、 任意の URL 形式をアプリケーションに認識させることが出来ます。 例えば、`post/view` アクションのための URL を生成するためには、次のコードを使うことが出来ます。

```
use yii\helpers\Url;

// Url::to() は UrlManager::createUrl() を呼び出して URL を生成します
$url = Url::to(['post/view', 'id' => 100]);
```

このコードによって生成される URL は、`urlManager` の構成に応じて、下記のどれか (またはその他) の形式になります。 そして、こうして生成された URL が後でリクエストされた場合には、解析されて元のルートとクエリ・パラメータの値に戻されます。

```
/index.php?r=post%2Fview&id=100
/index.php/post/100
/posts/100
```

4.3.1 URL 形式

`URL` マネージャ は二つの URL 形式をサポートします。すなわち、

- デフォルトの URL 形式と、
- 綺麗な URL (プリティ URL) の 形式。

デフォルトの URL 形式は、`r` という クエリ・パラメータ を使用してルートを表し、 通常のクエリ・パラメータを使用してルートに結び付けられたクエリ・パラメータを表します。 例えば、`/index.php?r=post/view&id=100` という URL は、`post/view` というルートと、`id` というクエリ・パラメータが 100 であることを表します。 デフォルトの URL 形式は、`URL` マネージャ についての構成を何も必要とせず、ウェブ・サーバの設定がどのようなものでも動作します。

綺麗な URL 形式は、エントリ・スクリプトの名前に続く追加のパスを使用して、ルートとそれに結び付けられたクエリ・パラメータを表します。 例えば、/index.php/post/100 という URL の追加のパスは /post/100 ですが、 適切な URL 規則 があれば、この追加のパスが post/view というルートと id のクエリ・パラメータ 100 を表すものとすることが出来ます。 綺麗な URL 形式を使用するためには、URL をどのように表現すべきかという実際の要求に従って、 一連の URL 規則 を設計する必要があります。

この二つの URL 形式は、URL マネージャ の enablePrettyUrl プロパティを ON/OFF することによって、 他のアプリケーション・コードを少しも変えることなく、切り替えることが出来ます。

4.3.2 ルーティング

ルーティングは二つのステップを含みます。

- まず、入ってくるリクエストが解析されて、ルートとそれに結び付けられたクエリ・パラメータに分解されます。
- そして、解析されたルートに対応する コントローラ・アクション がリクエストを処理するために生成されます。

デフォルトの URL 形式を使っている場合は、リクエストからルートを解析することは、 r という名前の GET クエリ・パラメータを取得するだけの簡単なことです。

綺麗な URL 形式を使っている場合は、URL マネージャ が、登録されている URL 規則 を調べます。 合致する規則が見つかれば、リクエストをルートに解決することが出来ます。 合致する規則が見つからなかったら、yii\web\NotFoundHttpException 例外が投げられます。

いったんリクエストからルートが解析されたら、今度はルートによって特定されるコントローラ・アクションを生成する番です。 ルートはその中にあるスラッシュによって複数の部分に分けられます。例えば、site/index は site と index に分割されます。 その各部分は、モジュール、コントローラ、または、アクションを参照する ID です。 アプリケーションは、ルートの最初の部分の ID から始めて、下記のステップを踏んで、モジュール (もし有れば)、コントローラ、アクションを生成します。

1. アプリケーションをカレント・モジュールとして設定します。
2. カレント・モジュールの コントローラ・マップ が現在の ID を含むかどうかを調べます。 含んでいる場合は、マップの中で見つかった構成情報に従ってコントローラのオブジェクトが生成されます。 そして、ステップ 5 に跳んで、ルートの残りの部分を処理します。
3. 現在の ID がカレント・モジュールの modules プロパティのリストに挙げられたモジュールを指すものかどうかを調べます。 もしそうであれば、モジュールのリストで見つかった構成情報に従ってモジュールが生成されます。 そして、新しく生成されたモジュールの

コンテキストのもとで、ステップ 2 に戻って、ルートの次の部分を処理します。

4. 現在の ID を コントローラ ID として扱ってコントローラ・オブジェクトを生成します。 そしてルートの残りの部分を持って次のステップに進みます。

5. コントローラは、アクション・マップ の中に現在の ID があるかどうかを調べます。 もし有れば、マップの中で見つかった構成情報に従ってアクションを生成します。 もし無ければ、現在の アクション ID に対応するアクション・メソッドで定義されるインライン・アクションを生成しようと試みます。

上記のステップの中で、何かエラーが発生すると、yii\web\NotFoundHttpException が投げられて、 ルーティングのプロセスが失敗したことが示されます。

デフォルト・ルート

リクエストから解析されたルートが空になった場合は、いわゆる デフォルト・ルート が代りに使用されることになります。 デフォルトでは、デフォルト・ルートは site/index であり、site コントローラの index アクションを指します。 デフォルト・ルートは、次のように、アプリケーションの構成情報の中でアプリケーションの defaultRoute プロパティを構成することによって、カスタマイズすることが出来ます。

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

アプリケーションのデフォルト・ルートと同じく、モジュールにもデフォルト・ルートがあります。 従って、例えば、user というモジュールがあって、リクエストの解析結果が user というルートになった場合、このモジュールの defaultRoute がコントローラを決定するのに使用されます。 デフォルトでは、このコントローラの名前は default となります。 defaultRoute でアクションが指定されていない場合は、コントローラの defaultAction プロパティがアクションを決定するのに使用されます。 この例の場合だと、完全なルートは user/default/index となります。

catchAll ルート

たまには、ウェブ・アプリケーションを一時的にメンテナンス・モードにして、全てのリクエストに対して同じ「お知らせ」のページを表示したいことがあるでしょう。 この目的を達する方法はたくさんありますが、最も簡単な方法の一つは、次のように、 アプリケーションの構成情報の中で yii\web\Application::$catchAll プロパティを構成することです。

```
[
    // ...
    'catchAll' => ['site/offline'],
];
```

上記の構成によって、入ってくる全てのリクエストを処理するために site/offline アクションが使われるようになります。

catchAll プロパティは配列を取り、最初の要素はルートを指定し、残りの要素 (「名前-値」のペア) は アクションのパラメータ を指定するものでなければなりません。

> 情報: このプロパティを有効にすると、開発環境で デバッグ・ツール・バー[3]が 動作しなくなります。

4.3.3 URL を生成する

Yii は、与えられたルートとそれに結び付けられたクエリ・パラメータからさまざまな URL を生成する yii\helpers\Url::to() というヘルパ・メソッドを提供しています。例えば、

```
use yii\helpers\Url;

// ルートへの URL を生成する: /index.php?r=post%2Findex
echo Url::to(['post/index']);

// パラメータを持つルートへの URL を生成する: /index.php?r=post%2Fview&id=100
echo Url::to(['post/view', 'id' => 100]);

// アンカー付きの URL を生成する: /index.php?r=post%2Fview&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// 絶対 URL を生成する: http://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], true);

// https スキームを使って絶対 URL を生成する: https://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], 'https');
```

上記の例では、デフォルトの URL 形式が使われていると仮定していることに注意してください。 綺麗な URL 形式が有効になっている場合は、生成される URL は、使われている URL 規則 に従って、異なるものになります。

yii\helpers\Url::to() メソッドに渡されるルートの意味は、コンテキストに依存します。 ルートは 相対 ルートか 絶対 ルートかのどちらかであり、下記の規則によって正規化されます。

- ルートが空文字列である場合は、現在リクエストされている ルートが使用されます。

[3] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide-ja/README.md

- ルートがスラッシュを全く含まない場合は、カレント・コントローラのアクション ID であると見なされて、 カレント・コントローラの uniqueId の値が前置されます。
- ルートが先頭にスラッシュを含まない場合は、カレント・モジュールに対する相対ルートと見なされて、 カレント・モジュールの uniqueId の値が前置されます。

バージョン 2.0.2 以降では、エイリアス の形式でルートを指定することが出来ます。 その場合は、エイリアスが最初に実際のルートに変換され、そのルートが上記の規則に従って絶対ルートに変換されます。

例えば、カレント・モジュールが admin であり、カレント・コントローラが post であると仮定すると、

```
use yii\helpers\Url;

// 現在リクエストされているルート: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['']);

// アクション ID だけの相対ルート: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['index']);

// 相対ルート: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['post/index']);

// 絶対ルート: /index.php?r=post%2Findex
echo Url::to(['/post/index']);

// "/post/index" と定義されているエイリアス "@postsを使
    用": /index.php?r=post%2Findex
echo Url::to(['@posts']);
```

yii\helpers\Url::to() メソッドは、URL マネージャ の createUrl() メソッド、および、createAbsoluteUrl() を呼び出すことによって実装されています。 次に続くいくつかの項では、URL マネージャ を構成して、生成される URL の形式をカスタマイズする方法を説明します。

yii\helpers\Url::to() メソッドは、特定のルートとの関係を持たない URL の生成もサポートしています。 その場合、最初のパラメータには、配列を渡す代りに文字列を渡さなければなりません。例えば、

```
use yii\helpers\Url;

// 現在リクエストされている URL: /index.php?r=admin%2Fpost%2Findex
echo Url::to();

// エイリアス化された URL: http://example.com
Yii::setAlias('@example', 'http://example.com/');
echo Url::to('@example');

// 絶対 URL: http://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

to() メソッドの他にも、yii\helpers\Url‘ ヘルパ・クラスは、便利な URL 生成メソッドをいくつか提供しています。 例えば、

```
use yii\helpers\Url;

// ホームページの URL: /index.php?r=site%2Findex
echo Url::home();

// ベース 。アプリケーションがウェブ・ルートのサブ・ディレクトリに配置されているときに便利URL
echo Url::base();

// 現在リクエストされている URL の canonical 。URL
// https://en.wikipedia.org/wiki/Canonical_link_element を参照
echo Url::canonical();

// 現在リクエストされている URL を記憶し、それを後のリクエストの中で呼び戻す。
Url::remember();
echo Url::previous();
```

4.3.4 綺麗な URL を使う

綺麗な URL を使うためには、アプリケーションの構成情報の中で urlManager コンポーネントを次のように構成します。

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

enablePrettyUrl プロパティは、綺麗な URL 形式の有効/無効を切り替えますので、必須です。 その他のプロパティはオプションですが、上記で示されている構成が最もよく用いられているものです。

- showScriptName: このプロパティは、生成される URL にエントリ・スクリプトを含めるべきかどうかを決定します。 例えば、このプロパティを false にすると、/index.php/post/100 という URL を生成する代りに、 /post/100 という URL を生成することが出来ます。
- enableStrictParsing: このプロパティは、厳密なリクエスト解析を有効にするかどうかを決定します。 厳密な解析が有効にされた場合、リクエストされた URL が有効なリクエストとして扱われるためには、それが rules の少なくとも一つに合致しなければなりません。 そうでなければ、yii\web\NotFoundHttpException が投げ

られます。 厳密な解析が無効にされた場合は、リクエストされた URL が rules のどれにも合致しない場合は、 URL のパス情報の部分がリクエストされたルートとして扱われます。

- rules: このプロパティが URL を解析および生成するための一連の規則を含みます。 このプロパティが、アプリケーションの固有の要求を満たす形式を持つ URL を生成するために、 あなたが主として使うプロパティです。

  補足: 生成された URL からエントリ・スクリプト名を隠すためには、showScriptName を false に設定するだけでなく、ウェブ・サーバを構成して、リクエストされた URL が PHP スクリプトを明示的に指定していない場合でも、 正しい PHP スクリプトを特定出来るようにする必要があります。 もしあなたが Apache または nginx ウェブ・サーバを使うつもりなら、インストール のセクションで説明されている推奨設定を参照することが出来ます。

URL 規則

URL 規則は yii\web\UrlRuleInterface を実装するクラス、通常は、yii\web\UrlRule クラスです。 すべての URL 規則は、URL のパス情報の部分との照合に使われるパターン、ルート、そして、いくつかのクエリ・パラメータから構成されます。 URL 規則は、パターンがリクエストされた URL と合致する場合に、リクエストの解析に使用することが出来ます。 また、URL 規則は、ルートとクエリ・パラメータ名が与えられたものと合致する場合に、URL の生成に使用することが出来ます。

綺麗な URL 形式が有効にされている場合、URL マネージャ は、その rules プロパティに宣言されている URL 規則を使って、 入ってくるリクエストの解析と URL の生成を行います。 具体的に言えば、入ってくるリクエストを解析するためには、URL マネージャ は宣言されている順に規則を調べて、 リクエストされた URL に合致する 最初の 規則を探します。 そして、その合致する規則を使って URL を解析して、ルートとそれに結び付けられたパラメータを得ます。 同じように、URL を生成するためには、URL マネージャ は、与えられたルートとパラメータに合致する最初の規則を探して、それを使って URL を生成します。

yii\web\UrlManager::$rules は、 パターン をキーとし、それに対応する ルート を値とする配列として構成することが出来ます。 「パターン - ルート」のペアが、それぞれ、URL 規則を構成します。 例えば、次の rules の構成は、二つの URL 規則を宣言するものです。 最初の規則は posts という URL に合致し、それを post/index というルートにマップします。 第二の規則は post/(\d+) という正規表現にマッチする URL に合致し、それを post/view というルートと id という名前のパラメータにマップします。

```
'rules' => [
```

```
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

> 情報: 規則のパターンは URL のパス情報の部分との照合に使用されます。 例えば、/index.php/post/100?source=ad のパス情報は post/100 であり (先頭と末尾のスラッシュは無視します)、 これは post/(\d+) というパターンに合致します。

URL 規則は、「パターン - ルート」のペアとして宣言する以外に、構成情報配列として宣言することも出来ます。 構成情報の一つの配列が、それぞれ、一つの URL 規則のオブジェクトを構成するために使われます。この形式は、URL 規則の他のプロパティを構成したい場合に、よく必要になります。例えば、

```
'rules' => [
    // ... 他の URL 規則 ...
    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

URL 規則の構成情報で class を指定しない場合は、デフォルトとして、yii\web\UrlRule が使われます。 このクラスが、yii\web\UrlManager::$ruleConfig で デフォルト値として定義されています。

名前付きパラメータ

URL 規則は、パターンの中で <ParamName:RgExp> の形式で指定される、名前付きクエリ・パラメータと結び付けることが出来ます。 ここで、ParamName はパラメータ名を指定し、RegExp はパラメータの値との照合に使われるオプションの正規表現を指定するものです。 RegExp が指定されていない場合は、 パラメータの値がスラッシュを含まない文字列であるべきことを意味します。

> 補足: 正規表現はパラメータの中でのみ使用できます。パターンの残りの部分はプレーンテキストとして解釈されます。

規則が URL の解析に使われるときには、URL の対応する部分に合致した値が、結び付けられたパラメータに入れられます。 そして、そのパラメータは、後に request アプリケーション・コンポーネントによって、$_GET に入れられて利用できるようになります。 規則が URL の生成に使われるときは、提供されたパラメータの値を受けて、 パラメータが宣言されている所にその値が挿入されます。

名前付きパラメータの動作を説明するためにいくつかの例を挙げましょう。次の三つの URL 規則を宣言したと仮定してください。

```
'rules' => [
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

規則が URL 解析に使われる場合は、

- /index.php/posts は、二番目の規則を使って解析され、ルート post/index になります。
- /index.php/posts/2014/php は、最初の規則を使って解析され、ルートは post/index、year パラメータの値は 2014、 そして、category パラメータの値は php となります。
- /index.php/post/100 は、三番目の規則を使って解析され、ルートが post/view、 id パラメータの値が 100 となります。
- /index.php/posts/php は、どのパターンにも合致しないため、yii\web\UrlManager::$enableStrictParsing が true の場合は、yii\web\NotFoundHttpException を引き起こします。 yii\web\UrlManager::$enableStrictParsing が false (これがデフォルト値です) の場合は、パス情報の部分である posts/php がルートとして返されることになります。 こうして解析されたルートに対応するアクションがあればそれが実行され、そうでなければ yii\web\NotFoundHttpException が投げられます。

規則が URL 生成に使われる場合は、

- Url::to(['post/index']) は、二番目の規則を使って、/index.php/posts を生成します。
- Url::to(['post/index', 'year' => 2014, 'category' => 'php']) は、最初の規則を使って、/index.php/posts/2014/php を生成します。
- Url::to(['post/view', 'id' => 100]) は、三番目の規則を使って、/index.php/post/100 を生成します。
- Url::to(['post/view', 'id' => 100, 'source' => 'ad']) も、三番目の規則を使って、/index.php/post/100?source=ad を生成します。 source パラメータは規則の中で指定されていないので、クエリ・パラメータとして、生成される URL に追加されます。
- Url::to(['post/index', 'category' => 'php']) は、どの規則も使わずに、/index.php/post/index?category=php を生成します。 どの規則も当てはまらないため、URL は、単純に、ルートをパス情報とし、すべてのパラメータをクエリ文字列として追加して生成されます。

ルートをパラメータ化する

URL 規則のルートにはパラメータ名を埋め込むことが出来ます。このことによって、URL 規則を複数のルートに合致させることが可能になっています。 例えば、以下の規則は controller と action というパラメータをルートに埋め込んでいます。

```
'rules' => [
```

```
    '<controller:(post|comment)>/create' => '<controller>/create',
    '<controller:(post|comment)>/<id:\d+>/<action:(update|delete)>' => '<
    controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

/index.php/comment/100/update という URL の解析には、二番目の規則が適用され、controller パラメータには comment、action パラメータには update がセットされます。 こうして、<controller>/<action> というルートは、comment/update として解決されます。

同じように、comment/index というルートの URL を生成するためには、最後の規則が適用されて、index.php/comments という URL が生成されます。

> 情報: ルートをパラメータ化することによって、URL 規則の数を大幅に減らすことが可能になり、 URL マネージャ のパフォーマンスを目に見えて改善することが出来ます。

デフォルトのパラメータ値

デフォルトでは、規則の中で宣言されたパラメータは必須となります。リクエストされた URL が特定のパラメータを含まない場合や、特定のパラメータなしで URL を生成する場合には、規則は適用されません。 パラメータのどれかをオプション扱いにしたい場合は、規則の defaults プロパティを構成することが出来ます。 このプロパティのリストに挙げられたパラメータはオプション扱いとなり、提供されなかった場合は指定された値を取るようになります。

次の規則の宣言においては、page と tag のパラメータは両方ともオプション扱いで、 提供されなかった場合は、それぞれ、1 と空文字列を取ります。

```
'rules' => [
    // ... 他の規則 ...
    [
        'pattern' => 'posts/<page:\d+>/<tag>',
        'route' => 'post/index',
        'defaults' => ['page' => 1, 'tag' => ''],
    ],
]
```

上記の規則を以下の URL を解析または生成するために使用することが出来ます。

- /index.php/posts: page は 1, tag は ''.
- /index.php/posts/2: page は 2, tag は ''.
- /index.php/posts/2/news: page は 2, tag は 'news'.
- /index.php/posts/news: page は 1, tag は 'news'.

オプション扱いのパラメータを使わなければ、同じ結果を得るために 4 個の規則を作らなければならなかったところです。

> 補足: pattern がオプション扱いのパラメータとスラッシュだけを含んでいるときは、 最初のパラメータは、他のパラメータが省略されている場合に限り、省略することが出来ます。

サーバ名を持つ規則

URL 規則のパターンには、ウェブ・サーバ名を含むことが出来ます。このことが役に立つのは、主として、あなたのアプリケーションがウェブ・サーバ名によって異なる動作をしなければならない場合です。 例えば、次の規則は、http://admin.example.com/login という URL を admin/user/login のルートとして解析し、http://www.example.com/login を site/login として解析するものです。

```
'rules' => [
    'http://admin.example.com/login' => 'admin/user/login',
    'http://www.example.com/login' => 'site/login',
]
```

サーバ名にパラメータを埋め込んで、そこから動的な情報を抽出することも出来ます。 例えば、次の規則は http://en.example.com/posts という URL を解析して、post/index というルートと language=en というパラメータを取得するものです。

```
'rules' => [
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

バージョン 2.0.11 以降は、http と https の両方に通用する、プロトコル相対パターンを使うことも出来ます。 記法は上記と同じです、ただ、http: の部分を省略します。例えば、'//www.example.com/login' => 'site/login'。

> 補足: サーバ名を持つ規則は、そのパターンに、エントリ・スクリプトのサブフォルダを含まないようにすべきです。例えば、アプリケーションのエントリ・スクリプトが http://www.example.com/sandbox/blog/index.php である場合は、http://www.example.com/sandbox/blog/posts ではなく、http://www.example.com/posts というパターンを使うべきです。 こうすれば、アプリケーションをどのようなディレクトリに配置しても、URL 規則を変更する必要がなくなります。Yii はアプリケーションのベース URL を自動的に検出します。

URL 接尾辞

さまざまな目的から URL に接尾辞を追加したいことがあるでしょう。例えば、静的な HTML ページに見えるように、.html を URL に追加したいかも知れません。 また、レスポンスとして期待されているコンテント・タイプを示すために、.json を URL に追加したい場合もあるでしょう。 アプリケーションの構成情報で、次のように、yii\web\UrlManager

::$suffix プロパティを構成することによって、この目的を達することが出来ます。

```
[
    // ...
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

上記の構成によって、URL マネージャ は、接尾辞として .html の付いた URL を認識し、 また、生成するようになります。

> ヒント: URL が全てスラッシュで終るようにするためには、URL 接尾辞として / を設定することが出来ます。
>
> 補足: URL 接尾辞を構成すると、リクエストされた URL が接尾辞を持たない場合は、認識できない URL であると見なされるようになります。 これは、異なる URL 上の重複コンテンツを防止するためのものであり、SEO (検索エンジン最適化) の見地からも推奨されるプラクティスです。

場合によっては、URL によって異なる接尾辞を使いたいことがあるでしょう。 その目的は、個々の URL 規則の suffix プロパティを構成することによって達成できます。 URL 規則にこのプロパティが設定されている場合は、それが URL マネージャ レベルの接尾辞の設定をオーバーライドします。 例えば、次の構成には、グローバルな接尾辞 .html の代りに .json を使用するカスタマイズされた URL 規則が含まれています。

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
]
```

HTTP メソッド

RESTful API を実装するときは、使用されている HTTP メソッドに応じて、同一の URL を異なるルートとして解析することが必要になる場合がよくあります。 これは、規則のパターンにサポートされている HTTP メソッドを前置することによって、簡単に達成することが出来ます。 一つの規則が複数の HTTP メソッドをサポートする場合は、メソッド名をカンマで区切ります。 例えば、次の三つの規則は、`post/<id:\d+>` という同一のパターンを持って、異なる HTTP メソッドをサポートするものです。 `PUT post/100` に対するリクエストは `post/update` と解析され、`GET post/100` に対するリクエストは `post/view` と解析されることになります。

```
'rules' => [
    'PUT,POST post/<id:\d+>' => 'post/update',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

> 補足: URL 規則が HTTP メソッドをパターンに含む場合、指定されたメソッドに `GET` が入っていない限り、その規則は解析目的にだけ使用されます。 `URL` マネージャ が URL 生成のために呼ばれたときは、その規則はスキップされます。

> ヒント: RESTful API のルーティングを簡単にするために、Yii は特別な URL 規則クラス `yii\rest\UrlRule` を提供しています。 これは非常に効率的なもので、コントローラ ID の自動的な複数形化など、いくつかの素敵な機能をサポートするものです。 詳細については、RESTful API 開発についての ルーティング のセクションを参照してください。

規則を動的に追加する

URL 規則は `URL` マネージャ に動的に追加することが出来ます。 このことは、再配布可能な モジュール が自分自身の URL 規則を管理する必要がある場合に、しばしば必要になります。 動的に追加された規則がルーティングのプロセスで効果を発揮するためには、その規則をアプリケーションの ブートストラップ の段階で追加しなければなりません。 これは、モジュールにとっては、次のように、`yii\base\BootstrapInterface` を実装して、 `bootstrap()` メソッドの中で規則を追加しなければならないことを意味します。

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // ここに規則の宣言
    ], false);
}
```

さらに、モジュールが ブートストラップ の過程に関与できるように、 それを yii\web\Application::bootstrap() のリストに挙げなければならないことに注意してください。

規則クラスを作成する

デフォルトの yii\web\UrlRule クラスはほとんどのプロジェクトに対して十分に柔軟なものであるというのは事実ですが、それでも、自分自身で規則クラスを作る必要があるような状況はあります。 例えば、自動車ディーラーのウェブ・サイトにおいて、/Manufacturer/Model のような URL 形式をサポートしたいけれども、 Manufacturer と Model は、両方とも、データベース・テーブルに保存されている何らかのデータに合致するものでなければならない、というような場合です。 デフォルトの規則クラスは、静的に宣言されるパターンに依拠しているため、ここでは役に立ちません。

この問題を解決するために、次のような URL 規則クラスを作成することが出来ます。

```
<?php

namespace app\components;

use yii\web\UrlRuleInterface;
use yii\base\BaseObject;

class CarUrlRule extends BaseObject implements UrlRuleInterface
{
    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {
                return $params['manufacturer'];
            }
        }
        return false; // この規則は適用されない
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\w+)(/(\w+))?$%', $pathInfo, $matches)) {
            // $matches[1] と $matches[3] をチェックして、
            // データベースの中の製造者とモデルに合致するかどうか調べる
            // 合致すれば、$params['manufacturer'] およびまた
    は/ $params['model']
            // をセットし、['car/index', $params] を返す
        }
        return false; // この規則は適用されない
```

```
    }
}
```

そして、yii\web\UrlManager::$rules の構成情報で、新しい規則クラスを使います。

```
'rules' => [
    // ... 他の規則 ...
    [
        'class' => 'app\components\CarUrlRule',
        // ... 他のプロパティを構成する ...
    ],
]
```

4.3.5 URL の正規化

バージョン 2.0.10 以降、UrlManager で UrlNormalizer を使って、 同一 URL のバリエーション (例えば、末尾のスラッシュの有無) の問題を処理する出来るようになりました。 技術的には http://example.com/path と http://example.com/path/ は別の URL ですから、これらの両方に同一のコンテントを提供することは SEO ランキングを低下させる可能性があります。 デフォルトでは、URL ノーマライザは、連続したスラッシュを畳み、サフィックスが末尾のスラッシュを持っているかどうかに従って末尾のスラッシュを追加または削除し、 正規化された URL に 恒久的な移動[4] を使ってリダイレクトします。 ノーマライザは、URL マネージャのためにグローバルに構成することも、各規則のために個別に構成することも出来ます。 各規則は、デフォルトでは、URL マネージャのノーマライザを使用します。 UrlRule::$normalizer を false にすれば、特定の URL 規則について正規化を無効にすることが出来ます。

次に、UrlNormalizer の構成例を示します。

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'enableStrictParsing' => true,
    'suffix' => '.html',
    'normalizer' => [
        'class' => 'yii\web\UrlNormalizer',
        // デバッグのために、恒久的移動のかわりに一時的リダイレクションを使う
        'action' => UrlNormalizer::ACTION_REDIRECT_TEMPORARY,
    ],
    'rules' => [
        // ... 他の規則 ...
        [
            'pattern' => 'posts',
            'route' => 'post/index',
            'suffix' => '/',
            'normalizer' => false, // この規則では正規化を無効にする
```

[4] https://en.wikipedia.org/wiki/HTTP_301

```
        ],
        [
            'pattern' => 'tags',
            'route' => 'tag/index',
            'normalizer' => [
                // この規則では連続するスラッシュを畳まない
                'collapseSlashes' => false,
            ],
        ],
    ],
]
```

> 補足: デフォルトでは `UrlManager::$normalizer` は無効になっています。 URL の正規化を有効にするためには、明示的に構成する必要があります。

4.3.6 パフォーマンスに対する考慮

複雑なウェブ・アプリケーションを開発するときは、リクエストの解析と URL 生成に要する時間を削減するために URL 規則を最適化することが重要になります。

パラメータ化したルートを使うことによって、URL 規則の数を減らして、パフォーマンスを著しく向上させることが出来ます。

URL を解析または生成するときに、`URL` マネージャ は、宣言された順序で URL 規則を調べます。 従って、より多く使われる規則がより少なく使われる規則より前に来るように順序を調整することを検討してください。

パターンまたはルートに共通の先頭部分を持つ URL 規則がある場合は、`URL` マネージャ がそれらをグループ化して効率的に調べることが出来るように、 `yii\web\GroupUrlRule` を使うことを検討してください。 あなたのアプリケーションがモジュールによって構成されており、モジュールごとに、モジュール ID を共通の先頭部分とする一群の URL 規則を持っている場合は、通常、このことが当てはまります。

4.4 リクエスト

アプリケーションに対するリクエストは、リクエストのパラメータ、HTTP ヘッダ、クッキーなどの情報を提供する `yii\web\Request` オブジェクトの形で表されます。 与えられたリクエストに対応するリクエスト・オブジェクトには、デフォルトでは `yii\web\Request` のインスタンスである `request` アプリケーション・コンポーネント を通じてアクセスすることが出来ます。 このセクションでは、アプリケーションの中でこのコンポーネントをどのように利用できるかを説明します。

4.4.1 リクエストのパラメータ

リクエストのパラメータを取得するためには、request コンポーネントの get() および post() メソッドを呼ぶことが出来ます。 これらは、ぞれぞれ、$_GET と $_POST の値を返します。例えば、

```
$request = Yii::$app->request;

$get = $request->get();
// $get = $_GET; と同等

$id = $request->get('id');
// $id = isset($_GET['id']) ? $_GET['id'] : null; と同等

$id = $request->get('id', 1);
// $id = isset($_GET['id']) ? $_GET['id'] : 1; と同等

$post = $request->post();
// $post = $_POST; と同等

$name = $request->post('name');
// $name = isset($_POST['name']) ? $_POST['name'] : null; と同等

$name = $request->post('name', '');
// $name = isset($_POST['name']) ? $_POST['name'] : ''; と同等
```

> 情報: 直接に $_GET と $_POST にアクセスしてリクエストのパラメータを読み出す代りに、上記に示されているように、request コンポーネントを通じてそれらを取得することが推奨されます。 このようにすると、ダミーのリクエスト・データを持った模擬リクエスト・コンポーネントを作ることが出来るため、テストを書くことがより容易になります。

RESTful API を実装するときは、PUT、PATCH またはその他の リクエスト・メソッド によって送信されたパラメータを読み出さなければならないことがよくあります。 そういうパラメータは yii\web\Request::getBodyParam() メソッドを呼ぶことで取得することが出来ます。 例えば、

```
$request = Yii::$app->request;

// 全てのパラメータを返す
$params = $request->bodyParams;

// パラメータ "id" を返す
$param = $request->getBodyParam('id');
```

> 情報: GET パラメータとは異なって、POST、PUT、PATCH などで送信されたパラメータは、リクエストのボディの中で送られ

ます。 上述のメソッドによってこれらのパラメータにアクセスすると、request コンポーネントがパラメータを解析します。 yii\web\Request::$parsers プロパティを構成することによって、これらのパラメータが解析される方法をカスタマイズすることが出来ます。

4.4.2 リクエスト・メソッド

現在のリクエストに使用された HTTP メソッドは、Yii::$app->request->method という式によって取得することが出来ます。 現在のメソッドが特定のタイプであるかどうかをチェックするための、一揃いの真偽値のプロパティも提供されています。 例えば、

```
$request = Yii::$app->request;

if ($request->isAjax) { /* リクエストは AJAX リクエスト */ }
if ($request->isGet)  { /* リクエスト・メソッドは GET */ }
if ($request->isPost) { /* リクエスト・メソッドは POST */ }
if ($request->isPut)  { /* リクエスト・メソッドは PUT */ }
```

4.4.3 リクエストの URL

request コンポーネントは現在リクエストされている URL を調べるための方法を数多く提供しています。

リクエストされた URL が http://example.com/admin/index.php/product?id=100 であると仮定したとき、 次にまとめたように、この URL のさまざまな部分を取得することが出来ます。

- url: /admin/index.php/product?id=100 を返します。ホスト情報の部分を省略した URL です。
- absoluteUrl: http://example.com/admin/index.php/product?id=100 を返します。 ホスト情報の部分を含んだ URL です。
- hostInfo: http://example.com を返します。URL のホスト情報の部分です。
- pathInfo: /product を返します。 エントリ・スクリプトの後、疑問符 (クエリ文字列) の前の部分です。
- queryString: id=100 を返します。疑問符の後の部分です。
- baseUrl: /admin を返します。ホスト情報の後、かつ、 エントリ・スクリプトの前の部分です。
- scriptUrl: /admin/index.php を返します。パス情報とクエリ文字列を省略した URL です。
- serverName: example.com を返します。URL の中のホスト名です。
- serverPort: 80 を返します。ウェブ・サーバによって使用されているポートです。

4.4.4 HTTP ヘッダ

yii\web\Request::$headers プロパティによって返される header コレクション を通じて、 HTTP ヘッダ情報を取得することが出来ます。例えば、

```
// $headers は yii\web\HeaderCollection のオブジェクト
$headers = Yii::$app->request->headers;

// Accept ヘッダの値を返す
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* User-Agent ヘッダが在る */ }
```

request コンポーネントは、よく使用されるいくつかのヘッダにすばやくアクセスする方法を提供しています。その中には下記のものが含まれます。

- userAgent: User-Agent ヘッダの値を返します。
- contentType: リクエスト・ボディのデータの MIME タイプを示す Content-Type ヘッダの値を返します。
- acceptableContentTypes: ユーザが受け入れ可能なコンテントの MIME タイプを返します。 返されるタイプは品質スコアによって順序付けられます。最もスコアの高いタイプが最初に返されます。
- acceptableLanguages: ユーザが受け入れ可能な言語を返します。返される言語は優先レベルによって順序付けられます。最初の要素が最も優先度の高い言語を表します。

あなたのアプリケーションが複数の言語をサポートしており、エンド・ユーザが最も優先する言語でページを表示したいと思う場合は、 言語ネゴシエーション・メソッド yii\web\Request::getPreferredLanguage() を使うことが出来ます。 このメソッドはアプリケーションによってサポートされている言語のリストを引数として取り、 acceptableLanguages と比較して、最も適切な言語を返します。

> ヒント: ContentNegotiator フィルタを使用して、 レスポンスにおいてどのコンテント・タイプと言語を使うべきかを動的に決定することも出来ます。 このフィルタは、上記で説明したプロパティとメソッドの上に、コンテント・ネゴシエーションを実装しています。

4.4.5 クライアント情報

クライアント・マシンのホスト名と IP アドレスを、 それぞれ、userHost と userIP によって取得することが出来ます。 例えば、

```
$userHost = Yii::$app->request->userHost;
$userIP = Yii::$app->request->userIP;
```

4.4.6 信頼できるプロキシとヘッダ

前のセクションでホストや IP アドレスなどのユーザ情報を取得する方法を説明しました。 単一のウェブ・サーバがウェブ・サイトをホストしている通常の環境では、このままで動作します。 しかし、Yii アプリケーションがリバース・プロキシの背後で動作している場合は、この情報を読み出すために構成情報を追加する必要があります。 なぜなら、その場合、直接のクライアントはプロキシになっており、ユーザの IP アドレスはプロキシがセットするヘッダによって Yii アプリケーションに渡されるからです。

明示的に信頼したプロキシ以外は、プロキシによって提供されるヘッダを盲目的に信頼してはいけません。 2.0.13 以降、Yii は `request` コンポーネントの以下のプロパティによって、 信頼できるプロキシの情報を構成することが出来るようになっています。 `trustedHosts`、`secureHeaders`、 `ipHeaders` および `secureProtocolHeaders`

以下は、リバース・プロキシ・アレイの背後で動作するアプリケーションのための、request の構成例です (リバース・プロキシ・アレイは `10.0.2.0/24` のネットワークに設置されているとします)。

```
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24',
    ],
],
```

プロキシは、デフォルトでは、IP を `X-Forwarded-For` ヘッダで送信し、プロトコル (`http` または `https`) を `X-Forwarded-Proto` で送信します。

あなたのプロキシが異なるヘッダを使っている場合は、request の構成情報を使って調整することが出来ます。例えば、

```
'request' => [
    // ...
    'trustedHosts' => [
        '10.0.2.0/24' => [
            'X-ProxyUser-Ip',
            'Front-End-Https',
        ],
    ],
    'secureHeaders' => [
        'X-Forwarded-For',
        'X-Forwarded-Host',
        'X-Forwarded-Proto',
        'X-Proxy-User-Ip',
        'Front-End-Https',
    ],
    'ipHeaders' => [
        'X-Proxy-User-Ip',
    ],
    'secureProtocolHeaders' => [
        'Front-End-Https' => ['on']
```

```
    ],
],
```

上記の構成によって、secureHeaders としてリストされているヘッダはリクエストから除去され、 信頼できるプロキシからのリクエストである場合にのみ、X-ProxyUser-Ip と Front-End-Https ヘッダが受け入れられます。 その場合、前者は ipHeaders で構成されているようにユーザの IP を読み出すために使用され、 後者は yii\web\Request::getIsSecureConnection() の結果を決定するために使用されます。

2.0.31 以降、RFC 7239[5] の Forwarded ヘッダがサポートされています。有効にするためには、ヘッダ名を secureHeaders に追加する必要があります。 あなたのプロキシにそれを設定させることを忘れないで下さい。さもないと、エンド・ユーザが IP とプロトコルを盗み見ることが可能になります。

解決済みのユーザ IP

ユーザの IP アドレスが Yii アプリケーション以前に解決済みである場合(例えば、ngx_http_realip_module など) は、 request コンポーネントは下記の構成で正しく動作します。

```
'request' => [
    // ...
    'trustedHosts' => [
        '0.0.0.0/0',
    ],
    'ipHeaders' => [],
],
```

この場合、userIP の値は $_SERVER['REMOTE_ADDR'] に等しくなります。 同時に、HTTP ヘッダから解決されるプロパティも正しく動作します (例えば、yii\web\Request::getIsSecureConnection())。

> 注意: trustedHosts=['0.0.0.0/0'] の設定は、全ての IP が信頼できることを前提としています。

4.5 レスポンス

アプリケーションは リクエスト の処理を完了すると、レスポンス・オブジェクトを生成して、エンド・ユーザに送信します。 レスポンス・オブジェクトは、HTTP ステータス・コード、HTTP ヘッダ、HTTP ボディなどの情報を含みます。 ウェブ・アプリケーション開発の最終的な目的は、本質的には、さまざまなリクエストに対してそのようなレスポンス・オブジェクトを作成することにあります。

[5] https://tools.ietf.org/html/rfc7239

ほとんどの場合は、主として、デフォルトでは yii\web\Response のインスタンスである response アプリケーション・コンポーネント を使用すべきです。 しかしながら、Yii は、以下で説明するように、あなた自身のレスポンス・オブジェクトを作成してエンド・ユーザに送信することも許容しています。

このセクションでは、レスポンスを構成してエンド・ユーザに送信する方法を説明します。

4.5.1 ステータス・コード

レスポンスを作成するときに最初にすることの一つは、リクエストが成功裡に処理されたかどうかを記述することです。 そのためには、yii\web\Response::$statusCode プロパティに有効な HTTP ステータス・コード[6] の一つを設定します。 例えば、下記のように、リクエストの処理が成功したことを示すために、ステータス・コードを 200 に設定します。

```
Yii::$app->response->statusCode = 200;
```

ただし、たいていの場合、ステータス・コードを明示的に設定する必要はありません。 これは、yii\web\Response::$statusCode のデフォルト値が 200 であるからです。 そして、リクエストが失敗したことを示したいときは、下記のように、適切な HTTP 例外を投げることが出来ます。

```
throw new \yii\web\NotFoundHttpException;
```

エラー・ハンドラ は、例外をキャッチすると、例外からステータス・コードを抽出してレスポンスに割り当てます。 上記の yii\web\NotFoundHttpException の場合は、HTTP ステータス 404 と関連付けられています。 次の HTTP 例外が Yii によって事前定義されています。

- yii\web\BadRequestHttpException: ステータス・コード 400
- yii\web\ConflictHttpException: ステータス・コード 409
- yii\web\ForbiddenHttpException: ステータス・コード 403
- yii\web\GoneHttpException: ステータス・コード 410
- yii\web\MethodNotAllowedHttpException: ステータス・コード 405
- yii\web\NotAcceptableHttpException: ステータス・コード 406
- yii\web\NotFoundHttpException: ステータス・コード 404
- yii\web\ServerErrorHttpException: ステータス・コード 500
- yii\web\TooManyRequestsHttpException: ステータス・コード 429
- yii\web\UnauthorizedHttpException: ステータス・コード 401
- yii\web\UnsupportedMediaTypeHttpException: ステータス・コード 415

投げたい例外が上記のリストに無い場合は、yii\web\HttpException から拡張したものを作成することが出来ます。 あるいは、ステータス・コードを指定して yii\web\HttpException を直接に投げることも出来ます。例えば、

[6] https://tools.ietf.org/html/rfc2616#section-10

```
throw new \yii\web\HttpException(402);
```

4.5.2 HTTP ヘッダ

response コンポーネントの ヘッダ・コレクション を操作することによって、HTTP ヘッダを送信することが出来ます。 例えば、

```
$headers = Yii::$app->response->headers;

// Pragma ヘッダを追加する。既存の Pragma ヘッダは上書きされない。
$headers->add('Pragma', 'no-cache');

// Pragma ヘッダを設定する。既存の Pragma ヘッダは全て破棄される。
$headers->set('Pragma', 'no-cache');

// Pragma ヘッダを削除して、削除された Pragma ヘッダの値を配列に返す。
$values = $headers->remove('Pragma');
```

> 情報: ヘッダ名は大文字小文字を区別しません。 そして、新しく登録されたヘッダは、yii\web\Response::send() メソッドが呼ばれるまで送信されません。

4.5.3 レスポンス・ボディ

ほとんどのレスポンスは、エンド・ユーザに対して表示したい内容を示すボディを持っていなければなりません。

既にフォーマットされたボディの文字列を持っている場合は、それをレスポンスの yii\web\Response::$content プロパティに割り付けることが出来ます。 例えば、

```
Yii::$app->response->content = 'hello world!';
```

データをエンド・ユーザに送信する前にフォーマットする必要がある場合は、format と data の両方のプロパティをセットしなければなりません。 format プロパティは data がどの形式でフォーマットされるべきかを指定するものです。 例えば、

```
$response = Yii::$app->response;
$response->format = \yii\web\Response::FORMAT_JSON;
$response->data = ['message' => 'hello world'];
```

Yii は下記の形式を初めからサポートしています。それぞれ、フォーマッタ クラスとして実装されています。 yii\web\Response::$formatters プロパティを構成することで、これらのフォーマッタをカスタマイズしたり、新しいフォーマッタを追加したりすることが出来ます。

- HTML: yii\web\HtmlResponseFormatter によって実装
- XML: yii\web\XmlResponseFormatter によって実装
- JSON: yii\web\JsonResponseFormatter によって実装
- JSONP: yii\web\JsonResponseFormatter によって実装

- RAW: 書式を何も適用せずにレスポンスを送信したいときは、このフォーマットを使用

レスポンス・ボディは、上記のように、明示的に設定することも出来ますが、たいていの場合は、アクション メソッドの返り値によって暗黙のうちに設定することが出来ます。 よくあるユースケースは下記のようなものになります。

```
public function actionIndex()
{
    return $this->render('index');
}
```

上記の index アクションは、index ビューのレンダリング結果を返しています。 返された値は response コンポーネントによって受け取られ、フォーマットされてエンド・ユーザに送信されます。

デフォルトのレスポンス形式が HTML であるため、アクション・メソッドの中では文字列を返すだけにすべきです。 別のレスポンス形式を使いたい場合は、データを返す前にレスポンス形式を設定しなければなりません。 例えば、

```
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    return [
        'message' => 'hello world',
        'code' => 100,
    ];
}
```

既に述べたように、デフォルトの response アプリケーション・コンポーネントを使う代りに、自分自身のレスポンス・オブジェクトを作成してエンド・ユーザに送信することも出来ます。 そうするためには、次のように、アクション・メソッドの中でそのようなオブジェクトを返します。

```
public function actionInfo()
{
    return \Yii::createObject([
        'class' => 'yii\web\Response',
        'format' => \yii\web\Response::FORMAT_JSON,
        'data' => [
            'message' => 'hello world',
            'code' => 100,
        ],
    ]);
}
```

> 補足: 自分自身のレスポンス・オブジェクトを作成しようとする場合は、アプリケーションの構成情報で response コンポーネントのために設定した構成情報を利用することは出来ません。 しかし、 依存の注入 を使えば、 共通の構成情報をあな

たの新しいレスポンス・オブジェクトに適用することが出来ます。

4.5.4 ブラウザのリダイレクト

ブラウザのリダイレクトは Location HTTP ヘッダの送信に依存しています。 この機能は通常よく使われるものであるため、Yii はこれについて特別のサポートを提供しています。

yii\web\Response::redirect() メソッドを呼ぶことによって、ユーザのブラウザをある URL にリダイレクトすることが出来ます。 このメソッドは与えられた URL を持つ適切な Location ヘッダを設定して、レスポンス・オブジェクトそのものを返します。 アクション・メソッドの中では、そのショートカット版である yii\web\Controller::redirect() を呼ぶことが出来ます。例えば、

```
public function actionOld()
{
    return $this->redirect('http://example.com/new', 301);
}
```

上記のコードでは、アクション・メソッドが redirect() メソッドの結果を返しています。 前に説明したように、アクション・メソッドによって返されるレスポンス・オブジェクトが、エンド・ユーザに送信されるレスポンスとして使用されることになります。

アクション・メソッド以外の場所では、yii\web\Response::redirect() を直接に呼び出し、 メソッド・チェーンで yii\web\Response::send() メソッドを呼んで、レスポンスに余計なコンテントが追加されないことを保証しなければなりません。

```
\Yii::$app->response->redirect('http://example.com/new', 301)->send();
```

> 情報: デフォルトでは、yii\web\Response::redirect() メソッドはレスポンスのステータス・コードを 302 にセットします。 これはブラウザに対して、リクエストされているリソースが 一時的に 異なる URI に配置されていることを示すものです。 ブラウザに対してリソースが 恒久的に 配置替えされたことを教えるためには、ステータス・コード 301 を渡すことが出来ます。

現在のリクエストが AJAX リクエストである場合は、Location ヘッダを送っても自動的にブラウザをリダイレクトすることにはなりません。この問題を解決するために、yii\web\Response::redirect() メソッドは X-Redirect ヘッダにリダイレクト先 URL を値としてセットします。 そして、クライアント・サイドで、このヘッダの値を読み、 それに応じてブラウザをリダイレクトする JavaScript を書くことが出来ます。

> 情報: Yii には yii.js という JavaScript ファイルが付属しています。 これは、よく使われる一連の JavaScript 機能を提供するもので、その中には X-Redirect ヘッダに基づくブラウザのリダイレクトも含まれています。 従って、あなたが (yii\web\YiiAsset アセット・バンドルを登録して) この JavaScript ファイルを使うつもりなら、AJAX のリダイレクトをサポートするためには、何も書く必要がなくなります。 yii.js に関する更なる情報は クライアント・スクリプトのセクション にあります。

4.5.5 ファイルを送信する

ブラウザのリダイレクトと同じように、ファイルの送信という機能も特定の HTTP ヘッダに依存しています。 Yii はさまざまなファイル送信の必要をサポートするための一連のメソッドを提供しています。それらはすべて、HTTP range ヘッダに対するサポートを内蔵しています。

- yii\web\Response::sendFile(): 既存のファイルをクライアントに送信する
- yii\web\Response::sendContentAsFile(): テキストの文字列をファイルとしてクライアントに送信する
- yii\web\Response::sendStreamAsFile(): 既存のファイル・ストリームをファイルとしてクライアントに送信する

これらのメソッドは同じメソッド・シグニチャを持ち、返り値としてレスポンス・オブジェクトを返します。 送信しようとしているファイルが非常に大きなものである場合は、メモリ効率の良い yii\web\Response::sendStreamAsFile() の使用を検討すべきです。 次の例は、コントローラ・アクションでファイルを送信する方法を示すものです。

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

ファイル送信メソッドをアクション・メソッド以外の場所で呼ぶ場合は、その後で yii\web\Response::send() メソッドも呼んで、 レスポンスに余計なコンテントが追加されないことを保証しなければなりません。

```
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

ウェブ・サーバには、*X-Sendfile* と呼ばれる特別なファイル送信をサポートするものがあります。 アイデアとしては、ファイルに対するリクエストをウェブ・サーバにリダイレクトして、ウェブ・サーバに直接にファイルを送信させる、というものです。 その結果として、ウェブ・サーバがファイルを送信している間でも、ウェブ・アプリケーションは早期に終了することが出来るようになります。 この機能を使うために、yii\web\Response::xSendFile() を呼ぶことが出来ます。 次のリストは、

よく使われるいくつかのウェブ・サーバにおいて X-Sendfile 機能を有効にする方法を要約するものです。

- Apache: X-Sendfile[7]
- Lighttpd v1.4: X-LIGHTTPD-send-file[8]
- Lighttpd v1.5: X-Sendfile[9]
- Nginx: X-Accel-Redirect[10]
- Cherokee: X-Sendfile and X-Accel-Redirect[11]

4.5.6 レスポンスを送信する

レスポンスの中のコンテントは、yii\web\Response::send() メソッドが呼ばれるまでは、エンド・ユーザに向けて送信されません。 デフォルトでは、このメソッドは yii\base\Application::run() の最後で自動的に呼ばれます。 しかし、このメソッドを明示的に呼んで、強制的にレスポンスを即座に送信することも可能です。

yii\web\Response::send() メソッドは次のステップを踏んでレスポンスを送出します。

1. yii\web\Response::EVENT_BEFORE_SEND イベントをトリガする。
2. yii\web\Response::prepare() を呼んで レスポンス・データ を レスポンス・コンテント としてフォーマットする。
3. yii\web\Response::EVENT_AFTER_PREPARE イベントをトリガする。
4. yii\web\Response::sendHeaders() を呼んで、登録された HTTP ヘッダを送出する。
5. yii\web\Response::sendContent() を呼んで、レスポンスのボディ・コンテントを送出する。
6. yii\web\Response::EVENT_AFTER_SEND イベントをトリガする。

yii\web\Response::send() メソッドが一度呼び出された後では、このメソッドに対する更なる呼び出しは無視されます。 このことは、いったんレスポンスが送出された後では、それにコンテントを追加することは出来なくなる、ということを意味します。

ごらんのように、yii\web\Response::send() メソッドはいくつかの有用なイベントをトリガします。 これらのイベントに反応することによって、レスポンスを調整したり修飾したりすることが出来ます。

[7] http://tn123.org/mod_xsendfile
[8] http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[9] http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[10] http://wiki.nginx.org/XSendfile
[11] http://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

4.6 セッションとクッキー

セッションとクッキーは、データが複数回のユーザ・リクエストにまたがって持続することを可能にします。 素の PHP では、それぞれ、グローバル変数 `$_SESSION` と `$_COOKIE` によってアクセスすることが出来ます。 Yii はセッションとクッキーをオブジェクトとしてカプセル化し、オブジェクト指向の流儀でアクセスできるようにするとともに、有用な機能強化を追加しています。

4.6.1 セッション

リクエスト や レスポンス と同じように、 デフォルトでは `yii\web\Session` のインスタンスである `session` [アプリケーション・コンポーネント] によって、 セッションにアクセスすることが出来ます。

セッションのオープンとクローズ

セッションのオープンとクローズは、次のようにして出来ます。

```
$session = Yii::$app->session;

// セッションが既に開かれているかチェックする
if ($session->isActive) ...

// セッションを開く
$session->open();

// セッションを閉じる
$session->close();

// セッションに登録されている全てのデータを破壊する
$session->destroy();
```

エラーを発生させずに `open()` と `close()` を複数回呼び出すことが出来ます。 内部的には、これらのメソッドは、セッションが既に開かれているかどうかを最初にチェックします。

セッション・データにアクセスする

セッションに保存されているデータにアクセスするためには、次のようにすることが出来ます。

```
$session = Yii::$app->session;

// セッション変数を取得する。次の三つの用法は等価。
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;

// セッション変数を設定する。次の三つの用法は等価。
```

```
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';

// セッション変数を削除する。次の三つの用法は等価。
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// セッション変数が存在するかどうかをチェックする。次の三つの用法は等価。
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// 全てのセッション変数をたどる。次の二つの用法は等価。
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

> 情報: session コンポーネントによってセッション・データにアクセスする場合は、まだ開かれていないときは、自動的にセッションが開かれます。 これに対して $_SESSION によってセッション・データにアクセスする場合は、 session_start() を明示的に呼び出すことが必要になります。

配列であるセッション・データを扱う場合、session コンポーネントには、配列の要素を直接修正することができない、という制約があります。 例えば、

```
$session = Yii::$app->session;

// 次のコードは動かない
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// 次のコードは動く
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// 次のコードも動く
echo $session['captcha']['lifetime'];
```

次の回避策のどれかを使ってこの問題を解決することが出来ます。

```
$session = Yii::$app->session;

// $_SESSION を直接使う 既に( Yii::$app->session->open() が呼び出されていること を確認)
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;
```

```
// 配列全体を取得し、修正して、保存しなおす
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// 配列の代わりに ArrayObject を使う
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// 共通の接頭辞を持つキーを使って配列データを保存する
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

パフォーマンスとコードの可読性を高めるためには、最後の回避策を推奨します。 すなわち、配列を一つのセッション変数として保存する代りに、配列の個々の要素を、他の配列要素と共通の接頭辞を持つ、 個別のセッション変数として保存することです。

カスタム・セッション・ストレージ

デフォルトの `yii\web\Session` クラスはセッション・データをサーバ上のファイルとして保存します。 Yii は、また、さまざまなセッション・ストレージを実装する下記のクラスをも提供しています。

- `yii\web\DbSession`: セッション・データをデータベース・テーブルを使って保存する。
- `yii\web\CacheSession`: セッション・データを、構成された キャッシュ・コンポーネント の力を借りて、キャッシュを使って保存する。
- `yii\redis\Session`: セッション・データを redis[12] をストレージ媒体として使って保存する。
- `yii\mongodb\Session`: セッション・データを MongoDB[13] に保存する。

これらのセッション・クラスは全て一連の同じ API メソッドをサポートします。 その結果として、セッションを使用するアプリケーション・コードを修正することなしに、セッション・ストレージ・クラスを切り替えることが出来ます。

> 補足: カスタム・セッション・ストレージを使っているときに `$_SESSION` を通じてセッション・データにアクセスしたい場合は、 セッションが `yii\web\Session::open()` によって既に開始されていることを確認しなければなりません。 これは、カ

[12] http://redis.io/
[13] http://www.mongodb.org/

スタム・セッション・ストレージのハンドラは、この `open()` メソッドの中で登録されるからです。

補足: カスタム・セッション・ストレージを使うときは、セッションのガーベッジ・コレクタを明示的に構成する必要がある場合があります。 PHP のインストレーションによっては(例えば Debian では)、ガーベッジ・コレクタの蓋然性は 0 とされ、セッション・ファイルはクロン・ジョブでオフラインで消去することになっています。 このプロセスはあなたのカスタム・セッション・ストレージには当てはまりませんので、 `yii\web\Session::$GCProbability` に 0 でない値を設定して構成しなければなりません。

これらのコンポーネント・クラスの構成方法と使用方法については、それらの API ドキュメントを参照してください。 下記の例は、アプリケーションの構成情報において、データベース・テーブルをセッション・ストレージとして使うために `yii\web\DbSession` を構成する方法を示すものです。

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb',  // DB 接続のアプリケーション・コンポーネン
    ト 。デフォルトはID 'db。'
            // 'sessionTable' => 'my_session', // セッション・テーブル名。デ
    フォルトは 'session。'
        ],
    ],
];
```

セッション・データを保存するために、次のようなデータベース・テーブルを作成することも必要です。

```
CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

ここで 'BLOB' はあなたが選んだ DBMS の BLOB 型を指します。下記は人気のあるいくつかの DBMS で使用できる BLOB 型です。

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

補足: php.ini の `session.hash_function` の設定によっては、`id` カラムの長さを修正する必要があるかも知れません。 例えば、`session.hash_function=sha256` である場合は 40 の代りに 64 の長さを使わなければなりません。

別の方法として、次のマイグレーションを使ってこれを達成することも出来ます。

```
<?php

use yii\db\Migration;

class m170529_050554_create_table_session extends Migration
{
    public function up()
    {
        $this->createTable('{{%session}}', [
            'id' => $this->char(64)->notNull(),
            'expire' => $this->integer(),
            'data' => $this->binary()
        ]);
        $this->addPrimaryKey('pk-id', '{{%session}}', 'id');
    }

    public function down()
    {
        $this->dropTable('{{%session}}');
    }
}
```

フラッシュ・データ

フラッシュ・データは特殊な種類のセッション・データで、あるリクエストの中で設定されると、 次のリクエストの間においてのみ読み出すことが出来て、その後は自動的に削除されるものです。 フラッシュ・データが最もよく使われるのは、エンド・ユーザに一度だけ表示されるべきメッセージ、 例えば、ユーザのフォーム送信が成功した後に表示される確認メッセージなどを実装するときです。

session アプリケーション・コンポーネントによって、フラッシュ・データを設定し、アクセスすることが出来ます。例えば、

```
$session = Yii::$app->session;

// リクエスト #1
// "postDeleted" という名前のフラッシュ・メッセージを設定する
$session->setFlash('postDeleted', '投稿の削除に成功しました。');

// リクエスト #2
// "postDeleted" という名前のフラッシュ・メッセージを表示する
echo $session->getFlash('postDeleted');

// リクエスト #3
// フラッシュ・メッセージは自動的に削除されるので、$result は false になる
$result = $session->hasFlash('postDeleted');
```

通常のセッション・データと同様に、任意のデータをフラッシュ・データとして保存することが出来ます。

yii\web\Session::setFlash() を呼び出すと、同じ名前の既存のフラッシュ・データは上書きされます。 同じ名前の既存のメッセージに新しいフラッシュ・データを追加するためには、代りに yii\web\Session::addFlash() を使うことが出来ます。 例えば、

```
$session = Yii::$app->session;

// リクエスト #1
// "alerts" という名前の下にフラッシュ・メッセージを追加する
$session->addFlash('alerts', '投稿の削除に成功しました。');
$session->addFlash('alerts', '友達の追加に成功しました。');
$session->addFlash('alerts', 'あなたのレベルが上りました。');

// リクエスト #2
// $alerts は "alerts" という名前の下にあるフラッシュ・メッセージの配列となる
$alerts = $session->getFlash('alerts');
```

> 補足: 同じ名前のフラッシュ・データに対して、yii\web\Session::setFlash() と yii\web\Session::addFlash() を一緒に使わないようにしてください。 これは、後者のメソッドが、同じ名前のフラッシュ・データを追加できるように、フラッシュ・データを自動的に配列に変換するからです。 その結果、yii\web\Session::getFlash() を呼び出したとき、この二つのメソッドを呼び出した順序に従って、 あるときは配列を受け取り、あるときは文字列を受け取るということになります。
>
> ヒント: フラッシュ・メッセージを表示するためには、yii\bootstrap\Alert ウィジェットを次のように使用することが出来ます。
>
> ```
> echo Alert::widget([
>     'options' => ['class' => 'alert-info'],
>     'body' => Yii::$app->session->getFlash('postDeleted'),
> ]);
> ```

4.6.2 クッキー

Yii は個々のクッキーを yii\web\Cookie のオブジェクトとして表します。 yii\web\Request と yii\web\Response は、ともに、cookies という名前のプロパティによって、クッキーのコレクションを保持します。 前者のクッキー・コレクションはリクエストの中で送信されてきたクッキーを表し、 一方、後者のクッキー・コレクションは、これからユーザに送信されるクッキーを表します。

アプリケーションで、リクエストとレスポンスを直接に操作する部分は、コントローラです。 従って、クッキーの読み出しと送信はコントローラで実行されるべきです。

クッキーを読み出す

現在のリクエストに含まれるクッキーは、下記のコードを使って取得することが出来ます。

```
// "request" コンポーネントからクッキー・コレクション (yii\web\CookieCollection) を取得する。
$cookies = Yii::$app->request->cookies;

// "language" というクッキーの値を取得する。クッキーが存在しない場合は、デフォルト値として "en" を返す。
$language = $cookies->getValue('language', 'en');

// "language" というクッキーの値を取得する別の方法。
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// $cookies を配列のように使うことも出来る。
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// "language" というクッキーが在るかどうかチェックする。
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

クッキーを送信する

下記のコードを使って、クッキーをエンド・ユーザに送信することが出来ます。

```
// "response" コンポーネントからクッキー・コレクション (yii\web\CookieCollection) を取得する。
$cookies = Yii::$app->response->cookies;

// 送信されるレスポンスに新しいクッキーを追加する。
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// クッキーを削除する。
$cookies->remove('language');
// 次のようにしても同じ。
unset($cookies['language']);
```

yii\web\Cookie クラスは、上記の例で示されている name と value のプロパティ以外にも、domain や expire など、 他のプロパティを定義して、利用可能なクッキー情報の全てを完全に表しています。 クッキーを準備するときに必要に応じてこれらのプロパティを構成してから、レスポンスのクッキー・コレクションに追加することが出来ます。

クッキー検証

最後の二つの項で示されているように、request と response のコンポーネントを通じてクッキーを読んだり送信したりする場合には、 クッキーがクライアント・サイドで修正されるのを防止するクッキー検証という追加のセキュリティを享受することが出来ます。 これは、個々のクッキーにハッシュ文字列をサインとして追加することによって達成されます。アプリケーションは、サインを見て、クッキーがクライアント・サイドで修正されたかどうかを知ることが出来ます。 もし、修正されていれば、そのクッキーは request コンポーネントの クッキー・コレクションからはアクセスすることが出来なくなります。

> 補足: クッキー検証は値が修正されたクッキーの読み込みを防止するだけです。 検証に失敗した場合でも、$_COOKIE を通じてそのクッキーにアクセスすることは引き続いて可能です。これは、サードパーティのライブラリが、クッキー検証を含まない独自の方法でクッキーを操作することが出来るようにするするためです。

クッキー検証はデフォルトで有効になっています。 yii\web\Request::$enableCookieValidation プロパティを false に設定することによって無効にすることが出来ますが、無効にしないことを強く推奨します。

> 補足: $_COOKIE と setcookie() によって直接に 読み出し/送信 されるクッキーは検証されません。

クッキー検証を使用する場合は、前述のハッシュ文字列を生成するために使用される yii\web\Request::$cookieValidationKey を指定しなければなりません。 アプリケーションの構成情報で request コンポーネントを構成することによって、そうすることが出来ます。

```
return [
    'components' => [
        'request' => [
            'cookieValidationKey' => 'ここに秘密のキーを書く',
        ],
    ],
];
```

> 情報: cookieValidationKey は、あなたのアプリケーションにとって、決定的に重要なものです。 これは信頼する人にだけ教えるべきものです。バージョン・コントロール・システムに保存してはいけません。

4.6.3 セキュリティの設定

yii\web\Cookie と yii\web\Session の両者は下記のセキュリティ・フラグをサポートしています。

httpOnly

セキュリティを向上させるために、`yii\web\Cookie::$httpOnly` および `yii\web\Session::$cookieParams` の 'httponly' パラメータの デフォルト値は `true` に設定されています。 これによって、クライアント・サイド・スクリプトが保護されたクッキーにアクセスする危険が軽減されます (ブラウザがサポートしていれば)。 詳細については、httpOnly の wiki 記事[14] を読んでください。

secure

secure フラグの目的は、クッキーが平文で送信されることを防止することです。ブラウザが secure フラグをサポートしている場合、 リクエストが secure な接続 (TLS) によって送信される場合にのみクッキーがリクエストに含まれます。 詳細については Secure フラグの wiki 記事[15] を参照して下さい。

sameSite

Yii 2.0.21 以降、`yii\web\Cookie::$sameSite` 設定がサポートされています。これは PHP バージョン 7.3.0 以降を必要とします。 `sameSite` 設定の目的は CSRF (Cross-Site Request Forgery) 攻撃を防止することです。ブラウザが `sameSite` 設定をサポートしている場合、指定されたポリシー ('Lax' または 'Strict') に従うクッキーだけが送信されます。 詳細については SameSite の wiki 記事[16] を参照して下さい。 更なるセキュリティ強化のために、`sameSite` がサポートされていない PHP のバージョンで使われた場合には例外が投げられます。 この機能を PHP のバージョンに関わりなく使用する場合は、最初にバージョンをチェックして下さい。例えば、 'php [

```
'sameSite' => PHP_VERSION_ID >= 70300 ? yii\web\Cookie::SAME_SITE_LAX : null
    ,
```

] ' > Note: 今はまだ `sameSite` 設定をサポートしていないブラウザもありますので、 追加の CSRF 保護 を行うことを強く推奨します。

4.6.4 セッションに関する php.ini の設定

PHP マニュアル[17] で示されているように、`php.ini` にはセッションのセキュリティに関する重要な設定があります。 推奨される設定を必ず適用して下さい。特に、PHP インストールのデフォルトでは有効にされていない `session.use_strict_mode` を有効にして下さい。

[14] https://www.owasp.org/index.php/HttpOnly
[15] https://www.owasp.org/index.php/SecureFlag
[16] https://www.owasp.org/index.php/SameSite
[17] https://www.php.net/manual/ja/session.security.ini.php

4.7 エラー処理

Yii が内蔵している エラー・ハンドラ は、エラー処理を従来よりはるかに快適な経験にしてくれます。 具体的には、Yii のエラー・ハンドラはエラー処理をより良くするために、次のことを行います。

- 致命的でない全ての PHP エラー (警告や通知) は捕捉可能な例外に変換されます。
- 例外および致命的 PHP エラーは、デバッグ・モードでは、 詳細なコール・スタック情報とソース・コード行とともに表示されます。
- エラーを表示するために専用の コントローラ・アクション を使うことがサポートされています。
- さまざまなエラー・レスポンス形式をサポートしています。

エラー・ハンドラ はデフォルトで有効になっています。 アプリケーションの エントリ・スクリプト において、定数 `YII_ENABLE_ERROR_HANDLER` を `false` と定義することによって、これを無効にすることが出来ます。

4.7.1 エラー・ハンドラを使用する

エラー・ハンドラ は `errorHandler` という名前の アプリケーション・コンポーネント です。 次のように、アプリケーションの構成情報でこれをカスタマイズすることが出来ます。

```
return [
    'components' => [
        'errorHandler' => [
            'maxSourceLines' => 20,
        ],
    ],
];
```

上記の構成によって、例外ページで表示されるソース・コードの行数は最大で 20 までとなります。

既に述べたように、エラー・ハンドラは致命的でない全ての PHP エラーを捕捉可能な例外に変換します。 これは、次のようなコードを使って PHP エラーを処理することが出来るということを意味します。

```
use Yii;
use yii\base\ErrorException;

try {
    10/0;
} catch (ErrorException $e) {
    Yii::warning("0 による除算。");
}

// 実行を継続 ...
```

リクエストが無効または予期しないものであることをユーザに知らせるエラー・ページを表示したい場合は、 単に `yii\web\NotFoundHttpException`

のような HTTP 例外 を投げるだけで済ませることが出来ます。 そうすれば、エラー・ハンドラがレスポンスの HTTP ステータス・コードを正しく設定し、 適切なエラー・ビューを使ってエラー・メッセージを表示してくれます。

```
use yii\web\NotFoundHttpException;

throw new NotFoundHttpException();
```

4.7.2 エラー表示をカスタマイズする

エラー・ハンドラ は、定数 YII_DEBUG の値に従って、エラー表示を調整します。 YII_DEBUG が true である (デバッグ・モードである) 場合は、エラー・ハンドラは、デバッグがより容易になるように、 例外とともに、詳細なコール・スタック情報とソース・コード行を表示します。 そして、YII_DEBUG が false のときは、アプリケーションに関する公開できない情報の開示を防ぐために、エラー・メッセージだけが表示されます。

> 情報: 例外が yii\base\UserException の子孫である場合は、YII_DEBUG の値の如何にかかわらず、コール・スタックは表示されません。 これは、この種の例外はユーザの誤操作によって引き起こされるものであり、 開発者は何も修正する必要がないと考えられるからです。

デフォルトでは、エラー・ハンドラ は二つの ビュー を使ってエラーを表示します。

- @yii/views/errorHandler/error.php: エラーがコール・スタック情報なしで表示されるべき場合に使用されます。 YII_DEBUG が false の場合、これが表示される唯一のビューとなります。
- @yii/views/errorHandler/exception.php: エラーがコール・スタック情報と共に表示されるべき場合に使用されます。

エラー表示をカスタマイズするために、エラー・ハンドラの errorView および exceptionView プロパティを構成して、 自分自身のビューを使用することが出来ます。

エラー・アクションを使う

エラー表示をカスタマイズするためのもっと良い方法は、専用のエラー アクション を使うことです。 そうするためには、まず、errorHandler コンポーネントの errorAction プロパティを次のように構成します。

```
return [
    'components' => [
        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
    ]
];
```

errorAction プロパティは、アクションへの ルート を値として取ります。 上記の構成は、エラーをコール・スタック情報なしで表示する必要がある場合は、 site/error アクションが実行されるべきことを記述しています。

site/error アクションは次のようにして作成することが出来ます。

```
namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}
```

上記のコードは yii\web\ErrorAction クラスを使って error アクションを定義しています。 yii\web\ErrorAction クラスは error という名前のビューを使ってエラーをレンダリングします。

yii\web\ErrorAction を使う以外に、次のようにアクション・メソッドを使って error アクションを定義することも出来ます。

```
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

次に views/site/error.php に配置されるビュー・ファイルを作成しなければなりません。 エラー・アクションが yii\web\ErrorAction として定義されている場合は、このビュー・ファイルの中で次の変数にアクセスすることが出来ます。

- name: エラーの名前。
- message: エラー・メッセージ。
- exception: 例外オブジェクト。これを通じて、更に有用な情報、例えば、HTTP ステータス・コード、エラー・コード、 エラー・コール・スタックなどにアクセスすることが出来ます。

  情報: あなたが ベーシック・プロジェクト・テンプレート または アドバンスト・プロジェクト・テンプレート[18] を使って

[18] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/README.md

いる場合は、 エラー・アクションとエラー・ビューは、既にあなたのために定義されています。

補足: エラー・ハンドラの中でリダイレクトする必要がある場合は、次のようにしてください。

```
Yii::$app->getResponse()->redirect($url)->send();
return;
```

エラーのレスポンス形式をカスタマイズする

エラー・ハンドラは、レスポンス 形式の設定に従ってエラーを表示します。 レスポンス形式 が `html` である場合は、直前の項で説明したように、 エラー・ビューまたは例外ビューを使ってエラーを表示します。 その他のレスポンス形式の場合は、エラー・ハンドラは例外の配列表現を `yii\web\Response::$data` プロパティに代入し、 次に `data` プロパティを様々な形式に変換します。 例えば、レスポンス形式が `json` である場合は、次のようなレスポンスになります。

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": リクエストされたリソースは見つかりませんでした。"",
    "code": 0,
    "status": 404
}
```

エラーのレスポンス形式をカスタマイズするために、アプリケーションの構成情報の中で、 `response` コンポーネントの `beforeSend` イベントに反応するハンドラを構成することが出来ます。

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
```

```
    ],
];
```

上記のコードは、エラーのレスポンスを以下のようにフォーマットし直すものです。

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": リクエストされたリソースは見つかりませんでした。"",
        "code": 0,
        "status": 404
    }
}
```

4.8 ロギング

Yii は高度なカスタマイズ性と拡張性を持った強力なロギング・フレームワークを提供しています。 このフレームワークを使用すると、さまざまな種類のメッセージを記録し、それをフィルタして、ファイル、データベース、メールなど、 さまざまなターゲットに収集することが簡単に出来ます。

Yii のロギング・フレームワークを使うためには、下記のステップを踏みます。

- コードのさまざまな場所で ログ・メッセージ を記録する。
- ログ・メッセージをフィルタしてエクスポートするために、アプリケーションの構成情報で ログ・ターゲット を構成する。
- さまざまなターゲット (例えば Yii デバッガ) によって、フィルタされエクスポートされたログ・メッセージを調査する。

このセクションでは、主として最初の二つのステップについて説明します。

4.8.1 メッセージを記録する

ログ・メッセージを記録することは、次のログ記録メソッドのどれかを呼び出すだけの簡単なことです。

- `Yii::debug()`: コードの断片がどのように走ったかをトレースするメッセージを記録します。主として開発のために使用します。
- `Yii::info()`: 何らかの有用な情報を伝えるメッセージを記録します。

- Yii::warning(): 何か予期しないことが発生したことを示す警告メッセージを記録します。
- Yii::error(): 出来るだけ早急に調査すべき致命的なエラーを記録します。

これらのログ記録メソッドは、ログ・メッセージをさまざまな 重大性レベル と カテゴリ で記録するものです。 これらのメソッドは function ($message, $category = 'application') という関数シグニチャを共有しており、 $message は記録されるログ・メッセージを示し、$category はログ・メッセージのカテゴリを示します。 次のコード・サンプルは、トレース・メッセージをデフォルトのカテゴリである application の下に記録するものです。

```
Yii::debug('平均収益の計算を開始');
```

> 情報: ログ・メッセージは文字列でも、配列やオブジェクトのような複雑なデータでも構いません。 ログ・メッセージを適切に取り扱うのは ログ・ターゲット の責任です。 デフォルトでは、ログ・メッセージが文字列でない場合は、yii\helpers\VarDumper::export() が呼ばれて文字列に変換されることになります。

ログ・メッセージを上手に編成しフィルタするために、すべてのログ・メッセージにそれぞれ適切なカテゴリを指定することが推奨されます。 カテゴリに階層的な命名方法を採用すると、ログ・ターゲット がカテゴリに基づいてメッセージをフィルタすることが容易になります。 簡単でしかも効果的な命名方法は、カテゴリ名に PHP のマジック定数 __METHOD__ を使用することです。 これは、Yii フレームワークのコアコードでも使われている方法です。 例えば、

```
Yii::debug('平均収益の計算を開始', __METHOD__);
```

__METHOD__ という定数は、それが出現する場所のメソッド名 (完全修飾のクラス名が前置されます) として評価されます。 例えば、上記のコードが app\controllers\RevenueController::calculate というメソッドの中で呼ばれている場合は、 __METHOD__ は 'app\controllers\RevenueController::calculate' という文字列と同じになります。

> 情報: 上記で説明したメソッドは、実際には、ロガー・オブジェクト の log() メソッドへのショートカットです。 ロガー・オブジェクト は Yii::getLogger() という式でアクセス可能なシングルトンです。 ロガー・オブジェクトは、十分な量のメッセージが記録されたとき、または、アプリケーションが終了するときに、 メッセージ・ディスパッチャ を呼んで、登録された ログ・ターゲット に記録されたログ・メッセージを送信します。

4.8.2 ログ・ターゲット

ログ・ターゲットは yii\log\Target クラスまたはその子クラスのインスタンスです。 ログ・ターゲットは、ログ・メッセージを重大性レベルとカテゴリによってフィルタして、何らかの媒体にエクスポートします。例えば、データベース・ターゲット は、フィルタされたログ・メッセージをデータベース・テーブルにエクスポートし、 メール・ターゲットは、ログ・メッセージを指定されたメール・アドレスにエクスポートします。

一つのアプリケーションの中で複数のログ・ターゲットを登録することが出来ます。 そのためには、次のように、アプリケーションの構成情報の中で、log アプリケーション・コンポーネント によってログ・ターゲットを構成します。

```
return [
    // "log" コンポーネントはブートストラップ時にロードされなければならない
    'bootstrap' => ['log'],
    // "log" コンポーネントはタイムスタンプを持つメッセージを処理するので、正しいタイムスタンプを出力するように PHP タイムゾーンを設定
    'timeZone' => 'America/Los_Angeles',
    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error'],
                    'categories' => ['yii\db\*'],
                    'message' => [
                       'from' => ['log@example.com'],
                       'to' => ['admin@example.com', 'developer@example.com'],
                       'subject' => 'example.com で、データベースエラー発生',
                    ],
                ],
            ],
        ],
    ],
];
```

> 補足: log コンポーネントは、ログ・メッセージをターゲットに即座に送付することが出来るように、ブートストラップ 時にロードされなければなりません。 上記の例で bootstrap の配列に log がリストアップされているのは、そのためです。

上記のコードでは、二つのログ・ターゲットが yii\log\Dispatcher::$targets プロパティに登録されています。

- 最初のターゲットは、エラーと警告のメッセージを選択して、データベース・テーブルに保存します。
- 第二のターゲットは、名前が `yii\db\` で始まるカテゴリのエラー・メッセージを選んで、`admin@example.com` と `developer@example.com` の両方にメールで送信します。

Yii は下記のログ・ターゲットをあらかじめ内蔵しています。 その構成方法と使用方法を学ぶためには、これらのクラスの API ドキュメントを参照してください。

- `yii\log\DbTarget`: ログ・メッセージをデータベース・テーブルに保存する。
- `yii\log\EmailTarget`: ログ・メッセージを事前に指定されたメール・アドレスに送信する。
- `yii\log\FileTarget`: ログ・メッセージをファイルに保存する。
- `yii\log\SyslogTarget`: ログ・メッセージを PHP 関数 `syslog()` を呼んでシステム・ログに保存する。

以下では、全てのターゲットに共通する機能について説明します。

メッセージのフィルタリング

全てのログ・ターゲットについて、それぞれ、`levels` と `categories` のプロパティを構成して、 ターゲットが処理すべきメッセージの重要性レベルとカテゴリを指定することが出来ます。

`levels` プロパティは、次のレベルの一つまたは複数からなる配列を値として取ります。

- `error`: `Yii::error()` によって記録されたメッセージに対応。
- `warning`: `Yii::warning()` によって記録されたメッセージに対応。
- `info`: `Yii::info()` によって記録されたメッセージに対応。
- `trace`: `Yii::debug()` によって記録されたメッセージに対応。
- `profile`: `Yii::beginProfile()` と `Yii::endProfile()` によって記録されたメッセージに対応。 これについては、プロファイリング の項で詳細に説明します。

`levels` プロパティを指定しない場合は、 ターゲットが 全ての 重大性レベルのメッセージを処理することを意味します。

`categories` プロパティは、メッセージ・カテゴリの名前またはパターンからなる配列を値として取ります。 ターゲットは、カテゴリの名前がこの配列にあるか、または配列にあるパターンに合致する場合にだけ、メッセージを処理します。 カテゴリ・パターンというのは、最後にアスタリスク `*` を持つカテゴリ名接頭辞です。 カテゴリ名は、パターンと同じ接頭辞で始まる場合に、カテゴリ・パターンに合致します。 例えば、`yii\db\Command::execute` と `yii\db\Command::query` は、`yii\db\Command` クラスで記録されるログ・メッセージのためのカテゴリ名です。 そして、両者は共に `yii\db\*` というパターンに合致します。

`categories` プロパティを指定しない場合は、 ターゲットが 全ての カテゴリのメッセージを処理することを意味します。

処理するカテゴリを categories プロパティで指定する以外に、 処理から除外するカテゴリを except プロパティによって指定することも可能です。 カテゴリの名前がこの配列にあるか、または配列にあるパターンに合致する場合は、メッセージはターゲットによって処理されません。

次のターゲットの構成は、ターゲットが、yii\db* または yii\web\HttpException:* に合致するカテゴリ名を持つエラーおよび警告のメッセージだけを処理すべきこと、 ただし、yii\web\HttpException:404 は除外すべきことを指定するものです。

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

> 情報: HTTP 例外が エラー・ハンドラ によって捕捉されたときは、 yii\web\HttpException:ErrorCode という書式のカテゴリ名でエラー・メッセージがログに記録されます。 例えば、yii\web\NotFoundHttpException は、yii\web\HttpException:404 というカテゴリのエラー・メッセージを発生させます。

メッセージの書式設定

ログ・ターゲットはフィルタされたログ・メッセージを一定の書式でエクスポートします。 例えば、yii\log\FileTarget クラスのログ・ターゲットをインストールした場合は、 runtime/log/app.log ファイルに、下記と同様なログ・メッセージが書き込まれます。

```
2014-10-04 18:10:15 [::1][][-][trace][yii\base\Module::getModule] Loading
    module: debug
```

デフォルトでは、ログ・メッセージは yii\log\Target::formatMessage() によって、下記のように書式設定されます。

```
タイムスタンプ
[IP アドレスユーザ][ IDセッション][ ID重要性レベルカテゴリ][][] メッセージテ
    キスト
```

この書式は、yii\log\Target::$prefix プロパティを構成することでカスタマイズすることが出来ます。 yii\log\Target::$prefix プロパティは、カスタマイズされたメッセージ前置情報を返す PHP コーラブルを値として取ります。 例えば、次のコードは、ログ・ターゲットが全てのログ・メッセージの前にカレント・ユーザの ID を置くようにさせるもので

す(IP アドレスとセッション ID はプライバシー上の理由から削除されています)。

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') : null
    ;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

メッセージ前置情報以外にも、ログ・ターゲットは、一群のログ・メッセージごとに一定のコンテキスト情報を追加します。 デフォルトでは、その情報には、次のグローバル PHP 変数、すなわち、$_GET、$_POST、$_FILES、$_COOKIE、$_SESSION および $_SERVER の値が含まれます。 ログ・ターゲットに含ませたいグローバル変数の名前を yii\log\Target::$logVars プロパティに設定することによって、 この動作を調整することが出来ます。 例えば、次のログ・ターゲットの構成は、$_SERVER の値だけをログ・メッセージに追加するように指定するものです。

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

logVars を空の配列として構成して、コンテキスト情報をまったく含ませないようにすることも出来ます。 あるいは、また、コンテキスト情報の提供方法を自分で実装したい場合は、 yii\log\Target::getContextMessage() メソッドをオーバーライドすることも出来ます。

ログに出力したくない機密情報 (例えば、パスワードやアクセス・トークン) を含んでいるリクエストのフィールドについては、maskVars プロパティを追加で構成することが出来ます。 デフォルトでは、$_SERVER[HTTP_AUTHORIZATION]、$_SERVER[PHP_AUTH_USER]、$_SERVER[PHP_AUTH_PW] のリクエスト・パラメータが *** でマスクされまが、 自分自身で設定することも出来ます。例えば、

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
    'maskVars' => ['_SERVER.HTTP_X_PASSWORD']
]
```

メッセージのトレース・レベル

開発段階では、各ログ・メッセージがどこから来ているかを知りたい場合がよくあります。 これは、次のように、log コンポーネントの traceLevel プロパティを構成することによって達成できます。

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

上記のアプリケーションの構成は、traceLevel を YII_DEBUG が on のときは 3、YII_DEBUG が off のときは 0 に設定します。 これは、YII_DEBUG が on のときは、各ログ・メッセージに対して、ログ・メッセージが記録されたときのコール・スタックを最大 3 レベルまで追加し、 YII_DEBUG が 0 のときはコール・スタックを含めない、 ということを意味します。

> 情報: コール・スタック情報の取得は軽微な処理ではありません。 従って、この機能は開発時またはアプリケーションをデバッグするときに限って使用するべきです。

メッセージの吐き出しとエクスポート

既に述べたように、ログ・メッセージは ロガー・オブジェクト によって配列の中に保持されます。 この配列のメモリ消費を制限するために、この配列に一定数のログ・メッセージが蓄積されるたびに、 ロガーは記録されたメッセージを ログ・ターゲット に吐き出します。 この数は、log コンポーネントの flushInterval プロパティを構成することによってカスタマイズすることが出来ます。

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,   // デフォルトは 1000
            'targets' => [...],
        ],
    ],
];
```

> 情報: メッセージの吐き出しは、アプリケーションの終了時にも実行されます。これによって、ログ・ターゲットが完全なログ・メッセージを受け取ることが保証されます。

ロガー・オブジェクト が ログ・ターゲット にログ・メッセージを吐き出しても、ログ・メッセージはただちにはエクスポートされません。 そうではなく、ログ・ターゲットが一定数のフィルタされたメッセージを蓄積して初めて、メッセージのエクスポートが発生します。 この数は、下記のように、個々の ログ・ターゲット の exportInterval プロパティを構成することによってカスタマイズすることが出来ます。

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,  // デフォルトは 1000
]
```

デフォルトの状態では、吐き出しとエクスポートの間隔の設定のために、Yii::debug() やその他のログ記録メソッドを呼んでも、 ただちには、ログ・メッセージはログ・ターゲットに出現しません。 このことは、長時間にわたって走るコンソール・アプリケーションでは、問題になる場合もあります。 各ログ・メッセージがただちにログ・ターゲットに出現するようにするためには、下記のように、flushInterval と exportInterval の両方を 1 に設定しなければなりません。

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

> 補足: 頻繁なメッセージの吐き出しとエクスポートはアプリケーションのパフォーマンスを低下させます。

ログ・ターゲットの 有効/無効 を切り替える

enabled プロパティを構成することによって、ログ・ターゲットを有効にしたり無効にしたりすることが出来ます。 この切り替えは、ログ・ターゲットのコンフィギュレーションでも出来ますが、コードの中で次の PHP 文を使っても出来ます。

```
Yii::$app->log->targets['file']->enabled = false;
```

上記のコードでは、ターゲットが file という名前であることが必要とされています。 下記のように、targets の配列で文字列のキーを使ってターゲットの名前を指定して下さい。

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
```

```
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

バージョン 2.0.13 以降は、enabled を設定するのに、 ログ・ターゲットを有効にすべきか否かの動的な条件を定義するコーラブルを指定することが可能です。 yii\log\Target::setEnabled() のドキュメントに一例がありますので参照して下さい。

新しいターゲットを作る

新しいログ・ターゲット・クラスを作ることは非常に簡単です。 必要なことは、主として、yii\log\Target::$messages 配列の中身を指定された媒体に送出する yii\log\Target::export() メソッドを実装することです。 各メッセージに書式を設定するためには、yii\log\Target::formatMessage() を呼ぶことが出来ます。 詳細については、Yii リリースに含まれているログ・ターゲット・クラスのどれか一つを参照してください。

> ヒント: あなた自身のロガーを書く代りに、PSR ログ・ターゲット・エクステンション[19] によって、 Monolog[20] のような PSR-3 互換ロガーのどれかを使ってみるのも良いでしょう。

4.8.3 パフォーマンス・プロファイリング

パフォーマンス・プロファイリングは、特定のコード・ブロックに要した時間を測定してパフォーマンスのボトルネックになっている所を見つけ出すために使われる、 特殊なタイプのメッセージ・ロギングです。 例えば、yii\db\Command クラスは、各 DB クエリに要した時間を知るために、パフォーマンス・プロファイリングを使用しています。

パフォーマンス・プロファイリングを使用するためには、最初に、プロファイリングが必要なコード・ブロックを特定します。 そして、各コード・ブロックを次のように囲みます。

```
\Yii::beginProfile('myBenchmark');

... プロファイリングされるコード・ブロック ...

\Yii::endProfile('myBenchmark');
```

ここで myBenchmark はコード・ブロックを特定するユニークなトークンを表します。 後でプロファイリング結果を検査するときに、このトークン

[19] https://github.com/samdark/yii2-psr-log-target
[20] https://github.com/Seldaek/monolog

を使って、対応するコード・ブロックによって消費された時間を調べます。

beginProfile と endProfile のペアが適正な入れ子になっていることを確認することが非常に重要なことです。 例えば、

```
\Yii::beginProfile('block1');

    // プロファイリングされる何らかのコード

    \Yii::beginProfile('block2');
        // プロファイリングされる別のコード
    \Yii::endProfile('block2');

\Yii::endProfile('block1');
```

\Yii::endProfile('block1') を忘れたり、\Yii::endProfile('block1') と \Yii::endProfile('block2') の順序を入れ替えたりすると、 パフォーマンス・プロファイリングは機能しません。

プロファイルされるコード・ブロックの全てについて、おのおの、重大性レベルが profile であるログ・メッセージが記録されます。 そのようなメッセージを集めてエクスポートする ログ・ターゲット を構成してください。 Yii デバッガ が、プロファイリング結果を表示するパフォーマンス・プロファイリング・パネルを内蔵しています。

Chapter 5

鍵となる概念

5.1 コンポーネント

コンポーネントは、Yiiアプリケーションの主要な構成ブロックです。コンポーネントは yii\base\Component 、 またはその派生クラスのインスタンスです。コンポーネントが他のクラスに提供する主な機能は次の 3 つです:

- プロパティ
- イベント
- ビヘイビア

個々にでも、組み合わせでも、これらの機能は Yii のクラスのカスタマイズ性と使いやすさをとても高めてくれます。 たとえば、ユーザ・インタフェイス·コンポーネントである yii\jui\DatePicker は、 ビュー で次のように使用して、対話型の日付選択 UI を生成することができます:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ja',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

クラスが yii\base\Component を継承しているおかげで、ウィジェットのプロパティは簡単に記述できます。

コンポーネントは非常に強力ですが、 イベント と ビヘイビア をサポートするため、 余分にメモリと CPU 時間を要し、通常のオブジェクトよりも少し重くなります。 あなたのコンポーネントがこれら2つの機能を必要としない場合、yii\base\Component の代わりに、 yii\base\BaseObject からコンポーネント・クラスを派生することを検討してもよいでしょう。 そうすることで、あなたのコンポーネントは、 プロパティ のサポートが維持されたまま、通常の PHP オブジェクトのように

効率的になります。

yii\base\Component または yii\base\BaseObject からクラスを派生するときは、 次の規約に従うことが推奨されます:

- コンストラクタをオーバーライドする場合は、コンストラクタの 最後の パラメータとして $config パラメータを指定し、 親のコンストラクタにこのパラメータを渡すこと。
- 自分がオーバーライドしたコンストラクタの 最後で 、必ず親クラスのコンストラクタを呼び出すこと。
- yii\base\BaseObject::init() メソッドをオーバーライドする場合は、自分の init() メソッドの 最初に 、必ず init() の親実装を呼び出すようにすること。

例えば、

```
<?php

namespace yii\components\MyClass;

use yii\base\BaseObject;

class MyClass extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... 構成前の初期化

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... 構成後の初期化
    }
}
```

このガイドラインに従うことで、あなたのコンポーネントは生成時に コンフィグ可能 になります。例えば、

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// あるいは、また
$component = \Yii::createObject([
    'class' => MyClass::className(),
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

補足: Yii::createObject() を呼び出すアプローチは複雑に

見えますが、より強力です。 というのも、それが 依存性注入コンテナ 上に実装されているからです。

yii\base\BaseObject クラスには、次のオブジェクト・ライフサイクルが適用されます:

1. コンストラクタ内の事前初期化。ここでデフォルトのプロパティ値を設定することができます。
2. $config によるオブジェクトの構成。構成情報は、コンストラクタ内で設定されたデフォルト値を上書きすることがあります。
3. init() 内の事後初期化。サニティ・チェックやプロパティの正規化を行いたいときは、このメソッドをオーバーライドします。
4. オブジェクトのメソッド呼び出し。

最初の 3 つのステップは、すべて、オブジェクトのコンストラクタ内で発生します。これは、あなたがクラス・インスタンス (つまり、オブジェクト) を得たときには、 すでにそのオブジェクトが適切な、信頼性の高い状態に初期化されていることを意味します。

5.2 プロパティ

PHPでは、クラスのメンバ変数は プロパティ とも呼ばれます。 これらの変数は、クラス定義の一部で、クラスのインスタンスの状態を表すために(すなわち、クラスのあるインスタンスを別のものと区別するために)使用されます。 現実には、特別な方法でこのプロパティの読み書きを扱いたい場合がよくあります。 たとえば、label プロパティに割り当てられる文字列が常にトリミングされるようにしたい、など。 その仕事を成し遂げるために、あなたは次のようなコードを使おうと思えば使うことも出来ます。

```
$object->label = trim($label);
```

上記のコードの欠点は、label プロパティを設定するすべてのコードで、trim() を呼び出す必要があるということです。 もし将来的に、label プロパティに、最初の文字を大文字にしなければならない、といった新たな要件が発生したら、 label に値を代入するすべてのコードを変更しなければなりません。 コードの繰り返しはバグを誘発するので、可能な限り避けたいところです。

この問題を解決するために、Yii は *getter* メソッドと *setter* メソッドをベースにしたプロパティ定義をサポートする、 yii\base\BaseObject 基底クラスを提供しています。 クラスがその機能を必要とするなら、yii\base\BaseObject またはその子クラスを継承しましょう。

> 補足: Yiiのフレームワークのほぼすべてのコア・クラスは、yii\base\BaseObject またはその子クラスを継承しています。 これは、コア・クラスに getter または setter があれば、それをプロパティのように使用できることを意味します。

getter メソッドは、名前が get で始まるメソッドで、setter メソッドは、set で始まるメソッドです。 get または set 接頭辞の後の名前で、プロパティ名を定義します。次のコードに示すように、たとえば、getLabel() という getter と setLabel() という setter は、 label という名前のプロパティを定義します:

```
namespace app\components;

use yii\base\BaseObject;

class Foo extends BaseObject
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

    public function setLabel($value)
    {
        $this->_label = trim($value);
    }
}
```

詳しく言うと、getter および setter メソッドは、この場合には、内部的に _label と名付けられた private な属性を参照する label というプロパティを作っています。

getter と setter によって定義されたプロパティは、クラスのメンバ変数のように使用することができます。主な違いは、 それらのプロパティが読み取りアクセスされるときは、対応する getter メソッドが呼び出されることであり、プロパティに値が割り当てられるときには、 対応する setter メソッドが呼び出されるということです。例えば、

```
// $label = $object->getLabel(); と同じ
$label = $object->label;
```

```
// $object->setLabel('abc'); と同じ
$object->label = 'abc';
```

setter なしの getter で定義されたプロパティは、 読み取り専用 です。そのようなプロパティに値を代入しようとすると、 InvalidCallException が発生します。同様に、getter なしの setter で定義されたプロパティは、書き込み専用 で、そのようなプロパティを読み取りしようとしても、例外が発生します。 書き込み専用のプロパティを持つのは一般的ではありませんが。

getter と setter で定義されたプロパティには、いくつかの特別なルールと制限があります:

- この種のプロパティでは、名前の 大文字と小文字を区別しません 。たとえば、 `$object->label` と `$object->Label` は同じです。 これは、PHP のメソッド名が大文字と小文字を区別しないためです。
- この種のプロパティの名前と、クラスのメンバ変数の名前とが同じである場合、後者が優先されます。 たとえば、上記の `Foo` クラスがメンバ変数 `label` を持っている場合は、`$object->label = 'abc'` という代入は メンバ変数の `label` に作用することになります。その行から `setLabel()` setter メソッドは呼び出されません。
- これらのプロパティは可視性をサポートしていません。プロパティが public、protected、private であるかどうかを、getter または setter メソッドの定義によって決めることは出来ません。
- プロパティは、 静的でない getter および setter によってのみ定義することが出来ます。静的なメソッドは同様には扱われません。
- 通常の `property_exists()` の呼び出しでは、マジック・プロパティが存在するかどうかを知ることは出来ません。 それぞれ、`canGetProperty()` または `canSetProperty()` を呼び出さなければなりません。

このガイドの冒頭で説明した問題に戻ると、 `label` に値が代入されているあらゆる箇所で `trim()` を呼ぶのではなく、 もう `setLabel()` という setter の内部だけで `trim()` を呼べば済むのです。 さらに、新しい要求でラベルの先頭を大文字にする必要が発生しても、他のいっさいのコードに触れることなく、 すぐに `setLabel()` メソッドを変更することができます。一箇所の変更は、すべての `label` への代入に普遍的に作用します。

5.3 イベント

イベントを使うと、既存のコードの特定の実行ポイントに、カスタム・コードを挿入することができます。イベントにカスタム・コードをアタッチすると、 イベントがトリガされたときにコードが自動的に実行されます。たとえば、メーラ・オブジェクトがメッセージを正しく送信できたとき、 `messageSent` イベントをトリガするとします。もしメッセージの送信がうまく行ったことを知りたければ、単に `messageSent` イベントにトラッキング・コードを付与するだけで、それが可能になります。

Yiiはイベントをサポートするために、 `yii\base\Component` と呼ばれる基底クラスを導入してします。クラスがイベントをトリガする必要がある場合は、 `yii\base\Component` もしくはその子クラスを継承する必要があります。

5.3.1 イベント・ハンドラ

イベント・ハンドラとは、アタッチされたイベントがトリガされたとき

に実行される PHP コールバック[1] です。次のコールバックのいずれも使用可能です:

- 文字列で指定されたグローバル PHP 関数 (括弧を除く)、例えば `'trim'`。
- オブジェクトとメソッド名文字列の配列で指定された、オブジェクトのメソッド (括弧を除く)、例えば `[$object, 'methodName']`。
- クラス名文字列とメソッド名文字列の配列で指定された、静的なクラス・メソッド (括弧を除く)、例えば `['ClassName', 'methodName']`。
- 無名関数、例えば `function ($event) { ... }`。

イベント・ハンドラのシグネチャはこのようになります:

```
function ($event) {
    // $event は yii\base\Event またはその子クラスのオブジェクト
}
```

`$event` パラメータを介して、イベント・ハンドラは発生したイベントに関して次の情報を得ることができます:

- イベント名
- イベント送信元: `trigger()` メソッドが呼ばれたオブジェクト
- カスタム・データ: イベント・ハンドラをアタッチするときに提供されたデータ (次の項で説明します)

5.3.2 イベント・ハンドラをアタッチする

イベント・ハンドラは `yii\base\Component::on()` を呼び出すことでアタッチできます。たとえば:

```
$foo = new Foo;

// このハンドラはグローバル関数です
$foo->on(Foo::EVENT_HELLO, 'function_name');

// このハンドラはオブジェクトのメソッドです
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// このハンドラは静的なクラスメソッドです
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// このハンドラは無名関数です
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // イベント処理ロジック
});
```

また、 構成情報 を通じてイベント・ハンドラをアタッチすることもできます。詳細については 構成情報 の章を参照してください。

イベント・ハンドラをアタッチするとき、 `yii\base\Component::on()` の3番目のパラメータとして、付加的なデータを提供することができ

[1] https://secure.php.net/manual/ja/language.types.callable.php

ます。 そのデータは、イベントがトリガされてハンドラが呼び出されるときに、ハンドラ内で利用きます。たとえば:

```
// 次のコードはイベントがトリガされたとき "abc" を表示します
// "on" に番目の引数として渡されたデータを3 $event->data が保持しているからです
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

5.3.3 イベント・ハンドラの順序

ひとつのイベントには、ひとつだけでなく複数のハンドラをアタッチすることができます。イベントがトリガされると、アタッチされたハンドラは、 それらがイベントにアタッチされた順序どおりに呼び出されます。あるハンドラがその後に続くハンドラの呼び出しを停止する必要がある場合は、 $event パラメータの yii\base\Event::$handled プロパティを true に設定します:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

デフォルトでは、新たに接続されたハンドラは、イベントの既存のハンドラのキューに追加されます。その結果、 イベントがトリガされたとき、そのハンドラは一番最後に呼び出されます。もし、そのハンドラが最初に呼び出されるよう、 ハンドラのキューの先頭に新しいハンドラを挿入したい場合は、yii\base\Component::on() を呼び出すとき、4番目のパラメータ $append に false を渡します:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4 イベントをトリガする

イベントは、 yii\base\Component::trigger() メソッドを呼び出すことでトリガされます。このメソッドには イベント名 が必須で、 オプションで、イベント・ハンドラに渡されるパラメータを記述したイベント・オブジェクトを渡すこともできます。たとえば:

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class Foo extends Component
{
```

```
    const EVENT_HELLO = 'hello';

    public function bar()
    {
        $this->trigger(self::EVENT_HELLO);
    }
}
```

上記のコードでは、すべての `bar()` の呼び出しは、 `hello` という名前のイベントをトリガします。

> ヒント: イベント名を表すときはクラス定数を使用することをお勧めします。上記の例では、定数 `EVENT_HELLO` は `hello` イベントを表しています。このアプローチには 3 つの利点があります。まず、タイプミスを防ぐことができます。次に、IDE の自動補完サポートでイベントを 認識できるようになります。第 3 に、クラスでどんなイベントがサポートされているかを表したいとき、定数の宣言をチェックするだけで済みます。

イベントをトリガするとき、イベント・ハンドラに追加情報を渡したいことがあります。たとえば、メーラーが `messageSent` イベントのハンドラに メッセージ情報を渡して、ハンドラが送信されたメッセージの詳細を知ることができるようにしたいかもしれません。 これを行うために、`yii\base\Component::trigger()` メソッドの2番目のパラメータとして、イベント・オブジェクトを与えることができます。 イベント・オブジェクトは `yii\base\Event` クラスあるいはその子クラスのインスタンスでなければなりません。 たとえば:

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class MessageEvent extends Event
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ... $message 送信 ...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
```

```
}
```

yii\base\Component::trigger() メソッドが呼び出されたとき、この名前を付けられたイベントに アタッチされたハンドラがすべて呼び出されます。

5.3.5 イベント・ハンドラをデタッチする

イベントからハンドラを取り外すには、 yii\base\Component::off() メソッドを呼び出します。たとえば:

```
// このハンドラはグローバル関数です
$foo->off(Foo::EVENT_HELLO, 'function_name');

// このハンドラはオブジェクトのメソッドです
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// このハンドラは静的なクラスメソッドです
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// このハンドラは無名関数です
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

一般的には、イベントにアタッチされたときどこかに保存してある場合を除き、無名関数を取り外そうとはしないでください。 上記の例は、無名関数は変数 $anonymousFunction として保存されていたものとしています。

イベントから すべて のハンドラを取り外すには、単純に、第 2 パラメータを指定せずに yii\base\Component::off() を呼び出します。

```
$foo->off(Foo::EVENT_HELLO);
```

5.3.6 クラス・レベル・イベント・ハンドラ

ここまでの項では、インスタンス・レベル でのイベントにハンドラをアタッチする方法を説明してきました。 場合によっては、特定のインスタンスだけではなく、 クラスのすべてのインスタンスがトリガしたイベントに応答したいことがあります。 すべてのインスタンスにイベント・ハンドラをアタッチする代わりに、静的メソッド yii\base\Event::on() を呼び出すことで、 クラス・レベル でハンドラをアタッチすることができます。

たとえば、アクティブ・レコード オブジェクトは、データベースに新しいレコードを挿入するたびに、 EVENT_AFTER_INSERT イベントをトリガします。 すべての アクティブ・レコード オブジェクトによって行われる挿入を追跡するには、次のコードが使えます :

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;
```

```
Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT,
    function ($event) {
    Yii::debug(get_class($event->sender) . ' が挿入されました');
});
```

ActiveRecord またはその子クラスのいずれかが、 EVENT_AFTER_INSERT をトリガするといつでも、このイベント・ハンドラが呼び出されます。ハンドラの中では、 $event->sender を通して、 イベントをトリガしたオブジェクトを取得することができます。

オブジェクトがイベントをトリガするときは、最初にインスタンス・レベルのハンドラを呼び出し、続いてクラス・レベルのハンドラとなります。

静的メソッド yii\base\Event::trigger() を呼び出すことによって、 クラス・レベル でイベントをトリガすることができます。 クラス・レベルでのイベントは、特定のオブジェクトに関連付けられていません。そのため、これはクラス・レベルのイベント・ハンドラだけを 呼び出します。たとえば:

```
use yii\base\Event;

Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // "null" を表示
});

Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

この場合、$event->sender は、オブジェクト・インスタンスではなく、null になることに注意してください。

> 補足: クラス・レベルのハンドラは、そのクラスのあらゆるインスタンス、またはあらゆる子クラスのインスタンスがトリガしたイベントに応答 してしまうため、よく注意して使わなければなりません。 yii\base\BaseObject のように、クラスが低レベルの基底クラスの場合は特にそうです。

クラス・レベルのイベント・ハンドラを取り外すときは、 yii\base\Event::off() を呼び出します。たとえば:

```
// $handler をデタッチ
Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);

// Foo::EVENT_HELLO のすべてのハンドラをデタッチ
Event::off(Foo::className(), Foo::EVENT_HELLO);
```

5.3.7 インタフェイスを使うイベント

イベントを扱うためには、もっと抽象的な方法もあります。 特定のイベントのために専用のインタフェイスを作っておき、必要な場合にいろいろなクラスでそれを実装するのです。

例えば、次のようなインタフェイスを作ります。

```
namespace app\interfaces;

interface DanceEventInterface
{
    const EVENT_DANCE = 'dance';
}
```

そして、それを実装する二つのクラスを作ります。

```
class Dog extends Component implements DanceEventInterface
{
    public function meetBuddy()
    {
        echo "ワン!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}

class Developer extends Component implements DanceEventInterface
{
    public function testsPassed()
    {
        echo "よっしゃ!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}
```

これらのクラスのどれかによってトリガされた `EVENT_DANCE` を扱うためには、インタフェイス・クラスの名前を最初の引数にして `Event::on()` を呼びます。

```
Event::on('app\interfaces\DanceEventInterface', DanceEventInterface::
    EVENT_DANCE, function ($event) {
    Yii::debug(get_class($event->sender) . ' が躍り上がって喜んだ。'); // 犬
    または開発者が躍り上がって喜んだことをログに記録。
});
```

これらのクラスのイベントをトリガすることも出来ます。

```
// trigger event for Dog class
Event::trigger(Dog::className(), DanceEventInterface::EVENT_DANCE);

// trigger event for Developer class
Event::trigger(Developer::className(), DanceEventInterface::EVENT_DANCE);
```

ただし、このインタフェイスを実装する全クラスのイベントをトリガすることは出来ない、ということに注意して下さい。

```
// これは動かない。このインタフェイスを実装するクラスのイベントはトリガされな
    い。
Event::trigger('app\interfaces\DanceEventInterface', DanceEventInterface::
    EVENT_DANCE);
```

イベント・ハンドラをデタッチするためには、`Event::off()` を呼びます。例えば、

```
// $handler をデタッチ
Event::off('app\interfaces\DanceEventInterface', DanceEventInterface::
    EVENT_DANCE, $handler);

// DanceEventInterface::EVENT_DANCE の全てのハンドラをデタッチ
Event::off('app\interfaces\DanceEventInterface', DanceEventInterface::
    EVENT_DANCE);
```

5.3.8 グローバル・イベント

Yiiは、いわゆる グローバル・イベント をサポートしています。これは、実際には、上記のイベント・メカニズムに基づいたトリックです。 グローバル・イベントは、 アプリケーション インスタンス自身などの、グローバルにアクセス可能なシングルトンを必要とします。

グローバル・イベントを作成するには、イベント送信者は、送信者の自前の `trigger()` メソッドを呼び出す代わりに、シングルトンの `trigger()` メソッドを呼び出してイベントをトリガします。同じく、イベント・ハンドラも、シングルトンのイベントにアタッチされます。たとえば:

```
use Yii;
use yii\base\Event;
use app\components\Foo;

Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);  // "app\components\Foo" を表示
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

グローバル・イベントを使用する利点は、オブジェクトによってトリガされるイベント・ハンドラを設けたいとき、オブジェクトがなくてもいい ということです。その代わりに、ハンドラのアタッチとイベントのトリガはともに、(アプリケーションのインスタンスなど) シングルトンを介して行われます。

しかし、グローバル・イベントの名前空間はあらゆる部分から共有されているので、ある種の名前空間 ("frontend.mail.sent"、"backend.mail.sent" など) を導入するというような、賢いグローバル・イベントの名前付けをする必要があります。

5.3.9 ワイルドカード・イベント

2.0.14 以降は、ワイルドカード・パターンに一致する複数のイベントに対してイベント・ハンドラを設定することが出来ます。 例えば、

```
use Yii;

$foo = new Foo();

$foo->on('foo.event.*', function ($event) {
```

```
    // 'foo.event.' で始まる全てのイベントに対してトリガされる
    Yii::debug('trigger event: ' . $event->name);
});
```

クラス・レベル・イベントに対してもワイルドカード・パターンを用いることが出来ます。例えば、

```
use yii\base\Event;
use Yii;

Event::on('app\models\*', 'before*', function ($event) {
    // 名前空間 'app\models' の全てのクラスで、名前が 'before' で始まる全てのイベントに対してトリガされる
    Yii::debug('trigger event: ' . $event->name . ' for class: ' . get_class($event->sender));
});
```

これを利用すると、以下のコードを使って、全てのアプリケーション・イベントを一つのハンドラでキャッチすることが出来ます。

```
use yii\base\Event;
use Yii;

Event::on('*', '*', function ($event) {
    // 全てのクラスの全てのイベントに対してトリガされる
    Yii::debug('trigger event: ' . $event->name);
});
```

> 補足: イベント・ハンドラにワイルドカードを使用する設定は、アプリケーションの性能を低下させ得ます。 可能であれば避ける方が良いでしょう。

ワイルドカード・パターンで指定されたイベント・ハンドラをデタッチするためには、`yii\base\Component::off()` または `yii\base\Event::off()` の呼び出しにおいて、 同じパターンを使用しなければなりません。 イベント・ハンドラをデタッチする際にワイルドカードを指定すると、そのワイルドカードで指定されたハンドラだけがデタッチされることに留意して下さい。 通常のイベント名でアタッチされたハンドラは、パターンに合致する場合であっても、デタッチされません。例えば、

```
use Yii;

$foo = new Foo();

// 通常のハンドラをアタッチする
$foo->on('event.hello', function ($event) {
    echo 'direct-handler'
});

// ワイルドカード・ハンドラをアタッチする
$foo->on('*', function ($event) {
    echo 'wildcard-handler'
```

```
});

// ワイルドカード・ハンドラをデタッチする!
$foo->off('*');

$foo->trigger('event.hello'); // 出力: 'direct-handler'
```

5.4 ビヘイビア

ビヘイビアは yii\base\Behavior またその子クラスのインスタンスです。ビヘイビアは ミックスイン[2] としても知られ、既存の component クラスの 機能を、クラスの継承を変更せずに拡張することができます。コンポーネントにビヘイビアをアタッチすると、その コンポーネントにはビヘイビアのメソッドとプロパティが “注入” され、それらのメソッドとプロパティは、 コンポーネント・クラス自体に定義されているかのようにアクセスできるようになります。また、ビヘイビアは、 コンポーネントによってトリガされた イベント に応答することができるので、ビヘイビアでコンポーネントの通常のコード実行をカスタマイズすることができます。

5.4.1 ビヘイビアを定義する

ビヘイビアを定義するには、 yii\base\Behavior あるいは子クラスを継承するクラスを作成します。たとえば:

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

    public function getProp2()
    {
        return $this->_prop2;
    }

    public function setProp2($value)
    {
        $this->_prop2 = $value;
    }

    public function foo()
    {
```

[2] http://en.wikipedia.org/wiki/Mixin

```
        // ...
    }
}
```

上のコードは、prop1、prop2 という2つのプロパティと foo() というメソッドを持つ app\components\MyBehavior ビヘイビア・クラスを定義します。prop2 プロパティは、 getProp2() getter メソッドと setProp2() setter メソッドで定義されることに着目してください。 yii\base\Behavior は yii\base\BaseObject を継承しているので、getter と setter による プロパティ 定義をサポートします。

このクラスはビヘイビアなので、コンポーネントにアタッチされると、そのコンポーネントは prop1 と prop2 のプロパティと foo() メソッドを持つようになります。

> ヒント: ビヘイビア内から、yii\base\Behavior::$owner プロパティを介して、ビヘイビアをアタッチしたコンポーネントにアクセスすることができます。

> 補足: ビヘイビアの yii\base\Behavior::__get() および/または yii\base\Behavior::__set() メソッドをオーバーライドする場合は、 同時に yii\base\Behavior::canGetProperty() および/または yii\base\Behavior::canSetProperty() もオーバーライドする必要があります。

5.4.2 コンポーネントのイベントを処理する

ビヘイビアが、アタッチされたコンポーネントがトリガするイベントに応答する必要がある場合は、 yii\base\Behavior::events() メソッドをオーバーライドしなければなりません。たとえば:

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

yii\base\Behavior::events() メソッドは、イベントとそれに対応するハンドラのリストを返します。 上の例では EVENT_BEFORE_VALIDATE イベントがあること、 そのハンドラ定義である beforeValidate() を宣言しています。イベント・ハンドラを指定するときは、以下の表記方法が使えます:

- ビヘイビア・クラスのメソッド名を参照する文字列 (上の例など)
- オブジェクトまたはクラス名と文字列のメソッド名 (括弧なし) 例 [$object, 'methodName']
- 無名関数

イベント・ハンドラのシグニチャは次のようにしてください。$event はイベントのパラメータを参照します。イベントの詳細については イベント セクションを参照してください。

```
function ($event) {
}
```

5.4.3 ビヘイビアをアタッチする

コンポーネント へのビヘイビアのアタッチは、静的にも動的にも可能です。実際は、前者のほうがより一般的ですが。

ビヘイビアを静的にアタッチするには、ビヘイビアをアタッチしたいコンポーネント・クラスの behaviors() メソッドをオーバーライドします。 behaviors() メソッドは、ビヘイビアの 構成 のリストを返さなければなりません。 各ビヘイビアの構成内容は、ビヘイビアのクラス名でも、構成情報配列でもかまいません。

```
namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // 無名ビヘイビア ビヘイビア・クラス名のみ
            MyBehavior::className(),

            // 名前付きビヘイビア ビヘイビア・クラス名のみ
            'myBehavior2' => MyBehavior::className(),

            // 無名ビヘイビア 構成情報配列
            [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],
```

```
            // 名前付きビヘイビア 構成情報配列
            'myBehavior4' => [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop2' => 'value2',
            ]
        ];
    }
}
```

ビヘイビア構成に対応する配列のキーを指定することによって、ビヘイビアに名前を関連付けることができます。この場合、ビヘイビアは 名前付きビヘイビア と呼ばれます。上の例では、2つの名前付きビヘイビア myBehavior2 と myBehavior4 があります。ビヘイビアが名前と関連付けられていない場合は、 無名ビヘイビア と呼ばれます。

ビヘイビアを動的にアタッチするには、ビヘイビアがアタッチされるコンポーネントの yii\base\Component::attachBehavior() メソッドを呼びます:

```
use app\components\MyBehavior;

// ビヘイビア・オブジェクトをアタッチ
$component->attachBehavior('myBehavior1', new MyBehavior);

// ビヘイビア・クラスをアタッチ
$component->attachBehavior('myBehavior2', MyBehavior::className());

// 構成情報配列をアタッチ
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::className(),
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

yii\base\Component::attachBehaviors() メソッドを使うと、いちどに複数のビヘイビアをアタッチできます:

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior,  // 名前付きビヘイビア
    MyBehavior::className(),          // 無名ビヘイビア
]);
```

次のように、 構成情報 を通じてビヘイビアをアタッチすることもできます:

```
[
    'as myBehavior2' => MyBehavior::className(),

    'as myBehavior3' => [
        'class' => MyBehavior::className(),
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

詳しくは 構成情報 のセクションを参照してください。

5.4.4 ビヘイビアを使用する

ビヘイビアを使用するには、まず上記の方法に従って コンポーネント にアタッチします。ビヘイビアがコンポーネントにアタッチされれば、その使用方法はシンプルです。

あなたは、アタッチされているコンポーネントを介して、ビヘイビアの パブリック メンバ変数、 または getter や setter によって定義されたプロパティにアクセスすることができます:

```
// "prop1" はビヘイビア・クラス内で定義されたプロパティ
echo $component->prop1;
$component->prop1 = $value;
```

また同様に、ビヘイビアの パブリック・メソッドも呼ぶことができます:

```
// foo() はビヘイビア・クラス内で定義されたパブリック・メソッド
$component->foo();
```

ご覧のように、 `$component` は `prop1` と `foo()` を定義していないにもかかわらず、 アタッチされたビヘイビアによって、それらをコンポーネント定義の一部であるかのように使うことができるのです。

もし2つのビヘイビアが同じプロパティやメソッドを定義し、かつ両方とも同じコンポーネントにアタッチされている場合は、 プロパティやメソッドのアクセス時に、最初に コンポーネントにアタッチされたビヘイビアが優先されます。

ビヘイビアはコンポーネントにアタッチされるとき、名前と関連付けられているかもしれません。その場合、 その名前を使用してビヘイビア・オブジェクトにアクセスすることができます:

```
$behavior = $component->getBehavior('myBehavior');
```

また、コンポーネントにアタッチされた全てのビヘイビアを取得することもできます:

```
$behaviors = $component->getBehaviors();
```

5.4.5 ビヘイビアをデタッチする

ビヘイビアをデタッチするには、ビヘイビアに付けられた名前とともに `yii\base\Component::detachBehavior()` を呼び出します:

```
$component->detachBehavior('myBehavior1');
```

全ての ビヘイビアをデタッチすることもできます:

```
$component->detachBehaviors();
```

5.4.6 TimestampBehavior を利用する

しめくくりに、yii\behaviors\TimestampBehavior を見てみましょう。このビヘイビアは、 insert()、update() または save() のメソッドを通じてアクティブ・レコード モデルが保存されるときに、 タイムスタンプ属性の自動的な更新をサポートします。

まず、使用しようと考えている アクティブ・レコード クラスに、このビヘイビアをアタッチします:

```
namespace app\models\User;

use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::className(),
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', '
    updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                // UNIX タイムスタンプではなく datetime を使う場合は
                // 'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

上のビヘイビア構成は、レコードが:

- 挿入されるとき、ビヘイビアは現在の UNIX タイムスタンプを created_at と updated_at 属性に割り当てます
- 更新されるとき、ビヘイビアは現在の UNIX タイムスタンプを updated_at 属性に割り当てます

  補足: 上記の実装が MySQL データベースで動作するようにするためには、created_at と updated_at のカラムを UNIX タイムスタンプ になるように int(11) として宣言してください。

このコードが所定の位置にあれば、例えば User オブジェクトがあって、それを保存しようとしたら、そこで、 created_at と updated_at が自動的に現在の UNIX タイムスタンプで埋められます。

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
```

```
echo $user->created_at;  // 現在のタイムスタンプが表示される
```

TimestampBehavior は、また、指定された属性に現在のタイムスタンプを割り当てて それをデータベースに保存する、便利なメソッド touch() を提供しています。

```
$user->touch('login_time');
```

5.4.7 その他のビヘイビア

その他にも、内蔵または外部ライブラリによって利用できるビヘイビアがいくつかあります。

- yii\behaviors\BlameableBehavior - 指定された属性に現在のユーザ ID を自動的に設定します。
- yii\behaviors\SluggableBehavior - 指定された属性に、URL のスラグとして使用できる値を 自動的に設定します。
- yii\behaviors\AttributeBehavior - 特定のイベントが発生したときに、ActiveRecord オブジェクトの一つまたは複数の属性に、 指定された値を自動的に設定します。
- yii2tech\ar\softdelete\SoftDeleteBehavior[3] - ActiveRecord をソフト・デリートおよびソフト・リストアするメソッド、 すなわち、レコードの削除を示すフラグまたはステータスを設定するメソッドを提供します。
- yii2tech\ar\position\PositionBehavior[4] - レコードの順序を整数のフィールドによって管理することが出来るように、 順序変更メソッドを提供します。

5.4.8 ビヘイビアとトレイトの比較

ビヘイビアは、主となるクラスにそのプロパティやメソッドを「注入する」という点で トレイト[5] に似ていますが、これらは多くの面で異なります。以下に説明するように、それらは互いに長所と短所を持っています。 それらは代替手段というよりも、むしろ相互補完関係のようなものです。

ビヘイビアを使う理由

ビヘイビアは通常のクラスのように、継承をサポートしています。いっぽうトレイトは、 言語サポートされたコピー&ペーストとみなすことができます。トレイトは継承をサポートしません。

ビヘイビアは、コンポーネント・クラスの変更を必要とせず、コンポーネントに動的にアタッチまたはデタッチすることが可能です。 トレ

[3] https://github.com/yii2tech/ar-softdelete
[4] https://github.com/yii2tech/ar-position
[5] https://secure.php.net/traits

イトを使用するには、トレイトを使うクラスのコードを書き換える必要があります。

ビヘイビアは構成可能ですがトレイトは不可能です。

ビヘイビアは、イベントに応答することで、コンポーネントのコード実行をカスタマイズできます。

同じコンポーネントにアタッチされた異なるビヘイビア間で名前の競合がある場合、その競合は自動的に、 先にコンポーネントにアタッチされたものを優先することで解消されます。 異なるトレイトによって引き起こされる名前競合の場合は、 影響を受けるプロパティやメソッドの名前変更による、手動での解決が必要です。

トレイトを使う理由

ビヘイビアは時間もメモリも食うオブジェクトなので、トレイトはビヘイビアよりはるかに効率的です。

トレイトはネイティブな言語構造であるため、IDE との相性に優れています。

5.5 構成情報

新しいオブジェクトを作成したり、既存のオブジェクトを初期化するとき、Yii では構成情報が広く使用されています。 構成情報は通常、作成されるオブジェクトのクラス名、およびオブジェクトの プロパティ に割り当てられる初期値のリストを含みます。 構成情報は、オブジェクトの イベント にアタッチされるハンドラのリストや、オブジェクトにアタッチされる ビヘイビア のリストを含むこともできます。

以下では、データベース接続を作成して初期化するために、構成情報が使用されています:

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];

$db = Yii::createObject($config);
```

`Yii::createObject()` メソッドは引数に構成情報の配列を受け取り、構成情報で名前指定されたクラスをインスタンス化してオブジェクトを作成します。 オブジェクトがインスタンス化されるとき、構成情報の残りの部分を使って、 オブジェクトのプロパティ、イベント・ハンドラ、およびビヘイビアが初期化されます。

すでにオブジェクトがある場合は、構成情報配列でオブジェクトのプロパティを初期化するのに `Yii::configure()` を使用することができます:

```
Yii::configure($object, $config);
```

なお、この場合には、構成情報配列に class 要素を含んではいけません。

5.5.1 構成情報の形式

構成情報の形式は、フォーマルには次のように説明できます:

```
[
    'class' => 'ClassName',
    'propertyName' => 'propertyValue',
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

ここで

- class 要素は、作成されるオブジェクトの完全修飾クラス名を指定します。
- propertyName 要素は、名前で指定されたプロパティの初期値を指定します。 キーはプロパティ名で、値はそれに対応する初期値です。パブリック・メンバ変数と getter/setter によって定義されている プロパティ のみを設定することができます。
- on eventName 要素は、どのようなハンドラがオブジェクトの イベント にアタッチされるかを指定します。 配列のキーが on に続けてイベント名という書式になることに注意してください。サポートされているイベント・ハンドラの形式については、 イベント のセクションを参照してください。
- as behaviorName 要素は、どのような ビヘイビア がオブジェクトにアタッチされるかを指定します。 配列のキーが as に続けてビヘイビア名という書式になり、$behaviorConfig で示される値が、ここで説明する一般的な構成情報のような、 ビヘイビアを作成するための構成情報になることに注意してください。

下記は、初期プロパティ値、イベント・ハンドラ、およびビヘイビアでの構成を示した例です:

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... プロパティ初期値 ...
    ],
]
```

5.5.2 構成情報を使用する

構成情報は Yii の多くの場所で使用されています。このセクションの冒

頭では、 Yii::createObject() を使って、構成情報に応じてオブジェクトを作成する方法を示しました。この項では、 アプリケーションの構成とウィジェットの構成という、二つの主要な構成情報の用途を説明します。

アプリケーションの構成

アプリケーション の構成情報は、おそらく Yii の中で最も複雑な配列のひとつです。 それは アプリケーション クラスが、設定可能なプロパティとイベントを数多く持つためです。 さらに重要なことは、その components プロパティが、アプリケーションに登録されている コンポーネントの生成用の構成情報配列を受け取ることができることです。以下は、 ベーシック・プロジェクト・テンプレート のアプリケーション構成ファイルの概要です。

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
        'log' => [
            'class' => 'yii\log\Dispatcher',
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                ],
            ],
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=stay2',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
];
```

この構成情報には、 class キーがありません。それは、エントリ・スクリプト で以下のように、 クラス名が既に与えられて使用されているためです。

```
(new yii\web\Application($config))->run();
```

アプリケーションの components プロパティ構成の詳細については、 アプリケーション のセクションと サービス・ロケータ のセクションにあります。

バージョン 2.0.11 以降では、アプリケーション構成で container プロパティを使って 依存注入コンテナ を構成することがサポートされています。例えば、

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'container' => [
        'definitions' => [
            'yii\widgets\LinkPager' => ['maxButtonCount' => 5]
        ],
        'singletons' => [
            // 依存注入コンテナのシングルトンの構成
        ]
    ]
];
```

definitions と singletons の構成情報配列に使用できる値とその実例についてさらに知るためには、 依存注入コンテナ の記事の 高度な実際の使用方法 のセクションを読んでください。

ウィジェットの構成

ウィジェット を使用するときは、多くの場合、ウィジェットのプロパティをカスタマイズするために、構成情報を使用する必要があります。 yii\base\Widget::widget() と yii\base\Widget::begin() の両メソッドを使って、ウィジェットを作成できます。 それらは、以下のような構成情報配列を取ります。

```
use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
        ['label' => 'ホーム', 'url' => ['site/index']],
        ['label' => '製品', 'url' => ['product/index']],
        ['label' => 'ログイ
    ン', 'url' => ['site/login'], 'visible' => Yii::$app->user->isGuest],
    ],
]);
```

上記のコードは、 Menu ウィジェットを作成し、その activateItems プロパティが false になるよう初期化します。 items プロパティも、表示されるメニュー項目で構成されます。

クラス名がすでに与えられているので、構成情報配列が class キーを持つべきではないことに注意してください。

5.5.3 構成情報ファイル

構成情報がとても複雑になる場合、一般的な方法は、 構成情報ファイルと呼ばれる、ひとつまたは複数の PHP ファイルにそれを格納することです。 構成情報ファイルは、構成情報を表す PHP 配列を返します。 たとえば、次のように、 web.php と名づけたファイルにアプリケーションの構成情報を保持することができます。

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => require __DIR__ . '/components.php',
];
```

components の構成情報もまた複雑になるため、上記のように、 components.php と呼ぶ別のファイルにそれを格納し web.php でそのファイルを “require” しています。 この components.php の内容は、次のようになっています。

```
return [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
    'log' => [
        'class' => 'yii\log\Dispatcher',
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
            ],
        ],
    ],
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'mysql:host=localhost;dbname=stay2',
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
    ],
];
```

構成情報ファイルに格納されている構成情報を取得するには、以下のように、それを “require” するだけです:

```
$config = require 'path/to/web.php';
(new yii\web\Application($config))->run();
```

5.5.4 デフォルト設定

Yii::createObject() メソッドは、 依存性注入コンテナ をベースに実装されています。 そのため、指定されたクラスが Yii::createObject() を使用して作成されるとき、そのすべてのインスタンスに適用される、 いわゆる デフォルト設定 のセットを指定することができます。デフォルト設定は、 ブートストラップ 段階のコード内で Yii::$container->set() を呼び出すことで指定することができます。

たとえばあなたが、すべてのリンク・ページャが最大で5つのページ・ボタン (デフォルト値は10) を伴って表示されるよう yii\widgets\LinkPager をカスタマイズしたいとき、その目標を達成するには次のコードを使用することができます。

```
\Yii::$container->set('yii\widgets\LinkPager', [
    'maxButtonCount' => 5,
]);
```

デフォルト設定を使用しなければ、あなたは、リンク・ページャを使うすべての箇所で maxButtonCount を設定しなければなりません。

5.5.5 環境定数

構成情報は、多くの場合、アプリケーションが実行される環境に応じて変化します。たとえば、 開発環境では mydb_dev という名前のデータベースを使用し、本番サーバ上では mydb_prod データベースを 使用したいかもしれません。環境の切り替えを容易にするために、Yii は、あなたのアプリケーションの エントリ・スクリプト で定義可能な YII_ENV という名前の定数を提供します。 たとえば:

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

YII_ENV を次のいずれかの値と定義することができます:

- prod: 本番環境。定数 YII_ENV_PROD が true と評価されます。 とくに定義しない場合、これが YII_ENV のデフォルト値です。
- dev: 開発環境。定数 YII_ENV_DEV が true と評価されます。
- test: テスト環境。定数 YII_ENV_TEST が true と評価されます。

これらの環境定数を使用すると、現在の環境に基づいて条件付きで構成情報を指定することもできます。 たとえば、アプリケーション構成情報には、開発環境での デバッグ・ツールバーとデバッガ を有効にするために、次のコードを含むことができます。

```
$config = [...];

if (YII_ENV_DEV) {
    // 'dev' 環境用に構成情報を調整
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

5.6 エイリアス

ファイル・パスや URL を表すのにエイリアスを使用すると、あなたはプロジェクト内で絶対パスや URL をハードコードする必要がなくなります。 エイリアスは、通常のファイル・パスや URL と区別するために、 `@` 文字で始まる必要があります。 先頭に `@` を付けずに定義されたエイリアスは、`@` 文字が先頭に追加されます。

Yii はすでに利用可能な多くの事前定義エイリアスを持っています。 たとえば、 `@yii` というエイリアスは Yii フレームワークのインストール・パスを表し、`@web` は現在実行中のウェブ・アプリケーションのベース URL を表します。

5.6.1 エイリアスを定義する

`Yii::setAlias()` を呼び出すことにより、ファイル・パスまたは URL のエイリアスを定義することができます。

```
// ファイル・パスのエイリアス
Yii::setAlias('@foo', '/path/to/foo');

// URL のエイリアス
Yii::setAlias('@bar', 'http://www.example.com');

// \foo\Bar クラスを保持する具体的なファイルのエイリアス
Yii::setAlias('@foo/Bar.php', '/definitely/not/foo/Bar.php');
```

> 補足: エイリアスされているファイル・パスや URL は、必ずしも実在するファイルまたはリソースを参照しない場合があります。

定義済みのエイリアスがあれば、スラッシュ `/` に続けて 1 つ以上のパス・セグメントを追加することで (`Yii::setAlias()` の呼び出しを必要とせずに) 新しいエイリアスを導出することができます。 `Yii::setAlias()` を通じて定義されたエイリアスは ルート・エイリアス となり、それから派生したエイリアスは 派生エイリアス になります。たとえば、 `@foo` がルート・エイリアスなら、 `@foo/bar/file.php` は派生エイリアスです。

エイリアスを、他のエイリアス (ルートまたは派生のいずれか) を使用して定義することができます:

```
Yii::setAlias('@foobar', '@foo/bar');
```

ルート・エイリアスは通常、 ブートストラップ 段階で定義されます。 たとえば、エントリ・スクリプト で `Yii::setAlias()` を呼び出すことができます。 便利なように、 アプリケーション は、`aliases` という名前の書き込み可能なプロパティを提供しており、 それをアプリケーションの 構成情報 で設定することが可能です。

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'http://www.example.com',
    ],
];
```

5.6.2 エイリアスを解決する

Yii::getAlias() を呼び出して、ルート・エイリアスが表すファイル・パスまたは URL を解決することができます。 同メソッドで、派生エイリアスを対応するファイル・パスまたは URL に解決することもできます。

```
echo Yii::getAlias('@foo');               // /path/to/foo を表示
echo Yii::getAlias('@bar');               // http://www.example.com を表示
echo Yii::getAlias('@foo/bar/file.php'); // /path/to/foo/bar/file.php を表
    示
```

派生エイリアスによって表されるパスや URL は、 派生エイリアス内のルート・エイリアス部分を対応するパスや URL で置換して決定されます。

> 補足: Yii::getAlias() メソッドは、 結果のパスや URL が実在するファイルやリソースを参照しているかをチェックしません。

ルート・エイリアス名にはスラッシュ / 文字を含むことができます。 Yii::getAlias() メソッドは、 エイリアスのどの部分がルート・エイリアスであるかを賢く判別し、 正確に対応するファイル・パスや URL を決定します:

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php');  // /path/to/foo/test/file.php を表示
Yii::getAlias('@foo/bar/file.php');   // /path2/bar/file.php を表示
```

もし @foo/bar がルート・エイリアスとして定義されていなければ、最後のステートメントは /path/to/foo/bar/file.php を表示します。

5.6.3 エイリアスを使用する

Yii では、多くの場所で、パスや URL に変換する Yii::getAlias() を呼び出す必要なく、 エイリアスが認識されます。 たとえば、 yii\caching\FileCache::$cachePath は、ファイル・パスとファイル・パスを表すエイリアスの両方を受け入れることが出来ます。 これは、接頭辞 @ によって、ファイル・パスとエイリアスを区別することが出来るためです。

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

プロパティやメソッドのパラメータがエイリアスをサポートしているかどうかは、API ドキュメントに注意を払ってください。

5.6.4 事前定義されたエイリアス

Yii では、一般的に使用されるファイルのパスと URL を簡単に参照できるよう、エイリアスのセットが事前に定義されています:

- `@yii`, `BaseYii.php` ファイルがあるディレクトリ (フレームワーク・ディレクトリとも呼ばれます)
- `@app`, 現在実行中のアプリケーションの ベース・パス
- `@runtime`, 現在実行中のアプリケーションの ランタイム・パス 。デフォルトは `@app/runtime` 。
- `@webroot`, 現在実行中のウェブ・アプリケーションのウェブ・ルート・ディレクトリ。 エントリス・クリプトを含むディレクトリによって決定されます。
- `@web`, 現在実行中のウェブ・アプリケーションのベース URL。これは、 `yii\web\Request::$baseUrl` と同じ値を持ちます。
- `@vendor`, `Composer` のベンダー・ディレクトリ 。デフォルトは `@app/vendor` 。
- `@bower`, bower パッケージ[6] が含まれるルート・ディレクトリ。デフォルトは `@vendor/bower` 。
- `@npm`, npm パッケージ[7] が含まれるルート・ディレクトリ。デフォルトは `@vendor/npm` 。

`@yii` エイリアスは エントリ・スクリプト に `Yii.php` ファイルを読み込んだ時点で定義されます。 エイリアスの残りの部分は、アプリケーションのコンストラクタ内で、アプリケーションの 構成情報 を適用するときに定義されます。

> 補足: `@web` と `@webroot` のエイリアスは、その説明が示しているように、ウェブ・アプリケーション の中で定義されます。従って、デフォルトでは コンソール・アプリケーション では利用できません。

5.6.5 エクステンションのエイリアス

Composer でインストールされる エクステンション のそれぞれに対してエイリアスが自動的に定義されます。 各エイリアスは、その `composer.json` ファイルで宣言されたエクステンションのルート名前空間にちなん

[6] http://bower.io/
[7] https://www.npmjs.org/

で名付けられ、 パッケージのルート・ディレクトリを表します。たとえば、あなたが yiisoft/yii2-jui エクステンションをインストールしたとすると、 自動的に @yii/jui というエイリアスが ブートストラップ 段階で定義されます。これは次のものと等価です。

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.7 クラスのオートローディング

Yiiは、必要となるすべてのクラス・ファイルを特定してインクルードするにあたり、 クラスのオートローディング・メカニズム[8] を頼りにします。 Yii は、PSR-4 標準[9] に準拠した、高性能なクラスのオートローダを提供しています。 このオートローダは、あなたが Yii.php ファイルをインクルードするときにインストールされます。

> 補足: 説明を簡単にするため、このセクションではクラスのオートローディングについてのみ話します。しかし、 ここに記述されている内容は、インタフェイスとトレイトのオートローディングにも同様に適用されることに注意してください。

5.7.1 Yii オートローダを使用する

Yii のクラス・オートローダを使用するには、クラスを作成して名前を付けるとき、次の二つの単純なルールに従わなければなりません:

- 各クラスは 名前空間[10] の下になければなりません (例 foo\bar\MyClass)
- 各クラスは次のアルゴリズムで決定される個別のファイルに保存されなければなりません:

```
// $className は先頭にバック・スラッシュを持たない完全修飾クラス名
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) . '.php'
    );
```

たとえば、クラス名と名前空間が foo\bar\MyClass であれば、対応するクラス・ファイルのパスの エイリアス は、 @foo/bar/MyClass.php になります。このエイリアスがファイル・パスとして解決できるようにするためには、@foo または @foo/bar のどちらかが、 ルート・エイリアス でなければなりません。

ベーシック・プロジェクト・テンプレート を使用している場合、最上位の名前空間 app の下にクラスを置くことができ、 そうすると、新しいエイリアスを定義しなくても、Yii によってそれらをオートロードできる

[8] https://secure.php.net/manual/ja/language.oop5.autoload.php
[9] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md
[10] https://secure.php.net/manual/ja/language.namespaces.php

ようになります。これは @app が 事前定義されたエイリアス であるため で、app\components\MyClass のようなクラス名を 今説明したアルゴリズム に従って、クラス・ファイル AppBasePath/components/MyClass.php であると 解決することが出来ます。

アドバンスト・プロジェクト・テンプレート[11] では、各層がそれ自身 のルート・エイリアスを持っています。たとえば、 フロントエンド層は ルート・エイリアス @frontend を持ち、バックエンド層のルート・エイリ アスは @backend です。その結果、名前空間 frontend の下に フロントエン ド・クラスを置き、バックエンド・クラスを backend の下に置けます。これで、これらのクラスは Yii のオートローダによって オートロードでき るようになります。

独自の名前空間をオートローダに追加するためには、Yii::setAlias() を使って、その名前空間のベース・ディレクトリに対するエイリアスを 定義する必要があります。 例えば、path/to/foo ディレクトリに配置され ている foo 名前空間に属するクラスをロードするためには、'Yii::setAlias('@foo', 'path/to/foo') を呼び出します。

5.7.2 クラス・マップ

Yii のクラス・オートローダは、 クラス・マップ 機能をサポートして おり、クラス名を対応するクラス・ファイルのパスにマップできます。 オートローダがクラスをロードするときは、クラスがマップに見つかる かどうかを最初にチェックします。もしあれば、対応する ファイル・パ スは、それ以上チェックされることなく、直接インクルードされます。 これでクラスのオートローディングを非常に高速化できます。 実際のと ころ、すべての Yii のコア・クラスは、この方法でオートロードされて います。

次の方法で、 Yii::$classMap に格納されるクラス・マップにクラスを 追加できます:

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

クラス・ファイルのパスを指定するのに、 エイリアス を使うことがで きます。クラスが使用される前にマップが準備できるように、 クラス・ マップの設定は ブートストラップ プロセス内でする必要があります。

5.7.3 他のオートローダの使用

Yii はパッケージ依存関係マネージャとして Composer を包含しているの で、Composer のオートローダもインストールすることをお勧めします。 あなたが独自のオートローダを持つサードパーティ・ライブラリを使用 している場合は、 それらもインストールする必要があります。

Yii オートローダを他のオートローダと一緒に使うときは、他のすべ てのオートローダがインストールされた 後で 、 Yii.php ファイルをイン

[11] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/README.md

クルードする必要があります。これで Yii のオートローダが、任意クラスのオートローディング要求に応答する最初のものになります。 たとえば、次のコードは ベーシック・プロジェクト・テンプレート の エントリ・スクリプト から抜き出したものです。 最初の行は、Composer のオートローダをインストールしており、二行目は Yii のオートローダをインストールしています。

```
require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';
```

あなたは Yii のオートローダを使わず、Composer のオートローダだけを単独で使用することもできます。しかし、そうすることによって、 あなたのクラスのオートローディングのパフォーマンスは低下し、クラスをオートロード可能にするために Composer が設定したルールに従わなければならなくなります。

> 情報: Yiiのオートローダを使用したくない場合は、`Yii.php` ファイルのあなた独自のバージョンを作成し、 それを エントリ・スクリプト でインクルードする必要があります。

5.7.4 エクステンション・クラスのオートロード

Yii のオートローダは、 エクステンション クラスのオートロードが可能です。唯一の要件は、 エクステンションがその `composer.json` ファイルに正しく `autoload` セクションを指定していることです。 `autoload` の指定方法の詳細については Composer のドキュメント[12] 参照してください。

Yii のオートローダを使用しない場合でも、まだ Composer のオートローダがエクステンション・クラスをオートロードすることが可能です。

5.8 サービス・ロケータ

サービス・ロケータは、アプリケーションが必要とする可能性のある各種のサービス (またはコンポーネント) を提供する方法を知っているオブジェクトです。 サービス・ロケータ内では、各コンポーネントは単一のインスタンスとして存在し、ID によって一意に識別されます。 あなたは、この ID を使用してサービス・ロケータからコンポーネントを取得できます。

Yii では、サービス・ロケータは単純に `yii\di\ServiceLocator` のインスタンス、またはその子クラスのインスタンスです。

Yii の中で最も一般的に使用されるサービス・ロケータは、`\Yii::$app` を通じてアクセスできる アプリケーション・オブジェクトです。 これが提供するサービスは、 アプリケーション・コンポーネント と呼ばれる `request` 、 `response`、 `urlManager` などのコンポーネントです。あなたは

[12] https://getcomposer.org/doc/04-schema.md#autoload

サービス・ロケータによって提供される機能を通じて、 簡単に、これらのコンポーネントを構成、あるいは独自の実装に置き換え、といったことができます。

アプリケーション・オブジェクトの他に、各モジュール・オブジェクトもまたサービス・ロケータです。モジュールは ツリー走査 を実装しています。

サービス・ロケータを使用する最初のステップは、コンポーネントを登録することです。コンポーネントは、 `yii\di\ServiceLocator::set()` を通じて登録することができます。次のコードは、コンポーネントを登録するさまざまな方法を示しています。

```
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// コンポーネントの作成に使われるクラス名を使用して "cache" を登録
$locator->set('cache', 'yii\caching\ApcCache');

// コンポーネントの作成に使われる構成情報配列を使用して "db" を登録
$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// コンポーネントを構築する匿名関数を使って "search" を登録
$locator->set('search', function () {
    return new app\components\SolrService;
});

// コンポーネントを使って "pageCache" を登録
$locator->set('pageCache', new FileCache);
```

いったんコンポーネントが登録されたら、次の二つの方法のいずれかで、その ID を使ってそれにアクセスすることができます:

```
$cache = $locator->get('cache');
// または代りに
$cache = $locator->cache;
```

上記のように、`yii\di\ServiceLocator` を使うと、コンポーネント ID を使用して、プロパティのようにコンポーネントにアクセスすることができます。 あなたが最初にコンポーネントにアクセスしたとき、`yii\di\ServiceLocator` は コンポーネントの登録情報を使用してコンポーネントの新しいインスタンスを作成し、 それを返します。後でそのコンポーネントが再度アクセスされた場合、サービス・ロケータは同じインスタンスを返します。

`yii\di\ServiceLocator::has()` を使って、コンポーネント ID がすでに登録されているかをチェックできます。 無効な ID で `yii\di`

\ServiceLocator::get() を呼び出した場合、例外が投げられます。

サービス・ロケータは多くの場合、 構成情報 で作成されるため、components という名前の書き込み可能プロパティが提供されています。これで一度に複数のコンポーネントを設定して登録することができます。 次のコードは、サービス・ロケータ (例えば アプリケーション) を db、cache、tz、search コンポーネントとともに構成するための構成情報配列を示しています。

```
return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'tz' => function() {
            return new \DateTimeZone(Yii::$app->formatter->defaultTimeZone);
        },
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');
            // ... その他の初期化 ...
            return $solr;
        },
    ],
];
```

上記において、search コンポーネントを構成する別の方法があります。SolrService のインスタンスを構築する PHP コールバックを直接に書く代りに、 下記のように、そういうコールバックを返すスタティックなクラス・メソッドを使うことが出来ます。

```
class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... その他の初期化 ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];
```

この方法は、Yii に属さないサードパーティのライブラリをカプセル化する Yii コンポーネントをリリースしようとする場合に、特に推奨される代替手法です。 上で示されているようなスタティックなメソッドを使ってサードパーティのオブジェクトを構築する複雑なロジックを表現します。 そうすれば、あなたのコンポーネントのユーザは、コンポーネントを構成するスタティックなメソッドを呼ぶ必要があるだけになります。

5.8.1 ツリー走査

モジュールは任意にネストすることが出来ます。Yii アプリケーションは本質的にモジュールのツリーなのです。 これらのモジュールのそれぞれがサービス・ロケータである訳ですから、子がその親にアクセスできるようにするのは理にかなった事です。 これによって、モジュールは、ルートのサービス・ロケータを参照して `Yii::$app->get('db')` とする代りに、`$this->get('db')` とすることが出来ます。 また、開発者にモジュール内で構成をオーバーライドするオプションを提供できることも、この仕組の利点です。

モジュールからサービスを引き出そうとする全てのリクエストは、そのモジュールが要求に応じられない場合は、すべてその親に渡されます。

モジュール内のコンポーネントの構成情報は、親モジュール内のコンポーネントの構成情報とは決してマージされないことに注意して下さい。 サービス・ロケータ・パターンによって私たちは名前の付いたサービスを定義することが出来ますが、同じ名前のサービスが同じ構成パラメータを使用すると想定することは出来ません。

5.9 依存注入コンテナ

依存注入 (DI) コンテナは、オブジェクトとそれが依存するすべてのオブジェクトを、インスタンス化し、設定する方法を知っているオブジェクトです。 なぜ DI コンテナが便利なのかは、Martin Fowler の記事[13] の説明がわかりやすいでしょう。 ここでは、主に Yii の提供する DI コンテナの使用方法を説明します。

5.9.1 依存注入

Yii は `yii\di\Container` クラスを通して DI コンテナの機能を提供します。 これは、次の種類の依存注入をサポートしています:

- コンストラクタ・インジェクション
- メソッド・インジェクション
- セッター/プロパティ・インジェクション
- PHP コーラブル・インジェクション

[13] http://martinfowler.com/articles/injection.html

コンストラクタ・インジェクション

DI コンテナは、コンストラクタ引数の型ヒントの助けを借りて、コンストラクタ・インジェクションをサポートしています。 コンテナが新しいオブジェクトの作成に使用されるさい、そのオブジェクトがどういうクラスやインタフェイスに依存しているかを、型ヒントがコンテナに教えます。 コンテナは、依存するクラスやインタフェイスのインスタンスを取得して、 コンストラクタを通して、新しいオブジェクトにそれらを注入しようと試みます。たとえば

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}

$foo = $container->get('Foo');
// これは下記と等価:
$bar = new Bar;
$foo = new Foo($bar);
```

メソッド・インジェクション

通常、クラスの依存はコンストラクタに渡されて、そのクラスの内部でライフサイクル全体にわたって利用可能になります。 メソッド・インジェクションを使うと、クラスのメソッドの一つだけに必要となる依存、例えば、コンストラクタに渡すことが不可能であったり、 大半のユース・ケースにおいてはオーバーヘッドが大きすぎるような依存を提供することが可能になります。

クラス・メソッドを次の例の `doSomething` メソッドのように定義することが出来ます。

```
class MyClass extends \yii\base\Component
{
    public function __construct(/* 軽量の依存はここに */, $config = [])
    {
        // ...
    }

    public function doSomething($param1, \my\heavy\Dependency $something)
    {
        // $something を使って何かをする
    }
}
```

このメソッドを呼ぶためには、あなた自身で `\my\heavy\Dependency` のインスタンスを渡すか、または、次のように `yii\di\Container::invoke()` を使います。

```
$obj = new MyClass(/*...*/);
```

```
Yii::$container->invoke([$obj, 'doSomething'], ['param1' => 42]); //
    $something は DI コンテナによって提供され
    る
```

セッター/プロパティ・インジェクション

セッター/プロパティ・インジェクションは、構成情報 を通してサポートされます。 依存を登録するときや、新しいオブジェクトを作成するときに、対応するセッターまたはプロパティを通しての依存注入に使用される構成情報を、 コンテナに提供することが出来ます。 たとえば

```
use yii\base\BaseObject;

class Foo extends BaseObject
{
    public $bar;

    private $_qux;

    public function getQux()
    {
        return $this->_qux;
    }

    public function setQux(Qux $qux)
    {
        $this->_qux = $qux;
    }
}

$container->get('Foo', [], [
    'bar' => $container->get('Bar'),
    'qux' => $container->get('Qux'),
]);
```

> 情報: yii\di\Container::get() メソッドは三番目のパラメータを、生成されるオブジェクトに適用されるべき構成情報配列として受け取ります。 クラスが yii\base\Configurable インタフェイスを実装している場合 (例えば、クラスが yii\base\BaseObject である場合) には、 この構成情報配列がクラスのコンストラクタの最後のパラメータとして渡されます。 そうでない場合は、構成情報はオブジェクトが生成された 後で 適用されることになります。

PHP コーラブル・インジェクション

この場合、コンテナは、登録された PHP のコーラブルを使用して、クラスの新しいインスタンスを構築します。 yii\di\Container::get() が呼ばれるたびに、対応するコーラブルが起動されます。 このコーラブル

が、依存を解決し、新しく作成されたオブジェクトに適切に依存を注入する役目を果たします。 たとえば

```
$container->set('Foo', function ($container, $params, $config) {
    $foo = new Foo(new Bar);
    // ... その他の初期化 ...
    return $foo;
});

$foo = $container->get('Foo');
```

新しいオブジェクトを構築するための複雑なロジックを隠蔽するために、スタティックなクラスメソッドをコーラブルとして使うことが出来ます。例えば、

```
class FooBuilder
{
    public static function build($container, $params, $config)
    {
        $foo = new Foo(new Bar);
        // ... その他の初期化 ...
        return $foo;
    }
}

$container->set('Foo', ['app\helper\FooBuilder', 'build']);

$foo = $container->get('Foo');
```

このようにすれば、Foo クラスを構成しようとする人は、Foo がどのように構築されるかを気にする必要はもうなくなります。

5.9.2 依存を登録する

yii\di\Container::set() を使って依存を登録することができます。登録には依存の名前だけでなく、依存の定義が必要です。 依存の名前は、クラス名、インタフェイス名、エイリアス名を指定することができます。 依存の定義には、クラス名、構成情報配列、PHPのコーラブルを指定できます。

```
$container = new \yii\di\Container;

// クラス名そのままの登録。これは省略可能です。
$container->set('yii\db\Connection');

// インタフェイスの登録
// クラスがインタフェイスに依存する場合、対応するクラスが
// 依存オブジェクトとしてインスタンス化されます
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// エイリアス名の登録。$container->get('foo') を使って
// Connection のインスタンスを作成できます
$container->set('foo', 'yii\db\Connection');
```

```
// 'Instance::of' を使ってエイリアスの登録。
$container->set('bar', Instance::of('foo'));

// 構成情報をともなうクラスの登録。クラスが get() でインスタンス化
// されるとき構成情報が適用されます
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// クラスの構成情報をともなうエイリアス名の登録
// この場合、クラスを指定する "class" または "__class" 要素が必要です
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// コーラブルなクロージャまたは配列の登録
// このコーラブルは $container->get('db') が呼ばれるたびに実行されます
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});
$container->set('db', ['app\db\DbFactory', 'create']);

// コンポーネント・インスタンスの登録
// $container->get('pageCache') は呼ばれるたびに毎回同じインスタンスを返します
$container->set('pageCache', new FileCache);
```

> 補足: 依存の名前が対応する依存の定義と同じである場合は、
> それを DI コンテナに登録する必要はありません。

`set()` を介して登録された依存は、依存が必要とされるたびにインスタンスを生成します。 `yii\di\Container::setSingleton()` を使うと、 単一のインスタンスしか生成しない依存を登録することができます:

```
$container->setSingleton('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

5.9.3 依存を解決する

依存を登録すると、新しいオブジェクトを作成するのに DI コンテナを使用することができます。 そして、コンテナが自動的に依存をインスタ

ンス化し、新しく作成されたオブジェクトに注入して、 依存を解決します。依存の解決は再帰的に行われます。つまり、ある依存が他の依存を持っている場合、 それらの依存も自動的に解決されます。

get() を使って、オブジェクトのインスタンスを作成または取得することができます。 このメソッドは依存の名前を引数として取りますが、依存の名前は、クラス名、インタフェイス名、あるいは、エイリアス名で指定できます。 依存の名前は、 set() を介して登録されていることもあれば、 setSingleton() を介して登録されていることもあります。オプションで、クラスのコンストラクタのパラメータのリストや、設定情報 を渡して、新しく作成されるオブジェクトを構成することも出来ます。

たとえば、

```
// "db" は事前に登録されたエイリアス名
$db = $container->get('db');

// これと同じ意味: $engine = new \app\components\SearchEngine($apiKey,
    $apiSecret, ['type' => 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey,
    $apiSecret], ['type' => 1]);
```

見えないところで、DIコンテナは、単に新しいオブジェクトを作成するよりもはるかに多くの作業を行います。 コンテナは、最初にクラスのコンストラクタを調査し、依存するクラスまたはインタフェイスの名前を見つけると、 自動的にそれらの依存を再帰的に解決します。

次のコードでより洗練された例を示します。UserLister クラスは UserFinderInterface インタフェイスを実装するオブジェクトに依存します。UserFinder クラスはこのインタフェイスを実装していて、かつ、Connection オブジェクトに依存します。これらのすべての依存は、クラスのコンストラクタのパラメータの型ヒントによって宣言されています。依存の登録が適切にされていれば、DI コンテナは自動的にこれらの依存を解決し、単純に get('userLister') を呼び出すだけで新しい UserLister インスタンスを作成できます。

```
namespace app\models;

use yii\base\BaseObject;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends BaseObject implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
```

```
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends BaseObject
{
    public $finder;

    public function __construct(UserFinderInterface $finder, $config = [])
    {
        $this->finder = $finder;
        parent::__construct($config);
    }
}

$container = new Container;
$container->set('yii\db\Connection', [
    'dsn' => '...',
]);
$container->set('app\models\UserFinderInterface', [
    'class' => 'app\models\UserFinder',
]);
$container->set('userLister', 'app\models\UserLister');

$lister = $container->get('userLister');
```

```
// と、いうのはこれと同じ:

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$lister = new UserLister($finder);
```

5.9.4 実際の使用方法

あなたのアプリケーションの エントリ・スクリプト で `Yii.php` ファイルをインクルードするとき、 Yii は DI コンテナを作成します。この DI コンテナは `Yii::$container` を介してアクセス可能です。 `Yii::createObject()` を呼び出したとき、 このメソッドは実際にはコンテナの `get()` メソッドを呼び出して新しいオブジェクトを作成します。 前述のとおり、DI コンテナは(もしあれば)自動的に依存を解決し、取得されたオブジェクトにそれらを注入します。 Yii は、新しいオブジェクトを作成するコアコードのほとんどにおいて `Yii::createObject()` を使用しています。このことは、 `Yii::$container` を操作することでグローバルにオブジェクトをカスタマイズすることができるということを意味してい

ます。

例として、 yii\widgets\LinkPager のページ・ネーションボタンのデフォルト個数をグローバルにカスタマイズしてみましょう。

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

そして、次のコードでビューでウィジェットを使用すれば、maxButtonCount プロパティは、 クラスで定義されているデフォルト値 10 の代わりに 5 で初期化されます。

```
echo \yii\widgets\LinkPager::widget();
```

ただし、DI コンテナを経由して設定された値を上書きすることは、まだ可能です:

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

> ヒント: ウィジェットの呼び出しで与えられたプロパティは常に DI コンテナが持つ定義を上書きします。 たとえ、'options' => ['id' => 'mypager'] のように配列を指定したとしても、それらは他のオプションとマージされるのでなく、他のオプションを置換えてしまいます。

もう一つの例は、DI コンテナの自動コンストラクタ・インジェクションの利点を活かすものです。 あなたのコントローラ・クラスが、ホテル予約サービスのような、いくつかの他のオブジェクトに依存するとします。 あなたは、コンストラクタのパラメータを通して依存を宣言して、DI コンテナにそれを解決させることができます。

```
namespace app\controllers;

use yii\web\Controller;
use app\components\BookingInterface;

class HotelController extends Controller
{
    protected $bookingService;

    public function __construct($id, $module, BookingInterface
    $bookingService, $config = [])
    {
        $this->bookingService = $bookingService;
        parent::__construct($id, $module, $config);
    }
}
```

あなたがブラウザからこのコントローラにアクセスすると、BookingInterface をインスタンス化できない、という不平を言う エラーが表示されるでしょう。これは、この依存に対処する方法を DI コンテナに教える必要があるからです:

```
\Yii::$container->set('app\components\BookingInterface', 'app\components\
    BookingService');
```

これで、あなたが再びコントローラにアクセスするときは、app\components\BookingService のインスタンスが作成され、コントローラのコンストラクタに3番目のパラメータとして注入されるようになります。

Yii 2.0.36 以降は、PHP 7 を使う場合に、ウェブおよびコンソール両方のコントローラでアクション・インジェクションを利用することが出来ます。

```
namespace app\controllers;

use yii\web\Controller;
use app\components\BookingInterface;

class HotelController extends Controller
{
    public function actionBook($id, BookingInterface $bookingService)
    {
        $result = $bookingService->book($id);
        // ...
    }
}
```

5.9.5 高度な実際の使用方法

API アプリケーションを開発していて、以下のクラスを持っているとします。

- app\components\Request クラス。yii\web\Request から拡張され、追加の機能を提供する。
- app\components\Response クラス。yii\web\Response から拡張。 生成されるときに、format プロパティが json に設定されなければならない。
- app\storage\FileStorage および app\storage\DocumentsReader クラス。何らかのファイルストレージに配置されているドキュメントを操作するロジックを実装する。

```
class FileStorage
{
    public function __construct($root) {
        // あれやこれや
    }
}

class DocumentsReader
{
    public function __construct(FileStorage $fs) {
        // なんやかんや
    }
}
```

setDefinitions() または setSingletons() のメソッドに構成情報の配列を渡して、複数の定義を一度に構成することが可能です。 これらの

メソッドは、構成情報配列を反復して、各アイテムに対し、 それぞれ `set()` を呼び出します。

構成情報配列のフォーマットは、

- `key`: クラス名、インタフェイス名、または、エイリアス名。 このキーが `set()` メソッドの最初の引数 `$class` として渡されます。
- `value`: `$class` と関連づけられる定義。指定できる値は、`set()` の `$definition` パラメータのドキュメントで説明されています。 `set()` メソッドに二番目のパラメータ `$definition` として渡されます。

例として、上述の要求に従うように私たちのコンテナを構成しましょう。

```
$container->setDefinitions([
    'yii\web\Request' => 'app\components\Request',
    'yii\web\Response' => [
        'class' => 'app\components\Response',
        'format' => 'json'
    ],
    'app\storage\DocumentsReader' => function ($container, $params, $config)
     {
        $fs = new app\storage\FileStorage('/var/tempfiles');
        return new app\storage\DocumentsReader($fs);
    }
]);

$reader = $container->get('app\storage\DocumentsReader);
// 構成情報に書かれている依存とともに DocumentReader オブジェクトが生成されます
```

> ヒント: バージョン 2.0.11 以降では、アプリケーションの構成情報を使って、宣言的なスタイルでコンテナを構成することが出来ます。 構成情報 のガイドの アプリケーションの構成 のセクションを参照してください。

これで全部動きますが、`DocumentWriter` クラスを生成する必要がある場合には、`FileStorage` オブジェクトを生成する行をコピペすることになるでしょう。 もちろん、それが一番スマートな方法ではありません。

依存を解決する のセクションで説明したように、`set()` と `setSingleton()` は、 オプションで、第三の引数として依存のコンストラクタのパラメータを取ることが出来ます。 コンストラクタのパラメータを設定するために、`__construct()` オプションを使うことが出来ます。

では、私たちの例を修正しましょう。

```
$container->setDefinitions([
    'tempFileStorage' => [ // 便利なようにエイリアスを作りました
        'class' => 'app\storage\FileStorage',
        '__construct()' => ['/var/tempfiles'], // 何らかの構成ファイルから抽
    出することも可能
    ],
    'app\storage\DocumentsReader' => [
        'class' => 'app\storage\DocumentsReader',
```

```
        '__construct()' => [Instance::of('tempFileStorage')],
    ],
    'app\storage\DocumentsWriter' => [
        'class' => 'app\storage\DocumentsWriter',
        '__construct()' => [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader);
// 前の例と全く同じオブジェクトが生成されます
```

`Instance::of('tempFileStorage')` という記法に気づいたことでしょう。 これは、`Container` が、`tempFileStorage` という名前で登録されている依存を黙示的に提供して、 `app\storage\DocumentsWriter` のコンストラクタの最初の引数として渡す、ということを意味しています。

> 補足: `setDefinitions()` および `setSingletons()` のメソッドは、バージョン 2.0.11 以降で利用できます。

構成情報の最適化にかかわるもう一つのステップは、いくつかの依存をシングルトンとして登録することです。 `set()` を通じて登録された依存は、必要になるたびに、毎回インスタンス化されます。 しかし、ある種のクラスは実行時を通じて状態を変化させませんので、 アプリケーションのパフォーマンスを高めるためにシングルトンとして登録することが出来ます。

`app\storage\FileStorage` クラスが好例でしょう。これは単純な API によってファイル・システムに対する何らかの操作を実行するもの (例えば `$fs->read()` や `$fs->write()`) ですが、これらの操作はクラスの内部状態を変化させないものです。 従って、このクラスのインスタンスを一度だけ生成して、それを複数回使用することが可能です。

```
$container->setSingletons([
    'tempFileStorage' => [
        'class' => 'app\storage\FileStorage',
        '__construct()' => ['/var/tempfiles']
    ],
]);

$container->setDefinitions([
    'app\storage\DocumentsReader' => [
        'class' => 'app\storage\DocumentsReader',
        '__construct()' => [Instance::of('tempFileStorage')],
    ],
    'app\storage\DocumentsWriter' => [
        'class' => 'app\storage\DocumentsWriter',
        '__construct()' => [Instance::of('tempFileStorage')],
    ]
]);

$reader = $container->get('app\storage\DocumentsReader');
```

5.9.6 いつ依存を登録するか

依存は、新しいオブジェクトが作成されるとき必要とされるので、それらの登録は可能な限り早期に行われるべきです。 推奨されるプラクティスは以下のとおりです:

- あなたがアプリケーションの開発者である場合は、アプリケーションの構成情報を使って依存を登録することが出来ます。 構成情報のガイドの アプリケーションの構成 のセクションを読んでください。
- あなたが再配布可能な エクステンション の開発者である場合は、エクステンションのブートストラップ・クラス内で 依存を登録することができます。

5.9.7 まとめ

依存注入と サービス・ロケータ はともに、疎結合でよりテストしやすい方法でのソフトウェア構築を可能にする、 定番のデザインパターンです。 依存注入とサービス・ロケータへのより深い理解を得るために、Martin の記事[14] を読むことを強くお勧めします。

Yii はその サービス・ロケータ を、依存注入 (DI) コンテナの上に実装しています。 サービス・ロケータは、新しいオブジェクトのインスタンスを作成しようとするとき、DI コンテナに呼び出しを転送します。 後者は、依存を、上で説明したように自動的に解決します。

[14] http://martinfowler.com/articles/injection.html

Chapter 6

データベースの取り扱い

6.1 データベース・アクセス・オブジェクト

PDO[1] の上に構築された Yii DAO (データベース・アクセス・オブジェクト) は、 リレーショナル・データベースにアクセスするためのオブジェクト指向 API を提供するものです。 これは、データベースにアクセスする他のもっと高度な方法、例えば クエリ・ビルダ や アクティブ・レコード の基礎でもあります。

Yii DAO を使うときは、主として素の SQL と PHP 配列を扱う必要があります。 結果として、Yii DAO はデータベースにアクセスする方法としては最も効率的なものになります。 しかし、SQL の構文はデータベースによってさまざまに異なる場合がありますので、Yii DAO を使用するということは、特定のデータベースに依存しないアプリケーションを作るためには追加の労力が必要になる、ということをも同時に意味します。

Yii 2.0 では、DAO は下記の DBMS のサポートを内蔵しています。

- MySQL[2]
- MariaDB[3]
- SQLite[4]
- PostgreSQL[5]: バージョン 8.4 以上。
- CUBRID[6]: バージョン 9.3 以上。
- Oracle[7]
- MSSQL[8]: バージョン 2008 以上。

補足: PHP 7 用の pdo_oci の新しいバージョンは、現在、

[1] https://secure.php.net/manual/ja/book.pdo.php
[2] http://www.mysql.com/
[3] https://mariadb.com/
[4] http://sqlite.org/
[5] http://www.postgresql.org/
[6] http://www.cubrid.org/
[7] http://www.oracle.com/us/products/database/overview/index.html
[8] https://www.microsoft.com/en-us/sqlserver/default.aspx

> ソース・コードとしてのみ存在します。 コミュニティによる説明[9] に従ってコンパイルするか、 または、PDO エミュレーション・レイヤ[10] を使って下さい。

6.1.1 DB 接続を作成する

データベースにアクセスするためには、まずは、`yii\db\Connection` のインスタンスを作成して、データベースに接続する必要があります。

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

DB 接続は、たいていは、さまざまな場所でアクセスする必要がありますので、次のように、 アプリケーション・コンポーネント の形式で構成するのが通例です。

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=example',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

こうすると `Yii::$app->db` という式で DB 接続にアクセスすることが出来るようになります。

> ヒント: あなたのアプリケーションが複数のデータベースにアクセスする必要がある場合は、複数の DB アプリケーション・コンポーネントを構成することが出来ます。

DB 接続を構成するときは、つねに `dsn` プロパティによってデータ・ソース名 (DSN) を指定しなければなりません。 DSN の形式はデータベースによってさまざまに異なります。 詳細は PHP マニュアル[11] を参照して下さい。下記にいくつかの例を挙げます。

- MySQL, MariaDB: `mysql:host=localhost;dbname=mydatabase`

[9] https://github.com/yiisoft/yii2/issues/10975#issuecomment-248479268
[10] https://github.com/taq/pdooci
[11] https://secure.php.net/manual/ja/function.PDO-construct.php

- SQLite: `sqlite:/path/to/database/file`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID: `cubrid:dbname=demodb;host=localhost;port=33000`
- MS SQL Server (sqlsrv ドライバ経由): `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (dblib ドライバ経由): `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (mssql ドライバ経由): `mssql:host=localhost;dbname=mydatabase`
- Oracle: `oci:dbname=//localhost:1521/mydatabase`

ODBC 経由でデータベースに接続しようとする場合は、`yii\db\Connection::$driverName` プロパティを構成して、Yii に実際のデータベースのタイプを知らせなければならないことに注意してください。 例えば、

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

`dsn` プロパティに加えて、たいていは `username` と `password` も構成しなければなりません。 構成可能なプロパティの全てのリストは `yii\db\Connection` を参照して下さい。

> 情報: DB 接続のインスタンスを作成するとき、実際のデータベース接続は、最初の SQL を実行するか、 `open()` メソッドを明示的に呼ぶかするまでは確立されません。

> ヒント: 時として、何らかの環境変数を初期化するために、データベース接続を確立した直後に何かクエリを実行したい場合があるでしょう (例えば、タイムゾーンや文字セットを設定するなどです)。 そうするために、データベース接続の `afterOpen` イベントに対するイベント・ハンドラを登録することが出来ます。 以下のように、アプリケーションの構成情報に直接にハンドラを登録することが出来ます。

```
'db' => [
    // ...
    'on afterOpen' => function($event) {
        // $event->sender は DB 接続を指す
        $event->sender->createCommand("SET time_zone = 'UTC'")->
    execute();
    }
]
```

MS SQL Server でバイナリ・データを正しく処理するためには追加の接続オプションが必要になります。

```
'db' => [
 'class' => 'yii\db\Connection',
    'dsn' => 'sqlsrv:Server=localhost;Database=mydatabase',
    'attributes' => [
        \PDO::SQLSRV_ATTR_ENCODING => \PDO::SQLSRV_ENCODING_SYSTEM
    ]
],
```

6.1.2 SQL クエリを実行する

いったんデータベース接続のインスタンスを得てしまえば、次の手順に従って SQL クエリを実行することが出来ます。

1. 素の SQL クエリで `yii\db\Command` を作成する。
2. パラメータをバインドする (オプション)。
3. `yii\db\Command` の SQL 実行メソッドの一つを呼ぶ。

次に、データベースからデータを読み出すさまざまな方法を例示します。

```
// 行のセットを返す。各行は、カラム名と値の連想配列。
// クエリが結果を返さなかった場合は空の配列が返される。
$posts = Yii::$app->db->createCommand('SELECT * FROM post')
            ->queryAll();

// 一つの行 最初の行() を返す。
// クエリの結果が無かった場合は false が返される。
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=1')
           ->queryOne();

// 一つのカラム 最初のカラム() を返す。
// クエリが結果を返さなかった場合は空の配列が返される。
$titles = Yii::$app->db->createCommand('SELECT title FROM post')
             ->queryColumn();

// スカラ値を返す。
// クエリの結果が無かった場合は false が返される。
$count = Yii::$app->db->createCommand('SELECT COUNT(*) FROM post')
             ->queryScalar();
```

> 補足: 精度を保つために、対応するデータベース・カラムの型が数値である場合でも、 データベースから取得されたデータは、全て文字列として表現されます。

パラメータをバインドする

パラメータを持つ SQL から DB コマンドを作成するときは、SQL インジェクション攻撃を防止するために、 ほとんど全ての場合においてパラメータをバインドする手法を用いるべきです。例えば、

```
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status')
           ->bindValue(':id', $_GET['id'])
           ->bindValue(':status', 1)
           ->queryOne();
```

SQL 文において、一つまたは複数のパラメータ・プレースホルダ (例えば、上記のサンプルにおける :id) を埋め込むことが出来ます。 パラメータ・プレースホルダは、コロンから始まる文字列でなければなりません。 そして、次に掲げるパラメータをバインドするメソッドの一つを使って、パラメータの値をバインドします。

- bindValue(): 一つのパラメータの値をバインドします。
- bindValues(): 一回の呼び出しで複数のパラメータの値をバインドします。
- bindParam(): bindValue() と似ていますが、 パラメータを参照渡しでバインドすることもサポートしています。

次の例はパラメータをバインドする方法の選択肢を示すものです。

```
$params = [':id' => $_GET['id'], ':status' => 1];

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status')
           ->bindValues($params)
           ->queryOne();

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status', $params)
           ->queryOne();
```

パラメータ・バインディングは プリペアド・ステートメント[12] によって実装されています。 パラメータ・バインディングには、SQL インジェクション攻撃を防止する以外にも、SQL 文を一度だけ準備して異なるパラメータで複数回実行することにより、 パフォーマンスを向上させる効果もあります。例えば、

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
// ...
```

bindParam() はパラメータを参照渡しでバインドすることをサポートしていますので、 上記のコードは次のように書くことも出来ます。

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id')
              ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();
```

[12] https://secure.php.net/manual/ja/mysqli.quickstart.prepared-statements.php

```
$id = 2;
$post2 = $command->queryOne();
```

クエリの実行の前にプレースホルダを変数 `$id` にバインドし、そして、後に続く各回の実行の前にその変数の値を変更していること (これは、たいてい、ループで行います) に着目してください。 このやり方でクエリを実行すると、パラメータの値が違うごとに新しいクエリを実行するのに比べて、 はるかに効率を良くすることが出来ます。

> 情報: パラメータ・バインディングは、素の SQL を含む文字列に値を挿入しなければならない場所でのみ使用されます。クエリ・ビルダ や アクティブ・レコード のような高レベルの抽象的レイヤーでは、 多くの場所で SQL に変換される値の配列を指定する場合がよくあります。 これらの場所では Yii によってパラメータ・バインディングが内部的に実行されますので、パラメータを手動で指定する必要はありません。

SELECT しないクエリを実行する

今までのセクションで紹介した `queryXyz()` メソッドは、すべて、データベースからデータを取得する SELECT クエリを扱うものでした。 データを返さないクエリのためには、代りに `yii\db\Command::execute()` メソッドを呼ばなければなりません。例えば、

```
Yii::$app->db->createCommand('UPDATE post SET status=1 WHERE id=1')
    ->execute();
```

`yii\db\Command::execute()` メソッドは SQL の実行によって影響を受けた行の数を返します。

INSERT、UPDATE および DELETE クエリのためには、素の SQL を書く代りに、それぞれ、 `insert()`、`update()`、`delete()` を呼んで、対応する SQL を構築することが出来ます。 これらのメソッドは、テーブルとカラムの名前を適切に引用符で囲み、パラメータの値をバインドします。例えば、

```
// INSERT テーブル名(, カラムの値)
Yii::$app->db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// UPDATE テーブル名(, カラムの値, 条件)
Yii::$app->db->createCommand()->update('user', ['status' => 1], 'age > 30')-
    >execute();

// DELETE テーブル名(, 条件)
Yii::$app->db->createCommand()->delete('user', 'status = 0')->execute();
```

`batchInsert()` を呼んで複数の行を一気に挿入することも出来ます。 この方法は、一度に一行を挿入するよりはるかに効率的です。

```
// テーブル名, カラム名, カラムの値
Yii::$app->db->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();
```

もう一つの有用なメソッドは upsert() です。upsert は、(ユニーク制約に合致する)行がまだ存在しない場合はデータベース・テーブルに行を挿入し、 既に行が存在している場合は行を更新する、アトミックな操作です。

```
Yii::$app->db->createCommand()->upsert('pages', [
    'name' => 'フロント・ページ',
    'url' => 'http://example.com/', // url はユニーク
    'visits' => 0,
], [
    'visits' => new \yii\db\Expression('visits + 1'),
], $params)->execute();
```

上記のコードは、新しいページのレコードを挿入するか、または、既存のレコードの訪問者カウンタをインクリメントします。

上述のメソッド群はクエリを生成するだけであり、実際にそれを実行するためには、常に execute() を呼び出す必要があることに注意してください。

6.1.3 テーブルとカラムの名前を引用符で囲む

特定のデータベースに依存しないコードを書くときには、テーブルとカラムの名前を適切に引用符で囲むことが、たいてい、頭痛の種になります。 データベースによって名前を引用符で囲む規則がさまざまに異なるからです。 この問題を克服するために、次のように、Yii によって導入された引用符の構文を使用することが出来ます。

- カラム名[[]]: 引用符で囲むべきカラム名は、二重角括弧で包む。
- テーブル名{{}}: 引用符で囲むべきテーブル名は、二重波括弧で包む。

Yii DAO は、このような構文を、DBMS 固有の文法に従って、 適切な引用符で囲まれたカラム名とテーブル名に自動的に変換します。 例えば、

```
// MySQL では SELECT COUNT(‘id‘) FROM ‘employee‘ という SQL が実行される
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{employee
    }}")
            ->queryScalar();
```

テーブル接頭辞を使う

あなたの DB テーブル名のほとんどが何か共通の接頭辞を持っている場合は、 Yii DAO によって提供されているテーブル接頭辞の機能を使うことが出来ます。

最初に、アプリケーションの構成情報で、yii\db\Connection::$tablePrefix プロパティによって、テーブル接頭辞を指定します。

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

そして、あなたのコードの中で、そのテーブル接頭辞を名前に持つテーブルを参照しなければならないときには、いつでも テーブル名{{%}} という構文を使います。 パーセント記号は DB 接続を構成したときに指定したテーブル接頭辞に自動的に置き換えられます。 例えば、

```
// MySQL では SELECT COUNT(‘id‘) FROM ‘tbl_employee‘ という SQL が実行される
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{%employee
    }}")
            ->queryScalar();
```

6.1.4 トランザクションを実行する

一続きになった複数の関連するクエリを実行するときは、データの整合性と一貫性を保証するために、 一連のクエリをトランザクションで囲む必要がある場合があります。 一連のクエリのどの一つが失敗した場合でも、データベースは、何一つクエリが実行されなかったかのような状態へとロールバックされます。

次のコードはトランザクションの典型的な使用方法を示すものです。

```
Yii::$app->db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... その他の SQL 文を実行 ...
});
```

上記のコードは、次のものと等価です。こちらの方が、エラー処理のコードをより細かく制御することが出来ます。

```
$db = Yii::$app->db;
$transaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... その他の SQL 文を実行 ...

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
```

```
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

beginTransaction() メソッドを呼ぶことによって、新しいトランザクションが開始されます。 トランザクションは、変数 $transaction に保存された yii\db\Transaction オブジェクトとして表現されます。 そして、実行されるクエリが try...catch... ブロックで囲まれます。 全てのクエリの実行が成功した場合には、トランザクションをコミットするために commit() が呼ばれます。 そうでなく、例外がトリガされてキャッチされた場合は、rollBack() が呼ばれて、トランザクションの中で失敗したクエリに先行するクエリによって行なわれた変更が、ロールバックされます。 そして、throw $e が、まるでそれをキャッチしなかったかのように、例外を再スローしますので、通常のエラー処理プロセスがその例外の面倒を見ることになります。

> 補足: 上記のコードでは、PHP 5.x と PHP 7.x との互換性のために、二つのcatch ブロックを持っています。 \Exception は PHP 7.0 以降では、\Throwable インタフェイス[13] を実装しています。 従って、あなたのアプリケーションが PHP 7.0 以上しか使わない場合は、\Exception の部分を省略することが出来ます。

分離レベルを指定する

Yii は、トランザクションの 分離レベル[14] の設定もサポートしています。デフォルトでは、新しいトランザクションを開始したときは、 データベースシステムによって設定された分離レベルを使用します。デフォルトの分離レベルは、次のようにしてオーバーライドすることが出来ます。

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

Yii::$app->db->transaction(function ($db) {
    ....
}, $isolationLevel);
```

```
// あるいは

$transaction = Yii::$app->db->beginTransaction($isolationLevel);
```

Yii は、最もよく使われる分離レベルのために、四つの定数を提供しています。

[13] https://secure.php.net/manual/ja/class.throwable.php

[14] http://ja.wikipedia.org/wiki/%E3%83%88%E3%83%A9%E3%83%B3%E3%82%B6%E3%82%AF%E3%82%B7%E3%83%A7%E3%83%B3%E5%88%86%E9%9B%A2%E3%83%AC%E3%83%99%E3%83%AB

- `yii\db\Transaction::READ_UNCOMMITTED` - 最も弱いレベル。ダーティー・リード、非再現リード、ファントムが発生しうる。
- `yii\db\Transaction::READ_COMMITTED` - ダーティー・リードを回避。
- `yii\db\Transaction::REPEATABLE_READ` - ダーティー・リードと非再現リードを回避。
- `yii\db\Transaction::SERIALIZABLE` - 最も強いレベル。上記の問題を全て回避。

分離レベルを指定するためには、上記の定数を使う以外に、あなたが使っている DBMS によってサポートされている有効な構文の文字列を使うことも出来ます。 例えば、PostreSQL では、`"SERIALIZABLE READ ONLY DEFERRABLE"` を使うことが出来ます。

DBMS によっては、接続全体に対してのみ分離レベルの設定を許容しているものがあることに注意してください。 その場合、すべての後続のトランザクションは、指定しなくても、それと同じ分離レベルで実行されます。 従って、この機能を使用するときは、矛盾する設定を避けるために、全てのトランザクションについて分離レベルを明示的に指定しなければなりません。 このチュートリアルを書いている時点では、この制約の影響を受ける DBMS は MSSQL と SQLite だけです。

> 補足: SQLite は、二つの分離レベルしかサポートしていません。すなわち、`READ UNCOMMITTED` と `SERIALIZABLE` しか使えません。 他のレベルを使おうとすると、例外が投げられます。
>
> 補足: PostgreSQL は、トランザクションを開始する前に分離レベルを指定することを許容していません。 すなわち、トランザクションを開始するときに、分離レベルを直接に指定することは出来ません。 この場合、トランザクションを開始した後に `yii\db\Transaction::setIsolationLevel()` を呼び出す必要があります。

トランザクションを入れ子にする

あなたの DBMS が Savepoint をサポートしている場合は、次のように、複数のトランザクションを入れ子にすることが出来ます。

```
Yii::$app->db->transaction(function ($db) {
    // 外側のトランザクション

    $db->transaction(function ($db) {
        // 内側のトランザクション
    });
});
```

あるいは、

```
$db = Yii::$app->db;
$outerTransaction = $db->beginTransaction();
```

```
try {
    $db->createCommand($sql1)->execute();

    $innerTransaction = $db->beginTransaction();
    try {
        $db->createCommand($sql2)->execute();
        $innerTransaction->commit();
    } catch (\Exception $e) {
        $innerTransaction->rollBack();
        throw $e;
    } catch(\Throwable $e) {
        $transaction->rollBack();
        throw $e;
    }

    $outerTransaction->commit();
} catch (\Exception $e) {
    $outerTransaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

6.1.5 レプリケーションと読み書きの分離

多くの DBMS は、データベースの可用性とサーバのレスポンス・タイムを向上させるために、 データベース・レプリケーション[15] をサポートしています。 データベース・レプリケーションによって、データはいわゆる マスタ・サーバ から スレーブ・サーバ に複製されます。 データの書き込みと更新はすべてマスタ・サーバ上で実行されなければなりませんが、データの読み出しはスレーブ・サーバ上でも可能です。

データベース・レプリケーションを活用して読み書きの分離を達成するために、 yii\db\Connection コンポーネントを下記のように構成することが出来ます。

```
[
    'class' => 'yii\db\Connection',

    // マスタの構成
    'dsn' => 'マスタ・サーバの DSN',
    'username' => 'master',
    'password' => '',

    // スレーブの共通の構成
    'slaveConfig' => [
        'username' => 'slave',
```

[15] http://ja.wikipedia.org/wiki/%E3%83%AC%E3%83%97%E3%83%AA%E3%82%B1%E3%83%BC%E3%82%B7%E3%83%A7%E3%83%B3#.E3.83.87.E3.83.BC.E3.82.BF.E3.83.99.E3.83.BC.E3.82.B9

```
        'password' => '',
        'attributes' => [
            // 短かめの接続タイムアウトを使う
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // スレーブの構成のリスト
    'slaves' => [
        ['dsn' => 'スレーブ・サーバ 1 の DSN'],
        ['dsn' => 'スレーブ・サーバ 2 の DSN'],
        ['dsn' => 'スレーブ・サーバ 3 の DSN'],
        ['dsn' => 'スレーブ・サーバ 4 の DSN'],
    ],
]
```

上記の構成は、一つのマスタと複数のスレーブを指定するものです。 読み出しのクエリを実行するためには、スレーブの一つが接続されて使用され、書き込みのクエリを実行するためには、マスタが使われます。 そのような読み書きの分離が、この構成によって、自動的に達成されます。例えば、

```
// 上記の構成を使って Connection のインスタンスを作成する
$db = Yii::createObject($config);

// スレーブの一つに対してクエリを実行する
$rows = Yii::$app->db->createCommand('SELECT * FROM user LIMIT 10')->
    queryAll();

// マスタに対してクエリを実行する
Yii::$app->db->createCommand("UPDATE user SET username='demo' WHERE id=1")->
    execute();
```

> 情報: yii\db\Command::execute() を呼ぶことで実行されるクエリは、書き込みのクエリと見なされ、 yii\db\Command の “query” メソッドのうちの一つによって実行されるその他すべてのクエリは、読み出しクエリと見なされます。 現在アクティブなスレーブ接続は Yii::$app->db->slave によって取得することが出来ます。

Connection コンポーネントは、スレーブ間のロード・バランス調整とフェイルオーバーをサポートしています。 読み出しクエリを最初に実行するときに、Connection コンポーネントはランダムにスレーブを選んで接続を試みま・す。 そのスレーブが「死んでいる」ことが分かったときは、他のスレーブを試します。 スレーブが一つも使用できないときは、マスタに接続します。 サーバ・ステータスキャッシュ を構成することによって、「死んでいる」サーバを記憶し、 一定期間 はそのサーバへの接続を再試行しないようにすることが出来ます。

> 情報: 上記の構成では、すべてのスレーブに対して 10 秒の接続タイムアウトが指定されています。 これは、10 秒以内に接続できなければ、そのスレーブは「死んでいる」と見なされることを意味します。 このパラメータは、実際の環境に基づいて調整することが出来ます。

複数のマスタと複数のスレーブという構成にすることも可能です。例えば、

```
[
    'class' => 'yii\db\Connection',

    // マスタの共通の構成
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // 短かめの接続タイムアウトを使う
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // マスタの構成のリスト
    'masters' => [
        ['dsn' => 'マスタ・サーバ 1 の DSN'],
        ['dsn' => 'マスタ・サーバ 2 の DSN'],
    ],

    // スレーブの共通の構成
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // 短かめの接続タイムアウトを使う
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // スレーブの構成のリスト
    'slaves' => [
        ['dsn' => 'スレーブ・サーバ 1 の DSN'],
        ['dsn' => 'スレーブ・サーバ 2 の DSN'],
        ['dsn' => 'スレーブ・サーバ 3 の DSN'],
        ['dsn' => 'スレーブ・サーバ 4 の DSN'],
    ],
]
```

上記の構成は、二つのマスタと四つのスレーブを指定しています。 `Connection` コンポーネントは、スレーブ間での場合と同じように、マスタ間でのロード・バランス調整とフェイルオーバーをサポートしています。 一つ違うのは、マスタが一つも利用できないときは例外が投げられ

る、という点です。

> 補足: masters プロパティを使って一つまたは複数のマスタを構成する場合は、 データベース接続を定義する Connection オブジェクト自体の他のプロパティ (例えば、dsn、username、password) は全て無視されます。

デフォルトでは、トランザクションはマスタ接続を使用します。 そして、トランザクション内では、全ての DB 操作はマスタ接続を使用します。例えば、

```
$db = Yii::$app->db;
// トランザクションはマスタ接続で開始される
$transaction = $db->beginTransaction();

try {
    // クエリは両方ともマスタに対して実行される
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE id=1")->
    execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

スレーブ接続を使ってトランザクションを開始したいときは、次のように、明示的にそうする必要があります。

```
$transaction = Yii::$app->db->slave->beginTransaction();
```

時として、読み出しクエリの実行にマスタ接続を使うことを強制したい場合があります。 これは、useMaster() メソッドを使うことによって達成できます。

```
$rows = Yii::$app->db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

直接に Yii::$app->db->enableSlaves を false に設定して、全てのクエリをマスタ接続に向けることも出来ます。

6.1.6 データベース・スキーマを扱う

Yii DAO は、新しいテーブルを作ったり、テーブルからカラムを削除したりなど、データベース・スキーマを操作することを可能にする一揃いのメソッドを提供しています。 以下がそのソッドのリストです。

- createTable(): テーブルを作成する

- renameTable(): テーブルの名前を変更する
- dropTable(): テーブルを削除する
- truncateTable(): テーブルの全ての行を削除する
- addColumn(): カラムを追加する
- renameColumn(): カラムの名前を変更する
- dropColumn(): カラムを削除する
- alterColumn(): カラムを変更する
- addPrimaryKey(): プライマリ・キーを追加する
- dropPrimaryKey(): プライマリ・キーを削除する
- addForeignKey(): 外部キーを追加する
- dropForeignKey(): 外部キーを削除する
- createIndex(): インデックスを作成する
- dropIndex(): インデックスを削除する

これらのメソッドは次のようにして使うことが出来ます。

```
// CREATE TABLE
Yii::$app->db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

上記の配列は、生成されるカラムの名前と型を記述しています。 Yii はカラムの型のために一連の抽象データ型を提供しているため、データベースの違いを意識せずにスキーマを定義することが可能です。 これらの抽象データ型は、テーブルが作成されるデータベースによって異なる DBMS 固有の型定義に変換されます。 詳しい情報は createTable() メソッドの API ドキュメントを参照してください。

データベースのスキーマを変更するだけでなく、テーブルに関する定義情報を DB 接続の getTableSchema() メソッドによって取得することも出来ます。 例えば、

```
$table = Yii::$app->db->getTableSchema('post');
```

このメソッドは、テーブルのカラム、プライマリ・キー、外部キーなどの情報を含む yii\db\TableSchema オブジェクトを返します。 これらの情報は、主として クエリ・ビルダ や アクティブ・レコード によって利用されて、 特定のデータベースに依存しないコードを書くことを助けてくれています。

6.2 クエリ・ビルダ

データベース・アクセス・オブジェクト の上に構築されているクエリ・ビルダは、SQL クエリをプログラム的に、 かつ、DBMS の違いを意識せずに作成することを可能にしてくれます。 クエリ・ビルダを使うと、生の SQL 文を書くことに比べて、より読みやすい SQL 関連のコードを書き、より安全な SQL 文を生成することが容易になります。

通常、クエリ・ビルダの使用は、二つのステップから成ります。

1. SELECT SQL 文のさまざまな部分 (例えば、SELECT、FROM) を表現する yii\db\Query オブジェクトを構築する。

2. yii\db\Query のクエリ・メソッド (例えば all()) を実行して、データベースからデータを取得する。

次のコードは、クエリ・ビルダを使用する典型的な方法を示すものです。

```
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->all();
```

上記のコードは、次の SQL クエリを生成して実行します。 ここでは、:last_name パラメータは 'Smith' という文字列にバインドされています。

```
SELECT `id`, `email`
FROM `user`
WHERE `last_name` = :last_name
LIMIT 10
```

> 情報: 通常は、yii\db\QueryBuilder ではなく、主として yii\db\Query を使用します。 前者は、クエリ・メソッドの一つを呼ぶときに、後者によって黙示的に起動されます。 yii\db\QueryBuilder は、DBMS に依存しない yii\db\Query オブジェクトから、DBMS に依存する SQL 文を生成する (例えば、テーブルやカラムの名前を DBMS ごとに違う方法で引用符で囲む) 役割を負っているクラスです。

6.2.1 クエリを構築する

yii\db\Query オブジェクトを構築するために、さまざまなクエリ構築メソッドを呼んで、SQL クエリのさまざまな部分を定義します。 これらのメソッドの名前は、SQL 文の対応する部分に使われる SQL キーワードに似たものになっています。 例えば、SQL クエリの FROM の部分を定義するためには、from() メソッドを呼び出します。 クエリ構築メソッドは、すべて、クエリ・オブジェクトそのものを返しますので、複数の呼び出しをチェーンしてまとめることが出来ます。

以下で、それぞれのクエリ構築メソッドの使用方法を説明しましょう。

select()

select() メソッドは、SQL 文の SELECT 句を定義します。 選択されるカラムは、次のように、配列または文字列として指定することが出来ます。 選択されるカラムの名前は、クエリ・オブジェクトから SQL 文が生成されるときに、自動的に引用符で囲まれます。

```
$query->select(['id', 'email']);

// 次と等価

$query->select('id, email');
```

選択されるカラム名は、生の SQL クエリを書くときにするように、テーブル接頭辞 および/または カラムのエイリアスを含むことが出来ます。 例えば、

```
$query->select(['user.id AS user_id', 'email']);

// 次と等価

$query->select('user.id AS user_id, email');
```

カラムを指定するのに配列形式を使っている場合は、配列のキーを使ってカラムのエイリアスを指定することも出来ます。 例えば、上記のコードは次のように書くことが出来ます。

```
$query->select(['user_id' => 'user.id', 'email']);
```

クエリを構築するときに select() メソッドを呼ばなかった場合は、* がセレクトされます。 すなわち、全て のカラムが選択されることになります。

カラム名に加えて、DB 式をセレクトすることも出来ます。 カンマを含む DB 式をセレクトする場合は、自動的に引用符で囲む機能が誤動作しないように、配列形式を使わなければなりません。例えば、

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name", 'email'])
    ;
```

生の SQL が使われる場所ではどこでもそうですが、セレクトに DB 式を書く場合には、テーブルやカラムの名前を表すために 特定のデータベースに依存しない引用符の構文 を使うことが出来ます。

バージョン 2.0.1 以降では、サブ・クエリもセレクトすることが出来ます。 各サブ・クエリは、yii\db\Query オブジェクトの形で指定しなければなりません。例えば、

```
$subQuery = (new Query())->select('COUNT(*)')->from('user');

// SELECT `id`, (SELECT COUNT(*) FROM `user`) AS `count` FROM `post`
$query = (new Query())->select(['id', 'count' => $subQuery])->from('post');
```

重複行を除外してセレクトするためには、次のように、distinct() を呼ぶことが出来ます。

```
// SELECT DISTINCT `user_id` ...
$query->select('user_id')->distinct();
```

追加のカラムをセレクトするためには addSelect() を呼ぶことが出来ます。例えば、

```
$query->select(['id', 'username'])
    ->addSelect(['email']);
```

from()

from() メソッドは、SQL 文の FROM 句を定義します。例えば、

```
// SELECT * FROM `user`
$query->from('user');
```

セレクトの対象になる (一つまたは複数の) テーブルは、文字列または配列として指定することが出来ます。 テーブル名は、生の SQL 文を書くときにするように、スキーマ接頭辞 および/または テーブル・エイリアスを含むことが出来ます。例えば、

```
$query->from(['public.user u', 'public.post p']);

// 次と等価

$query->from('public.user u, public.post p');
```

配列形式を使う場合は、次のように、配列のキーを使ってテーブル・エイリアスを指定することも出来ます。

```
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

テーブル名の他に、yii\db\Query オブジェクトの形で指定することによって、サブ・クエリをセレクトの対象とすることも出来ます。 例えば、

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');

// SELECT * FROM (SELECT `id` FROM `user` WHERE status=1) u
$query->from(['u' => $subQuery]);
```

プレフィックス　また、デフォルトの tablePrefix を適用することも出来ます。 実装の仕方は “データベース・アクセス・オブジェクト” ガイドの “テーブル名を引用符で囲む” のセクション にあります。

where()

where() メソッドは、SQL クエリの WHERE 句を定義します。 WHERE の条件を指定するために、次の4つの形式から一つを選んで使うことが出来ます。

- 文字列形式、例えば、'status=1'

- ハッシュ形式、例えば、['status' => 1, 'type' => 2]
- 演算子形式、例えば、['like', 'name', 'test']
- オブジェクト形式、例えば、new LikeCondition('name', 'LIKE', 'test')

文字列形式　文字列形式は、非常に単純な条件を定義する場合や、DBMSの組み込み関数を使う必要がある場合に最適です。 これは、生の SQL を書いている場合と同じように動作します。例えば、

```
$query->where('status=1');

// あるいは、パラメータ・バインディングを使って、動的にパラメータをバインドする
$query->where('status=:status', [':status' => $status]);

// date フィールドに対して MySQL の YEAR() 関数を使う生の SQL
$query->where('YEAR(somedate) = 2015');
```

次のように、条件式に変数を直接に埋め込んではいけません。 特に、変数の値がユーザの入力に由来する場合、あなたのアプリケーションを SQL インジェクション攻撃にさらすことになりますので、してはいけません。

```
// 危険! $status が整数であることが絶対に確実でなければ、してはいけない。
$query->where("status=$status");
```

パラメータ・バインディング を使う場合は、params() または addParams() を使って、パラメータの指定を分離することが出来ます。

```
$query->where('status=:status')
    ->addParams([':status' => $status]);
```

生の SQL が使われる場所ではどこでもそうですが、文字列形式で条件を書く場合には、テーブルやカラムの名前を表すために 特定のデータベースに依存しない引用符の構文 を使うことが出来ます。

ハッシュ形式　値が等しいことを要求する単純な条件をいくつか AND で結合する場合は、ハッシュ形式を使うのが最適です。 個々の条件を表す配列の各要素は、キーをカラム名、値をそのカラムが持つべき値とします。 例えば、

```
// ...WHERE (`status` = 10) AND (`type` IS NULL) AND (`id` IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

ご覧のように、クエリ・ビルダは頭が良いので、null や配列である値も、適切に処理します。

次のように、サブ・クエリをハッシュ形式で使うことも出来ます。

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE `id` IN (SELECT `id` FROM `user`)
$query->where(['id' => $userQuery]);
```

ハッシュ形式を使う場合、Yii は内部的にパラメータ・バインディングを使用します。 従って、文字列形式 とは対照的に、ここでは手動でパラメータを追加する必要はありません。ただし、Yii はカラム名を決してエスケープしないことに注意して下さい。 従って、ユーザから取得した変数を何ら追加のチェックをすることなくカラム名として渡すと、SQL インジェクション攻撃に対して脆弱になります。 アプリケーションを安全に保つためには、カラム名として変数を使わないこと、または、変数を許容リストによってフィルターすることが必要です。 カラム名をユーザから取得する必要がある場合は、ガイドの データをフィルタリングする という記事を読んで下さい。 例えば、次のコードは脆弱です。

```
// 脆弱なコード:
$column = $request->get('column');
$value = $request->get('value');
$query->where([$column => $value]);
// $value は安全です。しかし、$column の名前はエンコードされません。
```

演算子形式　演算子形式を使うと、任意の条件をプログラム的な方法で指定することが出来ます。これは次の形式を取るものです。

```
演算子
[, オペランド1, オペランド2, ...]
```

ここで、各オペランドは、文字列形式、ハッシュ形式、あるいは、再帰的に演算子形式として指定することが出来ます。 そして、演算子には、次のどれか一つを使うことが出来ます。

- `and`: 複数のオペランドが `AND` を使って結合されます。 例えば、`['and', 'id=1', 'id=2']` は `id=1 AND id=2` を生成します。 オペランドが配列である場合は、ここで説明されている規則に従って文字列に変換されます。 例えば、`['and', 'type=1', ['or', 'id=1', 'id=2']]` は `type=1 AND (id=1 OR id=2)` を生成します。 このメソッドは、文字列を引用符で囲ったりエスケープしたりしません。
- `or`: オペランドが `OR` を使って結合されること以外は `and` 演算子と同じです。
- `not`: オペランド1 だけを受け取って `NOT()` で包みます。例えば、`['not', 'id=1']` は `NOT (id=1)` を生成します。オペランド1 は、それ自体も複数の式を表す配列であっても構いません。例えば、`['not', ['status' => 'draft', 'name' => 'example']]` は `NOT ((status='draft') AND (name='example'))` を生成します。
- `between`: オペランド 1 はカラム名、オペランド 2 と 3 はカラムの値が属すべき範囲の開始値と終了値としなければなりません。 例えば、`['between', 'id', 1, 10]` は `id BETWEEN 1 AND 10` を生成します。

値が二つのカラムの値の間にあるという条件 (例えば、`11 BETWEEN min_id AND max_id`) を構築する必要がある場合は、 `BetweenColumnsCondition` を使用しなければなりません。 条件定義のオブジェクト形式について更に学習するためには 条件 – オブジェクト形式 のセクションを参照して下さい。

- `not between`: 生成される条件において `BETWEEN` が `NOT BETWEEN` に置き換えられる以外は、 `between` と同じです。
- `in`: オペランド 1 はカラム名または DB 式でなければなりません。 オペランド 2 は、配列または `Query` オブジェクトのどちらかを取ることが出来ます。 オペランド 2 が配列である場合は、その配列は、カラムまたは DB 式が該当すべき値域を表すものとされます。 オペランド 2 が `Query` オブジェクトである場合は、サブ・クエリが生成されて、カラムまたは DB 式の値域として使われます。 例えば、`['in', 'id', [1, 2, 3]]` は `id IN (1, 2, 3)` を生成します。 このメソッドは、カラム名を適切に引用符で囲み、値域の値をエスケープします。 `in` 演算子はまた複合カラムをもサポートしています。 その場合、オペランド 1 はカラム名の配列とし、オペランド 2 は配列の配列、または、複合カラムの値域を表す `Query` オブジェクトでなければなりません。 例えば、`['in', ['id', 'name'], [['id' => 1, 'name' => 'oy']]]` は `(id, name) IN ((1, 'oy'))` を生成します。
- `not in`: 生成される条件において `IN` が `NOT IN` に置き換えられる以外は、`in` と同じです。
- `like`: オペランド 1 はカラム名または DB 式、オペランド 2 はそのカラムまたは DB 式がマッチすべき値を示す 文字列または配列でなければなりません。 例えば、`['like', 'name', 'tester']` は `name LIKE '%tester%'` を生成します。 値域が配列として与えられた場合は、複数の `LIKE` 述語が生成されて 'AND' によって結合されます。 例えば、`['like', 'name', ['test', 'sample']]` は `name LIKE '%test%' AND name LIKE '%sample%'` を生成します。 さらに、オプションである三番目のオペランドによって、値の中の特殊文字をエスケープする方法を指定することも出来ます。 このオペランド 3 は、特殊文字とそのエスケープ結果のマッピングを示す配列でなければなりません。 このオペランドが提供されない場合は、デフォルトのエスケープ・マッピングが使用されます。 `false` または空の配列を使って、値が既にエスケープ済みであり、それ以上エスケープを適用すべきでないことを示すことが出来ます。 エスケープマッピングを使用する場合 (または第三のオペランドが与えられない場合) は、 値が自動的に一対のパーセント記号によって囲まれることに注意してください。

  補足: PostgreSQL を使っている場合は、`like` の代りに、大文字と小文字を区別しない比較のための `ilike`[16] を使う

[16] http://www.postgresql.org/docs/8.3/static/functions-matching.html#

ことも出来ます。

- or like: オペランド 2 が配列である場合に LIKE 述語が OR によって結合される以外は、 like 演算子と同じです。
- not like: 生成される条件において LIKE が NOT LIKE に置き換えられる以外は、 like 演算子と同じです。
- or not like: NOT LIKE 述語が OR によって結合される以外は、 not like 演算子と同じです。
- exists: 要求される一つだけのオペランドは、サブ・クエリを表す yii\db\Query のインスタンスでなければなりません。 これは EXISTS (sub-query) という式を構築します。
- not exists: exists 演算子と同じで、NOT EXISTS (sub-query) という式を構築します。
- >、<=、その他、二つのオペランドを取る有効な DB 演算子全て: 最初のオペランドはカラム名、第二のオペランドは値でなければなりません。 例えば、['>', 'age', 10] は age>10 を生成します。

演算子形式を使う場合、Yii は値に対して内部的にパラメータ・バインディングを使用します。 従って、文字列形式 とは対照的に、ここでは手動でパラメータを追加する必要はありません。 ただし、Yii はカラム名を決してエスケープしないことに注意して下さい。従って、変数をカラム名として渡すと、アプリケーションは SQL インジェクション攻撃に対して脆弱になります。 アプリケーションを安全に保つためには、カラム名として変数を使わないこと、または、変数を許容リストによってフィルターすることが必要です。 カラム名をユーザから取得する必要がある場合は、ガイドの データをフィルタリングする という記事を読んで下さい。 例えば、次のコードは脆弱です。

```
// 脆弱なコード:
$column = $request->get('column');
$value = $request->get('value');
$query->where([$column => $value]);
// $value は安全です。しかし、$column の名前はエンコードされません。
```

オブジェクト形式　オブジェクト形式は 2.0.14 から利用可能な、条件を定義するための最も強力でもあり、最も複雑でもある方法です。 クエリ・ビルダの上にあなた自身の抽象レイヤを構築したいときや、または独自の複雑な条件を実装したいときは、 この形式を採用する必要があります。

条件クラスのインスタンスはイミュータブルです。 条件クラスのインスタンスは条件データを保持し、条件ビルダにゲッターを提供することを唯一の目的とします。 そして、条件ビルダが、条件クラスのインスタンスに保存されたデータを SQL の式に変換するロジックを持つクラスです。

FUNCTIONS-LIKE

内部的には、上述の三つの形式は、生の SQL を構築するに先立って、暗黙のうちにオブジェクト形式に変換されます。 従って、複数の形式を単一の条件に結合することが可能です。

```
$query->andWhere(new OrCondition([
    new InCondition('type', 'in', $types),
    ['like', 'name', '%good%'],
    'disabled=false'
]))
```

演算子形式からオブジェクト形式への変換は、 演算子の名前とそれを表すクラス名を対応づける `QueryBuilder::conditionClasses` プロパティに従って行われます。

- `AND`, `OR` -> `yii\db\conditions\ConjunctionCondition`
- `NOT` -> `yii\db\conditions\NotCondition`
- `IN`, `NOT IN` -> `yii\db\conditions\InCondition`
- `BETWEEN`, `NOT BETWEEN` -> `yii\db\conditions\BetweenCondition`

等々。

オブジェクト形式を使うことによって、あなた独自の条件を作成したり、デフォルトの条件が作成される方法を変更したりすることが可能になります。 詳細は 特製の条件や式を追加する のセクションを参照して下さい。

条件を追加する `andWhere()` または `orWhere()` を使って、既存の条件に別の条件を追加することが出来ます。 これらのメソッドを複数回呼んで、複数の条件を別々に追加することが出来ます。 例えば、

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

`$search` が空でない場合は次の `WHERE` 条件 が生成されます。

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

フィルタ条件 ユーザの入力に基づいて `WHERE` の条件を構築する場合、普通は、空の入力値は無視したいものです。 例えば、ユーザ名とメール・アドレスによる検索が可能な検索フォームにおいては、 ユーザが username/email のインプット・フィールドに何も入力しなかった場合は、username/email の条件を無視したいでしょう。 `filterWhere()` メソッドを使うことによって、この目的を達することが出来ます。

```
// $username と $email はユーザの入力による
$query->filterWhere([
    'username' => $username,
```

```
    'email' => $email,
]);
```

filterWhere() と where() の唯一の違いは、 前者は ハッシュ形式 の条件において提供された空の値を無視する、という点です。 従って、$email が空で $sername がそうではない場合は、上記のコードは、結果として WHERE username=:username という SQL 条件になります。

> 情報: 値が空であると見なされるのは、null、空の配列、空の文字列、または空白のみを含む文字列である場合です。

andWhere() または orWhere() と同じように、 andFilterWhere() または orFilterWhere() を使って、 既存の条件に別のフィルタ条件を追加することも出来ます。

さらに加えて、値の方に含まれている比較演算子を適切に判断してくれる yii\db\Query::andFilterCompare() があります。

```
$query->andFilterCompare('name', 'John Doe');
$query->andFilterCompare('rating', '>9');
$query->andFilterCompare('value', '<=100');
```

演算子を明示的に指定することも可能です。

```
$query->andFilterCompare('name', 'Doe', 'like');
```

Yii 2.0.11 以降には、HAVING の条件のためにも、同様のメソッドがあります。

- filterHaving()
- andFilterHaving()
- orFilterHaving()

orderBy()

orderBy() メソッドは SQL クエリの ORDER BY 句を指定します。例えば、

```
// ... ORDER BY `id` ASC, `name` DESC
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

上記のコードにおいて、配列のキーはカラム名であり、配列の値は並べ替えの方向です。 PHP の定数 SORT_ASC は昇順、SORT_DESC は降順を指定するものです。

ORDER BY が単純なカラム名だけを含む場合は、生の SQL 文を書くときにするように、文字列を使って指定することが出来ます。 例えば、

```
$query->orderBy('id ASC, name DESC');
```

> 補足: ORDER BY が何らかの DB 式を含む場合は、配列形式を使わなければなりません。

addOrderBy() を呼んで、'ORDER BY' 句にカラムを追加することが出来ます。 例えば、

```
$query->orderBy('id ASC')
    ->addOrderBy('name DESC');
```

groupBy()

groupBy() メソッドは SQL クエリの GROUP BY 句を指定します。例えば、

```
// ... GROUP BY `id`, `status`
$query->groupBy(['id', 'status']);
```

GROUP BY が単純なカラム名だけを含む場合は、生の SQL 文を書くときにするように、文字列を使って指定することが出来ます。 例えば、

```
$query->groupBy('id, status');
```

> 補足: GROUP BY が何らかの DB 式を含む場合は、配列形式を使わなければなりません。

addGroupBy() を呼んで、GROUP BY 句にカラムを追加することが出来ます。 例えば、

```
$query->groupBy(['id', 'status'])
    ->addGroupBy('age');
```

having()

having() メソッドは SQL クエリの HAVING 句を指定します。 このメソッドが取る条件は、where() と同じ方法で指定することが出来ます。例えば、

```
// ... HAVING `status` = 1
$query->having(['status' => 1]);
```

条件を指定する方法の詳細については、where() のドキュメントを参照してください。

andHaving() または orHaving() を呼んで、HAVING 句に条件を追加することが出来ます。 例えば、

```
// ... HAVING (`status` = 1) AND (`age` > 30)
$query->having(['status' => 1])
    ->andHaving(['>', 'age', 30]);
```

limit() と offset()

limit() と offset() のメソッドは、SQL クエリの LIMIT 句と OFFSET 句を指定します。 例えば、

```
// ... LIMIT 10 OFFSET 20
$query->limit(10)->offset(20);
```

無効な上限やオフセット (例えば、負の数) を指定した場合は、無視されます。

> 情報: LIMIT と OFFSET をサポートしていない DBMS (例えば MSSQL) に対しては、 クエリ・ビルダが LIMIT/OFFSET の振る舞いをエミュレートする SQL 文を生成します。

join()

join() メソッドは SQL クエリの JOIN 句を指定します。例えば、
‘php
// ... LEFT JOIN post ON post.user_id = user.id
$query->join(’LEFT JOIN’, ‘post’, ‘post.user_id = user.id’);
‘

join() メソッドは、四つのパラメータを取ります。
- $type: 結合のタイプ、例えば、’INNER JOIN’、’LEFT JOIN’。
- $table: 結合されるテーブルの名前。
- $on: オプション。結合条件、すなわち、ON 句。
条件の指定方法の詳細については、where() を参照してください。
カラムに基づく条件を指定する場合は、配列記法は使えないことに注意してください。
例えば、[’user.id’ => ’comment.userId’] は、user の id が ’comment.userId’ という文字列でなければならない、という条件に帰結します。
配列記法ではなく文字列記法を使って、’user.id = comment.userId’ という条件を指定しなければなりません。
- $params: オプション。結合条件にバインドされるパラメータ。

INNER JOIN、LEFT JOIN および RIGHT JOIN を指定するためには、それぞれ、次のショートカット・メソッドを使うことが出来ます。

- innerJoin()
- leftJoin()
- rightJoin()

例えば、

```
$query->leftJoin('post', 'post.user_id = user.id');
```

複数のテーブルを結合するためには、テーブルごとに一回ずつ、上記の結合メソッドを複数回呼び出します。

テーブルを結合することに加えて、サブ・クエリを結合することも出来ます。 そうするためには、結合されるべきサブ・クエリを yii\db\Query オブジェクトとして指定します。例えば、

```
$subQuery = (new \yii\db\Query())->from('post');
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

この場合、サブ・クエリを配列に入れて、配列のキーを使ってエイリアスを指定しなければなりません。

union()

union() メソッドは SQL クエリの UNION 句を指定します。例えば、

```
$query1 = (new \yii\db\Query())
    ->select("id, category_id AS type, name")
    ->from('post')
    ->limit(10);

$query2 = (new \yii\db\Query())
    ->select('id, type, name')
    ->from('user')
    ->limit(10);

$query1->union($query2);
```

union() を複数回呼んで、UNION 句をさらに追加することが出来ます。

withQuery()

withQuery() メソッドは SQL クエリの WITH プレフィックスを指定するものです。サブクエリの代りに WITH を使うと読みやすさを向上させ、ユニークな機能(再帰 CTE)を利用することが出来ます。詳細は modern-sql[17] を参照して下さい。例えば、次のクエリは admin の持つ権限をその子も含めて全て再帰的に取得します。

```
$initialQuery = (new \yii\db\Query())
    ->select(['parent', 'child'])
    ->from(['aic' => 'auth_item_child'])
    ->where(['parent' => 'admin']);

$recursiveQuery = (new \yii\db\Query())
    ->select(['aic.parent', 'aic.child'])
    ->from(['aic' => 'auth_item_child'])
    ->innerJoin('t1', 't1.child = aic.parent');

$mainQuery = (new \yii\db\Query())
    ->select(['parent', 'child'])
    ->from('t1')
    ->withQuery($initialQuery->union($recursiveQuery), 't1', true);
```

withQuery() を複数回呼び出してさらなる CTE をメイン・クエリに追加することが出来ます。クエリはアタッチされたのと同じ順序でプリペンドされます。クエリのうちの一つが再帰的である場合は CTE 全体が再帰的になります。

6.2.2 クエリ・メソッド

yii\db\Query は、さまざまな目的のクエリのために、一揃いのメソッドを提供しています。

[17] https://modern-sql.com/feature/with

- all(): 各行を「名前-値」のペアの連想配列とする、結果の行の配列を返す。
- one(): 結果の最初の行を返す。
- column(): 結果の最初のカラムを返す。
- scalar(): 結果の最初の行の最初のカラムにあるスカラ値を返す。
- exists(): クエリが結果を含むか否かを示す値を返す。
- count(): COUNT クエリの結果を返す。
- その他の集計クエリ、すなわち、sum($q), average($q), max($q), min($q). これらのメソッドでは、$q パラメータは必須であり、カラム名または DB 式とすることが出来る。

例えば、

```
// SELECT ‘id‘, ‘email‘ FROM ‘user‘
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->all();

// SELECT * FROM ‘user‘ WHERE ‘username‘ LIKE ‘%test%‘
$row = (new \yii\db\Query())
    ->from('user')
    ->where(['like', 'username', 'test'])
    ->one();
```

> 補足: one() メソッドはクエリ結果の最初の行だけを返します。このメソッドは LIMIT 1 を生成された SQL 文に追加しません。このことは、クエリが一つまたは少数の行しか返さないことが判っている場合 (例えば、何らかのプライマリ・キーでクエリを発行する場合) は問題になりませんし、むしろ好ましいことです。 しかし、クエリ結果が多数のデータ行になる可能性がある場合は、パフォーマンスを向上させるために、明示的に limit(1) を呼ぶべきです。例えば、 (new \yii\db\Query())->from('user')->limit(1)->one()

上記のメソッドの全ては、オプションで、DB クエリの実行に使用されるべき DB 接続 を表す $db パラメータを取ることが出来ます。 このパラメータを省略した場合は、DB 接続として db アプリケーション・コンポーネント が使用されます。 次に count() クエリ・メソッドを使う例をもう一つ挙げます。

```
// 実行される SQL: SELECT COUNT(*) FROM ‘user‘ WHERE ‘last_name‘=:last_name
$count = (new \yii\db\Query())
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->count();
```

あなたが yii\db\Query のクエリ・メソッドを呼び出すと、実際には、内部的に次の仕事がなされます。

- yii\db\QueryBuilder を呼んで、yii\db\Query の現在の構成に基づいた SQL 文を生成する。
- 生成された SQL 文で yii\db\Command オブジェクトを作成する。
- yii\db\Command のクエリ・メソッド (例えば queryAll()) を呼んで、SQL 文を実行し、データを取得する。

場合によっては、yii\db\Query オブジェクトから構築された SQL 文を調べたり使ったりしたいことがあるでしょう。 次のコードを使って、その目的を達することが出来ます。

```
$command = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->createCommand();

// SQL 文を表示する
echo $command->sql;
// バインドされるパラメータを表示する
print_r($command->params);

// クエリ結果の全ての行を返す
$rows = $command->queryAll();
```

6.2.3 クエリ結果をインデックスする

all() を呼ぶと、結果の行は連続した整数でインデックスされた配列で返されます。 場合によっては、違う方法でインデックスしたいことがあるでしょう。 例えば、特定のカラムの値や、何らかの式の値によってインデックスするなどです。 この目的は、all() の前に indexBy() を呼ぶことによって達成することが出来ます。例えば、

```
// [100 => ['id' => 100, 'username' => '...', ...], 101 => [...], 103 =>
    [...], ...] を返
    す
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

式の値によってインデックスするためには、indexBy() メソッドに無名関数を渡します。

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

この無名関数は、現在の行データを含む `$row` というパラメータを取り、現在の行のインデックス値として使われるスカラ値を返さなくてはなりません。

> 補足: `groupBy()` や `orderBy()` のようなクエリ・メソッドが SQL に変換されてクエリの一部となるのとは対照的に、このメソッドはデータベースからデータが取得された後で動作します。 このことは、クエリの SELECT に含まれるカラム名だけを使うことが出来る、ということを意味します。 また、テーブル・プレフィックスを付けてカラムを選択した場合、例えば `customer.id` を選択した場合は、 リザルトセットのカラム名は `id` しか含みませんので、テーブルプレフィックス無しで `->indexBy('id')` と呼ぶ必要があります。

6.2.4 バッチ・クエリ

大量のデータを扱う場合は、`yii\db\Query::all()` のようなメソッドは適していません。 なぜなら、それらのメソッドは、クエリの結果全てをクライアントのメモリに読み込むことを必要とするためです。 この問題を解決するために、Yii はバッチ・クエリのサポートを提供しています。クエリ結果はサーバに保持し、 クライアントはカーソルを利用して1回に1バッチずつ結果セットを反復取得するのです。

> 警告: MySQL のバッチ・クエリの実装には既知の制約と回避策があります。下記を参照して下さい。

バッチ・クエリは次のようにして使うことが出来ます。

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users は user テーブルから取得した 100 以下の行の配列
}

// または、一行ずつ反復する場合は
foreach ($query->each() as $user) {
    // データはサーバから 100 行のバッチで取得される
    // しかし $user は user テーブルの一つの行を表す
}
```

`yii\db\Query::batch()` メソッドと `yii\db\Query::each()` メソッドは `yii\db\BatchQueryResult` オブジェクトを返します。このオブジェクトは `Iterator` インタフェイスを実装しており、従って、`foreach` 構文の中で使うことが出来ます。 初回の反復の際に、データベースに対する SQL

クエリが作成されます。データは、その後、反復のたびにバッチ・モードで取得されます。 デフォルトでは、バッチ・サイズは 100 であり、各バッチにおいて 100 行のデータが取得されます。 バッチ・サイズは、batch() または each() メソッドに最初のパラメータを渡すことによって変更することが出来ます。

yii\db\Query::all() とは対照的に、バッチ・クエリは一度に 100 行のデータしかメモリに読み込みません。

yii\db\Query::indexBy() によって、いずれかのカラムでクエリ結果をインデックスするように指定している場合でも、 バッチ・クエリは正しいインデックスを保持します。

例えば、

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users は "username" カラムでインデックスされている
}

foreach ($query->each() as $username => $user) {
    // ...
}
```

MySQL におけるバッチ・クエリの制約　MySQL のバッチ・クエリの実装は PDO ドライバのライブラリに依存しています。デフォルトでは、MySQL のクエリは バッファ・モード[18] で実行されます。 このことが、カーソルを使ってデータを取得する目的を挫折させます。というのは、バッファ・モードでは、 ドライバによって結果セット全体がクライアントのメモリに読み込まれることを防止できないからです。

> 補足: libmysqlclient が使われている場合 (PHP5 ではそれが普通ですが) は、 結果セットに使用されたメモリは PHP のメモリ使用量としてカウントされません。そのため、一見、バッチ・クエリが正しく動作するように見えますが、 実際には、データ・セット全体がクライアントのメモリに読み込まれて、クライアントのメモリを使い果たす可能性があります。

バッファ・モードを無効化してクライアントのメモリ要求量を削減するためには、PDO 接続のプロパティ PDO::MYSQL_ATTR_USE_BUFFERED_QUERY を false に設定しなければなりません。しかし、そうすると、 データ・セット全体を取得するまでは、同じ接続を通じては別のクエリを実行できなくなります。これによって ActiveRecord が必要に応じてテーブル・スキーマを取得するためのクエリを実行できなくなる可能性があります。 これ

[18] https://secure.php.net/manual/ja/mysqlinfo.concepts.buffering.php

が問題にならない場合 (テーブル・スキーマが既にキャッシュされている場合) は、元の接続を非バッファ・モードに切り替えて、 バッチ・クエリを実行した後に元に戻すということが可能です。

```
Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false)
    ;

// バッチ・クエリを実行

Yii::$app->db->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, true);
```

> 補足: MyISAM の場合は、バッチ・クエリが継続している間、テーブルがロックされて、 他の接続からの書き込みアクセスが遅延または拒絶されることがあります。非バッファ・モードのクエリを使う場合は、 カーソルを開いている時間を可能な限り短くするように努めて下さい。

スキーマがキャッシュされていない場合、またはバッチ・クエリを処理している間に他のクエリを走らせる必要がある場合は、 独立した非バッファ・モードのデータベース接続を作成することが出来ます。

```
$unbufferedDb = new \yii\db\Connection([
    'dsn' => Yii::$app->db->dsn,
    'username' => Yii::$app->db->username,
    'password' => Yii::$app->db->password,
    'charset' => Yii::$app->db->charset,
]);
$unbufferedDb->open();
$unbufferedDb->pdo->setAttribute(\PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false)
    ;
```

`$unbufferedDb` が `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY` が `false` であること以外は、 元のバッファ・モードの `$db` と同じ PDO 属性を持つことを保証したい場合は、 `$db` のディープ・コピー[19] をしてから、手動で false に設定することを考慮して下さい。

そして、クエリは普通に作成します。新しい接続を使ってバッチ・クエリを走らせ、結果をバッチで取得、 または一つずつ取得します。

```
// データを 1000 のバッチで取得
foreach ($query->batch(1000, $unbufferedDb) as $users) {
    // ...
}


// データは 1000 のバッチでサーバから取得されるが、一つずつ反復処理される
foreach ($query->each(1000, $unbufferedDb) as $user) {
    // ...
}
```

[19] https://github.com/yiisoft/yii2/issues/8420#issuecomment-301423833

結果セットが全て取得されて接続が必要なくなったら、接続を閉じることが出来ます。

```
$unbufferedDb->close();
```

> 補足: 非バッファ・モードのクエリは PHP 側でのメモリ消費は少なくなりますが、MySQL サーバの負荷を増加させ得ます。 特別に巨大なデータに対するアプリの動作については、あなた自身のコードを設計することが推奨されます。 例えば、整数のキーで範囲を分割して、非バッファ・モードのクエリでループする[20] など。

特製の条件や式を追加する

条件 – オブジェクト形式 のセクションで触れたように、特製の条件クラスを作成することが可能です。 例として、特定のカラムが一定の値より小さいことをチェックする条件を作ってみましょう。 演算子形式を使えば、それは次のようになるでしょう。

```
[
    'and',
    '>', 'posts', $minLimit,
    '>', 'comments', $minLimit,
    '>', 'reactions', $minLimit,
    '>', 'subscriptions', $minLimit
]
```

このような条件を一度に適用できたら良いですね。一つのクエリの中で複数回使われる場合には、最適化の効果が大きいでしょう。 特製の条件オブジェクトを作って、それを実証しましょう。

Yii には、条件を表現するクラスを特徴付ける `ConditionInterface` があります。 このインタフェイスは、配列形式から条件を作ることを可能にするための `fromArrayDefinition()` メソッドを実装することを要求します。 あなたがそれを必要としない場合は、例外を投げるだけのメソッドとして実装しても構いません。

特製の条件クラスを作るのですから、私たちの仕事に最適な API を構築すれば良いのです。

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
{
    private $columns;
    private $value;

    /**
     * @param string[] $columns $value よりも大きくなければならないカラムの配
    列
```

[20] https://github.com/yiisoft/yii2/issues/8420#issuecomment-296109257

```
     * @param mixed $value 各カラムと比較する値
     */
    public function __construct(array $columns, $value)
    {
        $this->columns = $columns;
        $this->value = $value;
    }

    public static function fromArrayDefinition($operator, $operands)
    {
        throw new InvalidArgumentException('未実装、あとでやる');
    }

    public function getColumns() { return $this->columns; }
    public function getValue() { return $this->vaule; }
}
```

これで条件オブジェクトを作ることが出来ます。

```
$conditon = new AllGreaterCondition(['col1', 'col2'], 42);
```

しかし QueryBuilder は、このオブジェクトから SQL 条件式を作る方法を知りません。 次に、この条件に対する式ビルダを作成する必要があります。 式ビルダは build() メソッドを提供する yii\db\ExpressionBuilderInterface を実装しなければいけません。

```
namespace app\db\conditions;

class AllGreaterConditionBuilder implements \yii\db\
    ExpressionBuilderInterface
{
    use \yii\db\ExpressionBuilderTrait; // コンストラクタと ‘queryBuilder‘ プ
    ロパティを含む。

    /**
     * @param ExpressionInterface $condition ビルドすべき条件
     * @param array $params バインディング・パラメータ
     * @return AllGreaterCondition
     */
    public function build(ExpressionInterface $expression, array &$params =
    [])
    {
        $value = $condition->getValue();

        $conditions = [];
        foreach ($expression->getColumns() as $column) {
            $conditions[] = new SimpleCondition($column, '>', $value);
        }

        return $this->queryBuilder->buildCondition(new AndCondition(
    $conditions), $params);
    }
}
```

後は、単に QueryBuilder に私たちの新しい条件について知らせるだけです – 条件のマッピングを expressionBuilders 配列に追加します。次のように、アプリケーション構成で直接に追加することが出来ます。

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' => 'app\db\conditions\
    AllGreaterConditionBuilder',
        ],
    ],
],
```

これで、私たちの新しい条件を where() で使用することが出来るようになりました。

```
$query->andWhere(new AllGreaterCondition(['posts', 'comments', 'reactions',
    'subscriptions'], $minValue));
```

演算子形式を使って私たちの特製の条件を作成することが出来るようにしたい場合は、 演算子を QueryBuilder::conditionClasses の中で宣言しなければなりません。

```
'db' => [
    'class' => 'yii\db\mysql\Connection',
    // ...
    'queryBuilder' => [
        'expressionBuilders' => [
            'app\db\conditions\AllGreaterCondition' => 'app\db\conditions\
    AllGreaterConditionBuilder',
        ],
        'conditionClasses' => [
            'ALL>' => 'app\db\conditions\AllGreaterCondition',
        ],
    ],
],
```

そして、app\db\conditions\AllGreaterCondition の中で AllGreaterCondition::fromArrayDefinition() メソッドの本当の実装を作成します。

```
namespace app\db\conditions;

class AllGreaterCondition implements \yii\db\conditions\ConditionInterface
{
    // ... 上記の実装を参照

    public static function fromArrayDefinition($operator, $operands)
    {
        return new static($operands[0], $operands[1]);
    }
}
```

これ以降は、私たちの特製の条件をより短い演算子形式を使って作成することが出来ます。

```
$query->andWhere(['ALL>', ['posts', 'comments', 'reactions', 'subscriptions'
    ], $minValue]);
```

お気付きのことと思いますが、ここには二つの概念があります。Expression(式)とCondition(条件)です。 `yii\db\ExpressionInterface` は、それを構築するために `yii\db\ExpressionBuilderInterface` を実装した式ビルダクラスを必要とするオブジェクトを特徴付けるインタフェイスです。 また `yii\db\condition\ConditionInterface` は、`ExpressionInterface` を拡張して、 上述されたように配列形式の定義から作成できるオブジェクトに対して使用されるべきものですが、同様にビルダを必要とするものです。

要約すると、

- Expression(式) – データセットのためのデータ転送オブジェクトであり、最終的に何らかの SQL 文にコンパイルされる。 (演算子、文字列、配列、JSON、等)
- Condition(条件) – Expression(式) のスーパーセットで、一つの SQL 条件にコンパイルすることが可能な複数の式 (またはスカラ値)の集合。

`ExpressionInterface` を実装する独自のクラスを作成して、 データを SQL 文に変換することの複雑さを隠蔽することが出来ます。 次の記事 では、式について、さらに多くの例を学習します。

6.3 アクティブ・レコード

アクティブ・レコード[21] は、データベースに保存されているデータにアクセスするために、 オブジェクト指向のインタフェイスを提供するものです。 アクティブ・レコード・クラスはデータベース・テーブルと関連付けられます。 アクティブ・レコードのインスタンスはそのテーブルの行に対応し、アクティブ・レコードのインスタンスの 属性 がその行にある特定のカラムの値を表現します。 生の SQL 文を書く代りに、アクティブ・レコードの属性にアクセスしたり、アクティブ・レコードのメソッドを呼んだりして、 データベース・テーブルに保存さているデータにアクセスしたり、データを操作したりします。

例えば、`Customer` が `customer` テーブルに関連付けられたアクティブ・レコード・クラスであり、 `name` が `customer` テーブルのカラムであると仮定しましょう。 `customer` テーブルに新しい行を挿入するために次のコードを書くことが出来ます。

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

上記のコードは、MySQL では、次のような生の SQL 文を使うのと等価なものです。 しかし、生の SQL 文の方は、直感的でなく、間違いも生じ

[21] http://ja.wikipedia.org/wiki/Active_Record

やすく、また、別の種類のデータベースを使う場合には、互換性の問題も生じ得ます。

```
$db->createCommand('INSERT INTO `customer` (`name`) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

Yii は次のリレーショナル・データベースに対して、アクティブ・レコードのサポートを提供しています。

- MySQL 4.1 以降: `yii\db\ActiveRecord` による。
- PostgreSQL 7.3 以降: `yii\db\ActiveRecord` による。
- SQLite 2 および 3: `yii\db\ActiveRecord` による。
- Microsoft SQL Server 2008 以降: `yii\db\ActiveRecord` による。
- Oracle: `yii\db\ActiveRecord` による。
- CUBRID 9.3 以降: `yii\db\ActiveRecord` による。(cubrid PDO 拡張の バグ[22] のために、値を引用符で囲む機能が動作しません。そのため、サーバだけでなくクライアントも CUBRID 9.3 が必要になります)
- Sphinx: `yii\sphinx\ActiveRecord` による。`yii2-sphinx` エクステンションが必要。
- ElasticSearch: `yii\elasticsearch\ActiveRecord` による。`yii2-elasticsearch` エクステンションが必要。

これらに加えて、Yii は次の NoSQL データベースに対しても、アクティブ・レコードの使用をサポートしています。

- Redis 2.6.12 以降: `yii\redis\ActiveRecord` による。`yii2-redis` エクステンションが必要。
- MongoDB 1.3.0 以降: `yii\mongodb\ActiveRecord` による。`yii2-mongodb` エクステンションが必要。

このチュートリアルでは、主としてリレーショナル・データベースのためのアクティブ・レコードの使用方法を説明します。 しかし、ここで説明するほとんどの内容は NoSQL データベースのためのアクティブ・レコードにも適用することが出来るものです。

6.3.1 アクティブ・レコード・クラスを宣言する

まずは、`yii\db\ActiveRecord` を拡張してアクティブ・レコード・クラスを宣言するところから始めましょう。

テーブル名を設定する

デフォルトでは、すべてのアクティブ・レコード・クラスはデータベース・テーブルと関連付けられます。 `tableName()` メソッドが、クラス名を `yii\helpers\Inflector::camel2id()` によって変換して、テーブル名を返します。 テーブル名がこの規約に従っていない場合は、このメソッドをオーバライドすることが出来ます。

[22] http://jira.cubrid.org/browse/APIS-658

同時に、デフォルトの tablePrefix を適用することも可能です。 例えば、tablePrefix が tbl_ である場合は、Customer は tbl_customer になり、OrderItem はtbl_order_item になります。

テーブル名が {{%TableName}} という形式で与えられた場合は、パーセント記号 % がテーブルプレフィックスに置き換えられます。 例えば、{{%post}} は {{tbl_post}} となります。 テーブル名を囲む二重波括弧は、テーブル名を囲む引用符号 となります。

次の例では、customer というデータベース・テーブルのための Customer という名前のアクティブ・レコード・クラスを宣言しています。

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string このアクティブ・レコード・クラスと関連付けられるテーブル
    の名前
     */
    public static function tableName()
    {
        return '{{customer}}';
    }
}
```

アクティブ・レコードは「モデル」と呼ばれる

アクティブ・レコードのインスタンスは モデル であると見なされます。この理由により、私たちは通常 app\models 名前空間 (あるいはモデル・クラスを保管するための他の名前空間) の下にアクティブ・レコード・クラスを置きます。

yii\db\ActiveRecord は yii\base\Model から拡張していますので、属性、検証規則、データのシリアル化など、 モデル が持つ 全ての 機能を継承しています。

6.3.2 データベースに接続する

デフォルトでは、アクティブ・レコードは、db アプリケーション・コンポーネント を DB 接続 として使用して、データベースのデータにアクセスしたり操作したりします。 データベース・アクセス・オブジェクト で説明したように、次のようにして、アプリケーションの構成情報ファイルの中で db コンポーネントを構成することが出来ます。

```
return [
    'components' => [
```

```
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

db コンポーネントとは異なるデータベース接続を使いたい場合は、getDb() メソッドをオーバーライドしなければなりません。

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // "db2" アプリケーション・コンポーネントを使用
        return \Yii::$app->db2;
    }
}
```

6.3.3 データをクエリする

アクティブ・レコード・クラスを宣言した後、それを使って対応するデータベース・テーブルからデータをクエリすることが出来ます。 このプロセスは通常次の三つのステップを踏みます。

1. yii\db\ActiveRecord::find() メソッドを呼んで、新しいクエリ・オブジェクトを作成する。
2. クエリ構築メソッド を呼んで、クエリ・オブジェクトを構築する。
3. クエリ・メソッド を呼んで、アクティブ・レコードのインスタンスの形でデータを取得する。

ご覧のように、このプロセスは クエリ・ビルダ による手続きと非常によく似ています。 唯一の違いは、new 演算子を使ってクエリ・オブジェクトを生成する代りに、yii\db\ActiveQuery クラスであるクエリ・オブジェクトを返す yii\db\ActiveRecord::find() を呼ぶ、という点です。

以下の例は、アクティブ・クエリを使ってデータをクエリする方法を示すものです。

```
// ID が 123 である一人の顧客を返す
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// アクティブな全ての顧客を返して、ID によって並べる
```

```
// SELECT * FROM ‘customer‘ WHERE ‘status‘ = 1 ORDER BY ‘id‘
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// アクティブな顧客の数を返す
// SELECT COUNT(*) FROM ‘customer‘ WHERE ‘status‘ = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// 全ての顧客を顧客によってインデックスされた配列として返すID
// SELECT * FROM ‘customer‘
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

上記において、$customer は Customer オブジェクトであり、$customers は Customer オブジェクトの配列です。 全てこれらには customer テーブルから取得されたデータが投入されます。

> 情報: yii\db\ActiveQuery は yii\db\Query から拡張しているため、クエリ・ビルダ のセクションで説明されたクエリ構築メソッドとクエリ・メソッドの 全て を使うことが出来ます。

プライマリ・キーの値や一群のカラムの値でクエリをすることはよく行われる仕事ですので、Yii はこの目的のために、 二つのショートカット・メソッドを提供しています。

- yii\db\ActiveRecord::findOne(): クエリ結果の最初の行を一つのアクティブ・レコード・インスタンスに投入して返す。
- yii\db\ActiveRecord::findAll(): 全ての クエリ結果をアクティブ・レコード・インスタンスの配列に投入して返す。

どちらのメソッドも、次のパラメータ形式のどれかを取ることが出来ます。

- スカラ値: 値は検索時に求められるプライマリ・キーの値として扱われます。 Yii は、データベースのスキーマ情報を読んで、どのカラムがプライマリ・キーのカラムであるかを自動的に判断します。
- スカラ値の配列: 配列は検索時に求められるプライマリ・キーの値の配列として扱われます。
- 連想配列: キーはカラム名であり、値は検索時に求められる対応するカラムの値です。 詳細については、ハッシュ形式 を参照してください。

次のコードは、これらのメソッドの使用方法を示すものです。

```
// ID が 123 である一人の顧客を返す
// SELECT * FROM ‘customer‘ WHERE ‘id‘ = 123
$customer = Customer::findOne(123);
```

```
// ID が 100, 101, 123, 124 のどれかである顧客を全て返す
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// ID が 123 であるアクティブな顧客を返す
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// アクティブでない全ての顧客を返す
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

> 警告: これらのメソッドにユーザ入力を渡す必要がある場合は、入力値がスカラ値であること、または、 入力値が配列形式の条件である場合は配列の構造が外部から変更され得ないことを保証して下さい。
>
> ```
> // yii\web\Controller が $id はスカラ値であることを保証しています
> public function actionView($id)
> {
>     $model = Post::findOne($id);
>     // ...
> }
>
> // 検索するカラムを明示的に指定する場合。ここでは、どんなスカラ値または配列を渡しても、単一のレコードを発見する結果になります。
> $model = Post::findOne(['id' => Yii::$app->request->get('id')]);
>
> // 次のコードを使用してはいけません! 任意のカラムの値による検索が可能な配列形式の条件を挿入される可能性があります!
> $model = Post::findOne(Yii::$app->request->get('id'));
> ```

> 補足: `yii\db\ActiveRecord::findOne()` も `yii\db\ActiveQuery::one()` も、生成される SQL 文に `LIMIT 1` を追加しません。 あなたのクエリが多数のデータ行を返すかもしれない場合は、パフォーマンスを向上させるために、`limit(1)` を明示的に呼ぶべきです。 例えば `Customer::find()->limit(1)->one()` のように。

クエリ構築メソッドを使う以外に、生の SQL を書いてデータをクエリして結果をアクティブ・レコード・オブジェクトに投入することも出来ます。 そうするためには `yii\db\ActiveRecord::findBySql()` メソッドを呼ぶことが出来ます。

```
// アクティブでない全ての顧客を返す
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, [':status' => Customer::
    STATUS_INACTIVE])->all();
```

findBySql() を呼んだ後は、追加でクエリ構築メソッドを呼び出してはいけません。 呼んでも無視されます。

6.3.4 データにアクセスする

既に述べたように、データベースから取得されたデータはアクティブ・レコードのインスタンスに投入されます。 そして、クエリ結果の各行がアクティブ・レコードの一つのインスタンスに対応します。 アクティブ・レコード・インスタンスの属性にアクセスすることによって、カラムの値にアクセスすることが出来ます。例えば、

```
// "id" と "email" は "customer" テーブルのカラム名
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

> 補足: アクティブ・レコードの属性の名前は、関連付けられたテーブルのカラムの名前に従って、大文字と小文字を区別して名付けられます。 Yii は、関連付けられたテーブルの全てのカラムに対して、アクティブ・レコードの属性を自動的に定義します。 これらの属性は、すべて、再宣言してはいけません。

アクティブ・レコードの属性はテーブルのカラムに従って命名されるため、 テーブルのカラム名がアンダースコアで単語を分ける方法で命名されている場合は、 $customer->first_name のような属性名を使って PHP コードを書くことになります。 コード・スタイルの一貫性が気になるのであれば、テーブルのカラム名を (例えば camelCase を使う名前に) 変更しなければなりません。

データ変換

入力または表示されるデータの形式が、データベースにデータを保存するときに使われるものと異なる場合がよくあります。 例えば、データベースでは顧客の誕生日を UNIX タイムスタンプで保存している (まあ、あまり良い設計ではありませんが) けれども、ほとんどの場合において誕生日を 'YYYY/MM/DD' という形式の文字列として操作したい、というような場合です。 この目的を達するために、次のように、Customer アクティブ・レコード・クラスにおいて データ変換 メソッドを定義することが出来ます。

```
class Customer extends ActiveRecord
{
```

```
    // ...

    public function getBirthdayText()
    {
        return date('Y/m/d', $this->birthday);
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

このようにすれば、PHP コードにおいて、`$customer->birthday` にアクセスする代りに、`$customer->birthdayText` にアクセスすれば、 顧客の誕生日を `'YYYY/MM/DD'` の形式で入力および表示することが出来ます。

> ヒント: 上記は、一般にデータの変換を達成するための簡単な方法を示すためのものです。 日付の値については、Yii は、DateValidator と DatePicker ウィジェットを使用するという、より良い方法を提供しています。 DatePicker については、JUI ウィジェットのセクション で説明されています。

データを配列に取得する

データをアクティブ・レコード・オブジェクトの形で取得するのは便利であり柔軟ですが、大きなメモリ使用量を要するために、 大量のデータを取得しなければならない場合は、必ずしも望ましい方法ではありません。 そういう場合は、クエリ・メソッドを実行する前に `asArray()` を呼ぶことによって、PHP 配列を使ってデータを取得することが出来ます。

```
// すべての顧客を返す
// 各顧客は連想配列として返される
$customers = Customer::find()
    ->asArray()
    ->all();
```

> 補足: このメソッドはメモリを節約してパフォーマンスを向上させますが、低レベルの DB 抽象レイヤに近いものであり、あなたはアクティブ・レコードの機能のほとんどを失うことになります。非常に重要な違いが、カラムの値のデータ型に現れます。 アクティブ・レコード・インスタンスとしてデータを返す場合、カラムの値は実際のカラムの型に従って自動的に型キャストされます。 一方、配列としてデータを返す場合は、実際のカラムの型に関係なく、カラムの値は文字列になります。 なぜなら、何も処理をしない場合の PDO の結果は文字列だからです。

データをバッチ・モードで取得する

クエリ・ビルダ において、大量のデータをデータベースから検索する場合に、メモリ使用量を最小化するために バッチ・クエリ を使うことが出来るということを説明しました。おなじテクニックをアクティブ・レコードでも使うことが出来ます。例えば、

```
// 一度に 10 人の顧客を読み出す
foreach (Customer::find()->batch(10) as $customers) {
    // $customers は 10 以下の Customer オブジェクトの配列
}

// 一度に 10 人の顧客を読み出して、一人ずつ反復する
foreach (Customer::find()->each(10) as $customer) {
    // $customer は Customer オブジェクト
}

// イーガー・ローディングをするバッチ・クエリ
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer は 'orders' リレーションを投入された Customer オブジェクト
}
```

6.3.5 データを保存する

アクティブ・レコードを使えば、次のステップを踏んで簡単にデータをデータベースに保存することが出来ます。

1. アクティブ・レコードのインスタンスを準備する
2. アクティブ・レコードの属性に新しい値を割り当てる
3. `yii\db\ActiveRecord::save()` を呼んでデータをデータベースに保存する

例えば、

```
// 新しいデータ行を挿入する
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// 既存のデータ行を更新する
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

`save()` メソッドは、アクティブ・レコード・インスタンスの状態に従って、データ行を挿入するか、 または、更新することが出来ます。インスタンスが `new` 演算子によって新しく作成されたものである場合は、`save()` を呼び出すと、新しい行が挿入されます。インスタンスがクエ

リ・メソッドの結果である場合は、 save() を呼び出すと、そのインスタンスと関連付けられた行が更新されます。

アクティブ・レコード・インスタンスの二つの状態は、その isNewRecord プロパティの値をチェックすることによって区別することが出来ます。下記のように、このプロパティは save() によっても内部的に使用されています。

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

> ヒント: insert() または update() を直接に呼んで、 行を挿入または更新することも出来ます。

データの検証

yii\db\ActiveRecord は yii\base\Model を拡張したものですので、同じ データ検証 機能を共有しています。 rules() メソッドをオーバーライドすることによって検証規則を宣言し、 validate() メソッドを呼ぶことによってテータの検証を実行することが出来ます。

save() を呼ぶと、デフォルトでは validate() が自動的に呼ばれます。 検証が通った時だけ、実際にデータが保存されます。 検証が通らなかった時は単に false が返され、errors プロパティをチェックして検証エラー・メッセージを取得することが出来ます。

> ヒント: データが検証を必要としないことが確実である場合 (例えば、データが信頼できるソースに由来するものである場合) は、 検証をスキップするために save(false) を呼ぶことが出来ます。

一括代入

通常の モデル と同じように、アクティブ・レコードのインスタンスも 一括代入機能 を享受することが出来ます。 この機能を使うと、下記で示されているように、一つの PHP 文で、アクティブ・レコード・インスタンスの複数の属性に値を割り当てることが出来ます。 ただし、安全な属性だけが一括代入が可能であることを記憶しておいてください。

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];
```

```
$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

カウンタを更新する

データベース・テーブルのあるカラムの値を増加・減少させるのは、よくある仕事です。私たちはそのようなカラムをカウンタ・カラムと呼んでいます。 updateCounters() を使って一つまたは複数のカウンタ・カラムを更新することが出来ます。 例えば、

```
$post = Post::findOne(100);

// UPDATE ‘post‘ SET ‘view_count‘ = ‘view_count‘ + 1 WHERE ‘id‘ = 100
$post->updateCounters(['view_count' => 1]);
```

> 補足: カウンタ・カラムを更新するのに yii\db\ActiveRecord ::save() を使うと、不正確な結果になってしまう場合があります。 というのは、同じカウンタの値を読み書きする複数のリクエストによって、同一のカウンタが保存される可能性があるからです。

ダーティな属性

save() を呼んでアクティブ・レコード・インスタンスを保存すると、ダーティな属性 だけが保存されます。 属性は、DB からロードされた後、または、最後に保存された後にその値が変更されると、ダーティ であると見なされます。 ただし、データ検証は、アクティブ・レコード・インスタンスがダーティな属性を持っているかどうかに関係なく実施される ことに注意してください。

アクティブ・レコードはダーティな属性のリストを自動的に保守します。 そうするために、一つ前のバージョンの属性値を保持して、最新のバージョンと比較します。 yii\db\ActiveRecord::getDirtyAttributes() を呼ぶと、現在ダーティである属性を取得することが出来ます。 また、yii\db\ActiveRecord::markAttributeDirty() を呼んで、ある属性をダーティであると明示的にマークすることも出来ます。

最新の修正を受ける前の属性値を知りたい場合は、getOldAttributes() または getOldAttribute() を呼ぶことが出来ます。

> 補足: 新旧の値は === 演算子を使って比較されるため、同じ値を持っていても型が違うとダーティであると見なされます。 このことは、モデルが HTML フォームからユーザの入力を受け取るときにしばしば生じます。 HTML フォームでは全ての値が文字列として表現されるからです。 入力値が正しい型、

例えば整数値となることを保証するために、`['attributeName', 'filter', 'filter' => 'intval']` のように 検証フィルタ を適用することが出来ます。 このフィルタは、intval()[23], floatval()[24], boolval[25] など、PHP の全てのタイプキャスト関数で動作します。

デフォルト属性値

あなたのテーブルのカラムの中には、データベースでデフォルト値が定義されているものがあるかも知れません。 そして、場合によっては、アクティブ・レコード・インスタンスのウェブ・フォームに、そういうデフォルト値をあらかじめ投入したいことがあるでしょう。 同じデフォルト値を繰り返して書くことを避けるために、`loadDefaultValues()` を呼んで、DB で定義されたデフォルト値を対応するアクティブ・レコードの属性に投入することが出来ます。

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz には、"xyz" カラムを定義するときに宣言されたデフォルト値が割り当てられる
```

属性の型キャスト

`yii\db\ActiveRecord` は、クエリの結果を投入されるときに、データベース・テーブル・スキーマ からの情報を使って、自動的な型キャストを実行します。これによって、整数として宣言されているテーブルカラムから取得されるデータを アクティブ・レコードのインスタンスでも PHP の integer として投入し、真偽値として宣言されているデータを boolean として投入することが出来るようになっています。 しかしながら、型キャストのメカニズムには、いくつかの制約があります。

- 浮動小数点数値は変換されず、文字列として表されます。そうしないと精度が失われるおそれがあるからです。
- 整数値の変換は、あなたが使っているオペレーティング・システムの整数の大きさに依存します。具体的に言うと、 'unsigned integer' または 'big integer' として宣言されたカラムの値は、64-bit オペレーティングシステムでのみ PHP の integer に変換されます。 32-bit オペレーティングシステムでは、文字列として表されます。

属性の型キャストは、アクティブ・レコードのインスタンスにクエリの結果から値を投入するときだけしか実行されないことに注意してください。 HTTP リクエストから値をロードしたり、プロパティにアクセスして直接に値を設定したりするときには、自動的な変換は行われません。また、アクティブ・レコードのデータ保存のための SQL 文を準備する際

[23] https://secure.php.net/manual/ja/function.intval.php
[24] https://secure.php.net/manual/ja/function.floatval.php
[25] https://secure.php.net/manual/ja/function.boolval.php

にもテーブル・スキーマが使用されて、 値が正しい型でクエリにバインドされることを保証します。 しかし、アクティブ・レコードのインスタンスの属性値は保存の過程において変換されることはありません。

> ヒント: アクティブ・レコードの検証や保存の際の属性型キャストを楽にするために `yii\behaviors\AttributeTypecastBehavior` を使うことが出来ます。

2.0.14 以降、Yii のアクティブ・レコードは、JSON や多次元配列のような複雑な型をサポートしています。

MySQL および PostgreSQL における JSON データが取得された後、JSON カラムの値は標準的な JSON デコード規則に従って、 自動的に JSON からデコードされます。

アクティブ・レコードは、属性値を JSON カラムに保存するために `JsonExpression` オブジェクトを自動的に生成します。このオブジェクトが クエリ・ビルダ レベルで JSON 文字列にエンコードされます。

PostgreSQL における配列 データが取得された後、配列カラムの値は PgSQL 記法から自動的に `ArrayExpression` オブジェクトにデコードされます。 このオブジェクトは PHP の `ArrayAccess` インタフェイスを実装しているため、これを配列として使うこと事が出来ます。また、`->getValue()` を呼んで配列そのものを取得することも出来ます。

アクティブ・レコードは、属性値を配列カラムに保存するために `ArrayExpression` オブジェクトを生成します。このオブジェクトが クエリ・ビルダ のレベルで配列を表す PgSQL 文字列にエンコードされます。

JSON カラムに対して条件を使用することも出来ます。

```
$query->andWhere(['=', 'json', new ArrayExpression(['foo' => 'bar'])
```

式を構築するシステムについて更に学習するためには クエリ・ビルダ – 特製の条件や式を追加する という記事を参照して下さい。

複数の行を更新する

上述のメソッドは、すべて、個別のアクティブ・レコード・インスタンスに対して作用し、個別のテーブル行を挿入したり更新したりするものです。 複数の行を同時に更新するためには、代りに、スタティックなメソッドである `updateAll()` を呼ばなければなりません。

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',
    '@example.com']);
```

同様に、`updateAllCounters()` を呼んで、 複数の行のカウンタカラムを同時に更新することが出来ます。

```
// UPDATE ‘customer‘ SET ‘age‘ = ‘age‘ + 1
Customer::updateAllCounters(['age' => 1]);
```

6.3.6 データを削除する

一行のデータを削除するためには、最初にその行に対応するアクティブ・レコード・インスタンスを取得して、 次に yii\db\ActiveRecord::delete() メソッドを呼びます。

```
$customer = Customer::findOne(123);
$customer->delete();
```

yii\db\ActiveRecord::deleteAll() を呼んで、複数またはすべてのデータ行を削除することが出来ます。例えば、

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

> 補足: deleteAll() を呼ぶときは、十分に注意深くしてください。 なぜなら、条件の指定を間違うと、あなたのテーブルからすべてのデータを完全に消し去ってしまうことになるからです。

6.3.7 アクティブ・レコードのライフサイクル

アクティブ・レコードがさまざまな目的で使用される場合のそれぞれのライフサイクルを理解しておくことは重要なことです。 それぞれのライフサイクルにおいては、特定の一続きのメソッドが呼び出されます。 そして、これらのメソッドをオーバーライドして、ライフサイクルをカスタマイズするチャンスを得ることが出来ます。 また、ライフサイクルの中でトリガされる特定のアクティブ・レコード・イベントに反応して、あなたのカスタム・コードを挿入することも出来ます。 これらのイベントが特に役に立つのは、アクティブ・レコードのライフサイクルをカスタマイズする必要のあるアクティブ・レコード・ビヘイビア を開発する際です。

次に、さまざまなアクティブ・レコードのライフサイクルと、 そのライフサイクルに含まれるメソッドやイベントを要約します。

新しいインスタンスのライフサイクル

new 演算子によって新しいアクティブ・レコード・インスタンスを作成する場合は、次のライフサイクルを経ます。

1. クラスのコンストラクタ。
2. init(): EVENT_INIT イベントをトリガ。

データをクエリする際のライフサイクル

クエリ・メソッド のどれか一つによってデータをクエリする場合は、 新しくデータを投入されるアクティブ・レコードは次のライフサイクルを経ます。

1. クラスのコンストラクタ。
2. init(): EVENT_INIT イベントをトリガ。
3. afterFind(): EVENT_AFTER_FIND イベントをトリガ。

データを保存する際のライフサイクル

save() を呼んでアクティブ・レコード・インスタンスを挿入または更新する場合は、 次のライフサイクルを経ます。

1. beforeValidate(): EVENT_BEFORE_VALIDATE イベントをトリガ。 このメソッドが false を返すか、yii\base\ModelEvent::$isValid が false であった場合、 残りのステップはスキップされる。
2. データ検証を実行。データ検証が失敗した場合、3 より後のステップはスキップされる。
3. afterValidate(): EVENT_AFTER_VALIDATE イベントをトリガ。
4. beforeSave(): EVENT_BEFORE_INSERT または EVENT_BEFORE_UPDATE イベントをトリガ。 このメソッドが false を返すか、yii\base\ModelEvent::$isValid が false であった場合、 残りのステップはスキップされる。
5. 実際のデータの挿入または更新を実行。
6. afterSave(): EVENT_AFTER_INSERT または EVENT_AFTER_UPDATE イベントをトリガ。

データを削除する際のライフサイクル

delete() を呼んでアクティブ・レコード・インスタンスを削除する際は、 次のライフサイクルを経ます。

1. beforeDelete(): EVENT_BEFORE_DELETE イベントをトリガ。このメソッドが false を返すか、yii\base\ModelEvent::$isValid が false であった場合は、 残りのステップはスキップされる。
2. 実際のデータの削除を実行。
3. afterDelete(): EVENT_AFTER_DELETE イベントをトリガ。

補足: 次のメソッドを呼んだ場合は、いずれの場合も、上記のライフサイクルのどれかを開始させることはありません。これらのメソッドは、レコード単位ではなく、データベース上で直接に動作するためです。

- yii\db\ActiveRecord::updateAll()
- yii\db\ActiveRecord::deleteAll()
- yii\db\ActiveRecord::updateCounters()
- yii\db\ActiveRecord::updateAllCounters()

データをリフレッシュする際のライフサイクル

refresh() を呼んでアクティブ・レコード・インスタンスをリフレッシュする際は、リフレッシュが成功してメソッドが true を返すと EVENT_AFTER_REFRESH イベントがトリガされます。

6.3.8 トランザクションを扱う

アクティブ・レコードを扱う際には、二つの方法で トランザクション を処理することができます。

最初の方法は、次に示すように、アクティブ・レコードのメソッドの呼び出しを明示的にトランザクションのブロックで囲む方法です。

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ... 他の DB 操作 ...
});

// あるいは、別の方法

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ... 他の DB 操作 ...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

補足: 上記のコードでは、PHP 5.x と PHP 7.x との互換性のために、二つの catch ブロックを持っています。 \Exception は

> PHP 7.0 以降では、\Throwable インタフェイス[26] を実装しています。 従って、あなたのアプリケーションが PHP 7.0 以上しか使わない場合は、\Exception の部分を省略することが出来ます。

第二の方法は、トランザクションのサポートが必要な DB 操作を yii\db\ActiveRecord::transactions() メソッドに列挙するという方法です。

```
class Post extends \yii\db\ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // 上は次と等価
            // 'api' => self::OP_ALL,
        ];
    }
}
```

yii\db\ActiveRecord::transactions() メソッドが返す配列では、キーは シナリオ の名前であり、 値はトランザクションで囲まれるべき操作でなくてはなりません。 いろいろな DB 操作を参照するのには、次の定数を使わなければなりません。

- OP_INSERT: insert() によって実行される挿入の操作。
- OP_UPDATE: update() によって実行される更新の操作。
- OP_DELETE: delete() によって実行される削除の操作。

複数の操作を示すためには、| を使って上記の定数を連結してください。ショートカット定数 OP_ALL を使って、上記の三つの操作すべてを示すことも出来ます。

このメソッドを使って生成されたトランザクションは、beforeSave() を呼ぶ前に開始され、 afterSave() を実行した後にコミットされます。

6.3.9 楽観的ロック

楽観的ロックは、一つのデータ行が複数のユーザによって更新されるときに発生しうる衝突を回避するための方法です。 例えば、ユーザ A とユーザ B が 同時に同じ wiki 記事を編集しており、ユーザ A が自分の編集結果を保存した後に、 ユーザ B も自分の編集結果を保存しようとして「保存」ボタンをクリックする場合を考えてください。 ユーザ B は、実際には古くなったバージョンの記事に対する操作をしようとしていますので、彼が記事を保存するのを防止し、 彼に何らかのヒント・メッセージを表示する方法があることが望まれます。

楽観的ロックは、あるカラムを使って各行のバージョン番号を記録するという方法によって、上記の問題を解決します。 古くなったバージョ

[26] https://secure.php.net/manual/ja/class.throwable.php

ン番号とともに行を保存しようとすると、yii\db\StaleObjectException 例外が投げられて、 行が保存されるのが防止されます。 楽観的ロックは、 yii\db\ActiveRecord::update() または yii\db\ActiveRecord::delete() メソッドを使って既存の行を更新または削除しようとする場合にだけサポートされます。

楽観的ロックを使用するためには、次のようにします。

1. アクティブ・レコード・クラスと関連付けられている DB テーブルに、各行のバージョン番号を保存するカラムを作成します。 カラムは長倍精度整数 (big integer) タイプでなければなりません (MySQL では BIGINT DEFAULT 0 です)。

2. yii\db\ActiveRecord::optimisticLock() メソッドをオーバーライドして、このカラムの名前を返すようにします。

3. あなたのモデル・クラスの中で OptimisticLockBehavior を実装し、受信したリクエストからその値を自動的に解析できるようにします。 OptimisticLockBehavior が検証を処理すべきですので、バージョンの属性は検証規則から削除します。

4. ユーザ入力を収集するウェブフォームに、更新されるレコードの現在のバージョン番号を保持する隠しフィールドを追加します。

5. アクティブ・レコードを使って行の更新を行うコントローラ・アクションにおいて、yii\db\StaleObjectException 例外を捕捉して、衝突を解決するために必要なビジネス・ロジック (例えば、変更をマージしたり、データの陳腐化を知らせたり) を実装します。

例えば、バージョン番号のカラムが version と名付けられているとすると、 次のようなコードによって楽観的ロックを実装することが出来ます。

```
// ------ ビューのコード -------

use yii\helpers\Html;

// ... 他の入力フィールド
echo Html::activeHiddenInput($model, 'version');


// ------ コントローラのコード -------

use yii\db\StaleObjectException;

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    try {
```

```
        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } catch (StaleObjectException $e) {
        // 衝突を解決するロジック
    }
}

// ------ モデルのコード -------

use yii\behaviors\OptimisticLockBehavior;

public function behaviors()
{
    return [
        OptimisticLockBehavior::className(),
    ];
}
```

補足: OptimisticLockBehavior は、ユーザが正しいバージョン番号を送信したときにだけ レコードが保存されるという事を保証します。そして、そのために、getBodyParam() の結果を直接に解析します。 そこで、あなたのモデル・クラスを拡張して、親モデルで第2段階を行い、ビヘイビアのアタッチ(第3段階)を子モデルで行うようにすると便利でしょう。 そうすれば、一方を内部使用のためだけのインスタンスとして使うことが出来、他方をエンド・ユーザの入力の受信に責任を持つモデルとしてコントローラと結びつける事が出来ます。 もう一つのやり方としては、value プロパティを構成して独自のロジックを実装することも可能です。

6.3.10 リレーショナル・データを扱う

個々のデータベース・テーブルを扱うだけでなく、アクティブ・レコードは関連したテーブルのデータも一緒に読み出して、 主たるデータを通して簡単にアクセス出来るようにすることが出来ます。 例えば、一人の顧客は一つまたは複数の注文を発することがあり得ますので、顧客のデータは注文のデータと関連を持っていることになります。 このリレーションが適切に宣言されていれば、$customer->orders という式を使って顧客の注文情報にアクセスすることが出来ます。 $customer->orders は、顧客の注文情報を Order アクティブ・レコード・インスタンスの配列として返してくれます。

リレーションを宣言する

アクティブ・レコードを使ってリレーショナル・データを扱うためには、最初に、アクティブ・レコード・クラスの中でリレーションを宣言する必要があります。 これは、以下のように、関心のあるそれぞれのリレーションについて リレーション・メソッド を宣言するだけの簡単な作業です。

```
class Customer extends ActiveRecord
{
    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    // ...

    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
    ;
    }
}
```

上記のコードでは、`Customer` クラスのために `orders` リレーションを宣言し、`Order` クラスのために `customer` リレーションを宣言しています。

各リレーション・メソッドは `getXyz` という名前にしなければなりません。ここで `xyz` (最初の文字は小文字です) が リレーション名 と呼ばれます。 リレーション名は 大文字と小文字を区別する ことに注意してください。

リレーションを宣言する際には、次の情報を指定しなければなりません。

- リレーションの多重性: `hasMany()` または `hasOne()` のどちらかを呼ぶことによって指定されます。 上記の例では、リレーションの宣言において、顧客は複数の注文を持ち得るが、一方、注文は一人の顧客しか持たない、ということが容易に読み取れます。
- 関連するアクティブ・レコード・クラスの名前: `hasMany()` または `hasOne()` の最初のパラメータとして指定されます。 クラス名を取得するのに `Xyz::className()` を呼ぶのが推奨されるプラクティスです。 そうすれば、IDE の自動補完のサポートを得ることことが出来るだけでなく、コンパイル段階でエラーを検出することが出来ます。
- 二つの型のデータ間のリンク: 二つの型のデータの関連付けに用いられるカラムを指定します。 配列の値は主たるデータ (リレーショ

ンを宣言しているアクティブ・レコード・クラスによって表されるデータ) のカラムであり、 配列のキーは関連するデータのカラムです。
これを記憶するための簡単な規則は、上の例で見るように、関連するアクティブ・レコードを書いた直後に、それに属するカラムを 続けて書く、ということです。ご覧のように、customer_id は Order のプロパティであり、 id はCustomer のプロパティです。

リレーショナル・データにアクセスする

リレーションを宣言した後は、リレーション名を通じてリレーショナル・データにアクセスすることが出来ます。 これは、リレーション・メソッドによって定義されるオブジェクト・プロパティ にアクセスするのと同様です。 このため、これを リレーション・プロパティ と呼びます。例えば、

```
// SELECT * FROM ‘customer‘ WHERE ‘id‘ = 123
$customer = Customer::findOne(123);

// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123
// $orders is an array of Order objects
$orders = $customer->orders;
```

> 情報: xyz という名前のリレーションを getter メソッド getXyz() によって宣言すると、xyz を オブジェクト・プロパティ のようにアクセスすることが出来るようになります。名前は大文字と小文字を区別することに注意してください。

リレーションが hasMany() によって宣言されている場合は、 このリレーション・プロパティにアクセスすると、関連付けられたアクティブ・レコード・インスタンスの配列が返されます。 リレーションが hasOne() によって宣言されている場合は、 このリレーション・プロパティにアクセスすると、関連付けられたアクティブ・レコード・インスタンスか、関連付けられたデータが見つからないときは null が返されます。

リレーション・プロパティに最初にアクセスしたときは、上記の例で示されているように、SQL 文が実行されます。 その同じプロパティに再びアクセスしたときは、SQL 文を再実行することなく、以前の結果が返されます。 SQL 文の再実行を強制するためには、まず、リレーション・プロパティの割り当てを解除 (unset) しなければなりません : `unset($customer->orders)`。

> 補足: リレーション・プロパティの概念は オブジェクト・プロパティ の機能と同一であるように見えますが、一つ、重要な相違点があります。 通常のオブジェクト・プロパティでは、プロパティの値はそれを定義する getter メソッドと同じ型を持ちます。 しかし、リレーション・プロパティにアクセ

スすると yii\db\ActiveRecord のインスタンスまたはその配列が返されるのに対して、 リレーション・メソッドは yii\db\ActiveQuery のインスタンスを返します。

```
$customer->orders; // ‘Order‘ オブジェクトの配列
$customer->getOrders(); // ActiveQuery のインスタンス
```

このことは、次のセクションで説明するように、カスタマイズしたクエリを作成するのに役に立ちます。

動的なリレーショナル・クエリ

リレーション・メソッドは yii\db\ActiveQuery のインスタンスを返すため、DB クエリを実行する前に、 クエリ構築メソッドを使ってこのクエリを更に修正することが出来ます。例えば、

```
$customer = Customer::findOne(123);

// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123 AND ‘subtotal‘ > 200
    ORDER BY ‘id‘
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

リレーション・プロパティにアクセスする場合と違って、リレーション・メソッドによって動的なリレーショナル・クエリを実行する場合は、 同じ動的なリレーショナル・クエリが以前に実行されたことがあっても、毎回、SQL 文が実行されます。

さらに進んで、もっと簡単に動的なリレーショナル・クエリを実行できるように、リレーションの宣言をパラメータ化したい場合もあるでしょう。 例えば、bigOrders リレーションを下記のように宣言することが出来ます。

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

これによって、次のようなリレーショナル・クエリを実行することが出来るようになります。

```
// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123 AND ‘subtotal‘ > 200
    ORDER BY ‘id‘
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123 AND ‘subtotal‘ > 100
    ORDER BY ‘id‘
```

```
$orders = $customer->bigOrders;
```

中間テーブルによるリレーション

データベースの設計において、二つの関連するテーブル間の多重性が多対多である場合は、通常、 中間テーブル[27] が導入されます。 例えば、order テーブルと item テーブルは、order_item と言う名前の中間テーブルによって関連付けることが出来ます。 このようにすれば、一つの注文を複数の商品に対応させ、また、一つの商品を複数の注文に対応させることが出来ます。

このようなリレーションを宣言するときは、via() または viaTable() のどちらかを呼んで中間テーブルを指定します。 via() と viaTable() の違いは、 前者が既存のリレーション名の形式で中間テーブルを指定するのに対して、後者は中間テーブルを直接に指定する、という点です。例えば、

```
class Order extends ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

あるいは、また、

```
class Order extends ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::className(), ['order_id' => 'id']);
    }

    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

中間テーブルを使って宣言されたリレーションの使い方は、通常のリレーションと同じです。例えば、

```
// SELECT * FROM `order` WHERE `id` = 100
$order = Order::findOne(100);

// SELECT * FROM `order_item` WHERE `order_id` = 100
// SELECT * FROM `item` WHERE `item_id` IN (...)
// 商品オブジェクトの配列を返す
$items = $order->items;
```

[27] https://en.wikipedia.org/wiki/Junction_table

複数のテーブルを経由するリレーション定義の連鎖

さらに、via() を使ってリレーション定義を連鎖させ、複数のテーブルを経由するリレーションを定義することも可能です。 上記の例で考えましょう。そこには Customer(顧客)、Order(注文) そして Item(品目) というクラスがあります。 Customer クラスに、発注された全ての注文によって購入された全ての品目を列挙するリレーションを追加して、 それに getPurchasedItems() という名前を付けることが出来ます。リレーション定義の連鎖が次のコード・サンプルで示されています。

```
class Customer extends ActiveRecord
{
    // ...

    public function getPurchasedItems()
    {
        // 顧客の購入品目、すなわち、‘Item‘ の ’id’ カラム
    が OrderItem の ’item_id’ に合致するもの
        return $this->hasMany(Item::className(), [’id’ => ’item_id’])
                    ->via(’orderItems’);
    }

    public function getOrderItems()
    {
        // 顧客の 、すなわち、OrderItems‘Order‘ の ‘id‘ カラム
    が ‘OrderItem‘ の ’order_id’ に合致するもの
        return $this->hasMany(OrderItem::className(), [’order_id’ => ’id’])
                    ->via(’orders’);
    }

    public function getOrders()
    {
        // 顧客の注文
        return $this->hasMany(Order::className(), [’customer_id’ => ’id’]);
    }
}
```

レイジー・ローディングとイーガー・ローディング

リレーショナル・データにアクセスする において、通常のオブジェクト・プロパティにアクセスするのと同じようにして、 アクティブ・レコード・インスタンスのリレーション・プロパティにアクセスすることが出来ることを説明しました。 SQL 文は、リレーション・プロパティに最初にアクセスするときにだけ実行されます。 このようなリレーショナル・データのアクセス方法を レイジー・ローディング と呼びます。例えば、

```
// SELECT * FROM ‘customer‘ WHERE ‘id‘ = 123
$customer = Customer::findOne(123);

// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123
```

```
$orders = $customer->orders;

// SQL は実行されない
$orders2 = $customer->orders;
```

レイジー・ローディングは非常に使い勝手が良いものです。 しかし、複数のアクティブ・レコード・インスタンスの同じリレーション・プロパティにアクセスする必要がある場合は、パフォーマンスの問題を生じ得ます。 次のコードサンプルを考えて見てください。実行される SQL 文の数はいくらになるでしょう?

```
// SELECT * FROM ‘customer‘ LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = ...
    $orders = $customer->orders;
}
```

上のコードのコメントから判るように、実行される SQL 文は 101 にもなります。 これは、for ループの中で、異なる `Customer` オブジェクトの `orders` リレーションにアクセスするたびに、 SQL 文が一つ実行されることになるからです。

このパフォーマンスの問題を解決するために、次に示すように、いわゆる イーガー・ローディング の手法を使うことが出来ます。

```
// SELECT * FROM ‘customer‘ LIMIT 100;
// SELECT * FROM ‘orders‘ WHERE ‘customer_id‘ IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
    // SQL は実行されない
    $orders = $customer->orders;
}
```

`yii\db\ActiveQuery::with()` を呼ぶことによって、最初の 100 人の顧客の注文をたった一つの SQL 文で返すように、アクティブ・レコードに指示をしています。 結果として、実行される SQL 文の数は 101 から 2 に減ります。

イーガー・ローディングは、一つだけでなく、複数のリレーションに対しても使うことが出来ます。さらには、ネストされたリレーション でさえ、イーガー・ロードすることが出来ます。 ネストされたリレーションというのは、関連するアクティブ・レコードの中で宣言されているリレーションです。 例えば、`Cutomer` が `orders` リレーションによって `Order` と関連しており、`Order` が `items` リレーションによって `Item` と関連している場合です。 `Customer` に対するクエリを実行するときに、ネストされたリレーションの記法である `orders.items` を使って、`items` をイーガー・ロードすることが出来ます。

次のコードは、with() のさまざまな使い方を示すものです。 ここでは、Customer クラスは orders と country という二つのリレーションを持っており、また、Order クラスは items という一つのリレーションを持っていると仮定しています。

```
// "orders" と "country" の両方をイーガー・ロードする
$customers = Customer::find()->with('orders', 'country')->all();
// これは下の配列記法と等価
$customers = Customer::find()->with(['orders', 'country'])->all();
// SQL は実行されない
$orders= $customers[0]->orders;
// SQL は実行されない
$country = $customers[0]->country;

// "orders" リレーションと、ネストされた "orders.items" をイーガー・ロード
$customers = Customer::find()->with('orders.items')->all();
// 最初の顧客の、最初の注文の品目にアクセスする
// SQL は実行されない
$items = $customers[0]->orders[0]->items;
```

深くネストされたリレーション、たとえば a.b.c.c をイーガー・ロードすることも出来ます。 このとき、すべての親リレーションもイーガー・ロードされます。 つまり、a.b.c.d を使って with() を呼ぶと、a、a.b、a.b.c そして a.b.c.d をイーガー・ロードすることになります。

> 情報: 一般化して言うと、N 個のリレーションのうち M 個のリレーションが 中間テーブル によって定義されている場合、 この N 個のリレーションをイーガー・ロードしようとすると、合計で 1+M+N 個の SQL クエリが実行されます。 ネストされたリレーション a.b.c.d は 4 個のリレーションとして数えられることに注意してください。

リレーションをイーガー・ロードするときに、対応するリレーショナル・クエリを無名関数を使ってカスタマイズすることが出来ます。 例えば、

```
// 顧客を検索し、その国とアクティブな注文を同時に返す
// SELECT * FROM `customer`
// SELECT * FROM `country` WHERE `id` IN (...)
// SELECT * FROM `order` WHERE `customer_id` IN (...) AND `status` = 1
$customers = Customer::find()->with([
    'country',
    'orders' => function ($query) {
        $query->andWhere(['status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

リレーションのためのリレーショナル・クエリをカスタマイズするときは、リレーション名を配列のキーとし、対応する値に無名関数を使わなければなりません。 無名関数が受け取る $query パラメータは、リレーションのためのリレーショナル・クエリを実行するのに使用される yii

\db\ActiveQuery オブジェクトを表します。 上のコード例では、注文の状態に関する条件を追加して、リレーショナル・クエリを修正しています。

> 補足: リレーションをイーガー・ロードするときに select() を呼ぶ場合は、 リレーションの宣言で参照されているカラムが選択されるように注意しなければなりません。 そうしないと、リレーションのモデルが正しくロードされないことがあります。例えば、

```
$orders = Order::find()->select(['id', 'amount'])->with('
    customer')->all();
// この場合、$orders[0]->customer は常に `null` になります。問題を
    修正するためには、次のようにしなければなりません。
$orders = Order::find()->select(['id', 'amount', 'customer_id'])
    ->with('customer')->all();
```

リレーションを使ってテーブルを結合する

> 補足: この項で説明されていることは、MySQL、PostgreSQL など、 リレーショナル・データベースに対してのみ適用されます。

ここまで説明してきたリレーショナル・クエリは、主たるデータを検索する際に主テーブルのカラムだけを参照するものでした。 現実には、関連するテーブルのカラムを参照しなければならない場合がよくあります。 例えば、少なくとも一つのアクティブな注文を持つ顧客を取得したい、というような場合です。 この問題を解決するためには、以下のようにして、テーブルを結合するクエリを構築することが出来ます。

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id`
// WHERE `order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->select('customer.*')
    ->leftJoin('order', '`order`.`customer_id` = `customer`.`id`')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->with('orders')
    ->all();
```

> 補足: JOIN SQL 文を含むリレーショナル・クエリを構築する場合は、カラム名の曖昧さを解消することが重要です。 カラム名に対応するテーブル名をプレフィクスするのが慣例です。

しかしながら、もっと良いのは、yii\db\ActiveQuery::joinWith() を呼んで、既にあるリレーションの宣言を利用するという手法です。

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

どちらの方法でも、実行される SQL 文のセットは同じです。けれども、後者の方がはるかに明快で簡潔です。

デフォルトでは、joinWith() は LEFT JOIN を使って、関連するテーブルを主テーブルに結合します。 第三のパラメータ $joinType によって異なる結合タイプ (例えば RIGHT JOIN) を指定することが出来ます。 指定したい結合タイプが INNER JOIN である場合は、代りに、innerJoinWith() を呼ぶだけで済ませることが出来ます。

デフォルトでは、joinWith() を呼ぶと、リレーションのデータがイーガー・ロード されます。 リレーションのデータを読み取りたくない場合は、第二のパラメータ $eagerLoading を false に指定することが出来ます。

> 補足: たとえイーガー・ローディングを有効にして joinWith() や innerJoinWith() を使う場合でも、 リレーションのデータを取得するのには JOIN クエリの結果は使われません。 その場合でも、やはり、イーガー・ローディング のセクションで説明したように、結合されたリレーションごとに追加のクエリが実行されます。

with() と同じように、一つまたは複数のリレーションを結合したり、 リレーションクエリをその場でカスタマイズしたり、ネストされたリレーションを結合したりすることが出来ます。 また、with() と joinWith() を混ぜて使用することも出来ます。例えば、

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

二つのテーブルを結合するときに、結合クエリの ON の部分に追加の条件を指定する必要がある場合があるでしょう。 これは、次のように、yii\db\ActiveQuery::onCondition() メソッドを呼ぶことによって実現できます。

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id` AND `order
    `.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->onCondition(['order.status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

上記のクエリは 全ての 顧客を返し、各顧客について全てのアクティブな注文を返します。 これは、少なくとも一つのアクティブな注文を持つ顧客を全て返す、という以前の例とは異なっていることに注意してください。

> 情報: yii\db\ActiveQuery が onCondition() によって条件を指定された場合、 クエリが JOIN 句を含む場合は、条件は ON の部分に置かれます。 クエリが JOIN 句を含まない場合は、条件は自動的に WHERE の部分に追加されます。 このようにして、リレーションのテーブルのカラムを含む条件だけが ON の部分に置かれます。

リレーションのテーブルのエイリアス　前に注意したように、クエリに JOIN を使うときは、カラム名の曖昧さを解消する必要があります。そのために、テーブルにエイリアスを定義することがよくあります。 リレーションのテーブルのためにエイリアスを設定することは、リレーショナル・クエリを次のようにカスタマイズすることによっても可能です。

```
$query->joinWith([
    'orders' => function ($q) {
        $q->from(['o' => Order::tableName()]);
    },
])
```

しかし、これでは非常に複雑ですし、リレーションオブジェクトのテーブル名をハードコーディングしたり、Order::tableName() を呼んだりしなければなりません。 バージョン 2.0.7 以降、Yii はこれに対するショートカットを提供しています。今では、次のようにしてリレーションのテーブルのエイリアスを定義して使うことが出来ます。

```
// orders リレーションを JOIN し、結果を orders.id でソートする
$query->joinWith(['orders o'])->orderBy('o.id');
```

上記の文法が動作するのは単純なリレーションの場合です。ネストされたリレーションを結合する (例えば、$query->joinWith(['orders.product'])) ときに、中間テーブルのエイリアスが必要になった場合は、 次の例のように、joinWith の呼び出しをネストさせる必要があります。

```
$query->joinWith(['orders o' => function($q) {
        $q->joinWith('product p');
    }])
    ->where('o.amount > 100');
```

逆リレーション

リレーションの宣言は、たいていの場合、二つのアクティブ・レコード・クラスの間で相互的なものになります。 例えば、Customer は orders リレーションによって Order に関連付けられ、逆に、Order はcustomer リレーションによって Customer に関連付けられる、という具合です。

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
    ;
    }
}
```

ここで、次のコード断片について考えて見てください。

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// SELECT * FROM `customer` WHERE `id` = 123
$customer2 = $order->customer;

// 異なる"" が表示される
echo $customer2 === $customer ? '同じ' : '異なる';
```

私たちは `$customer` と `$customer2` が同じであると期待しますが、そうではありません。 実際、二つは同じ顧客データを含んでいますが、オブジェクトとしては異なります。 `$order->customer` にアクセスするときに追加の SQL 文が実行されて、新しいオブジェクトである `$customer2` にデータが投入されます。

上記の例において、冗長な最後の SQL 文の実行を避けるためには、下に示すように、 `inverseOf()`メソッドを呼ぶことによって、`customer` が `orders` の 逆リレーション であることを Yii に教えておかなければなりません。

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])->
    inverseOf('customer');
    }
}
```

このようにリレーションの宣言を修正すると、次の結果を得ることが出来ます。

```
// SELECT * FROM `customer` WHERE `id` = 123
```

```
$customer = Customer::findOne(123);

// SELECT * FROM ‘order‘ WHERE ‘customer_id‘ = 123
$order = $customer->orders[0];

// No SQL will be executed
$customer2 = $order->customer;

// 同じ"" が表示される
echo $customer2 === $customer ? ’同じ’ : ’異なる’;
```

> 補足: 逆リレーションは 中間テーブル を含むリレーションについては宣言することが出来ません。 つまり、リレーションが via() または viaTable() によって定義されている場合は、inverseOf() を追加で呼んではいけません。

6.3.11 リレーションを保存する

リレーショナル・データを扱う時には、たいてい、さまざまなデータ間にリレーションを確立したり、既存のリレーションを破棄したりする必要があります。 そのためには、リレーションを定義するカラムの値を適切に設定することが必要です。 アクティブ・レコードを使う場合は、結局の所、次のようなコードを書くことになるでしょう。

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// Order において "customer" リレーションを定義する属性の値を設定する
$order->customer_id = $customer->id;
$order->save();
```

アクティブ・レコードは、この仕事をもっと楽に達成することが出来るように、link() メソッドを提供しています。

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link(’customer’, $customer);
```

link() メソッドは、リレーション名と、リレーションを確立する対象のアクティブ・レコード・インスタンスを指定することを要求します。 このメソッドは、二つのアクティブ・レコード・インスタンスをリンクする属性の値を修正して、それをデータベースに書き込みます。 上記の例では、Order インスタンスの customer_id 属性を Customer インスタンスの id 属性の値になるようにセットして、それをデータベースに保存します。

> 補足: 二つの新規作成されたアクティブ・レコード・インスタンスをリンクすることは出来ません。

link() を使用することの利点は、リレーションが 中間テーブル によって定義されている場合に、さらに明白になります。 例えば、一つの Order インスタンスと一つのItem インスタンスをリンクするのに、次のコードを使うことが出来ます。

```
$order->link('items', $item);
```

上記のコードによって、order_item 中間テーブルに、注文と商品を関連付けるための行が自動的に挿入されます。

> 情報: link() メソッドは、 影響を受けるアクティブ・レコード・インスタンスを保存する際に、データ検証を実行しません。 このメソッドを呼ぶ前にすべての入力値を検証することはあなたの責任です。

link() の逆の操作が unlink() です。 これは、既存の二つのアクティブ・レコード・インスタンスのリレーションを破棄します。例えば、

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

デフォルトでは、unlink() メソッドは、 既存のリレーションを指定している外部キーの値を null に設定します。 ただし、$delete パラメータを true にしてメソッドに渡して、その外部キーを含むテーブル行を削除するという方法を選ぶことも出来ます。

リレーションに中間テーブルが含まれている場合は、unlink() を呼ぶと、 中間テーブルにある外部キーがクリアされるか、または、$delete が true であるときは、 中間テーブルにある対応する行が削除されるかします。

6.3.12 DBMS 間のリレーション

アクティブ・レコードは、異なるデータベースをバックエンドに持つアクティブ・レコードの間でリレーションを宣言することを可能にしています。 データベースは異なるタイプ (例えば、MySQL と PostgreSQL、または、MS SQL と MongoDB) であってもよく、別のサーバで動作していても構いません。 同じ構文を使ってリレーショナル・クエリを実行することが出来ます。例えば、

```
// Customer はリレーショナル・データベース 例えば( MySQL) の "customer" テーブルと関連付けられている
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
```

```
    }

    public function getComments()
    {
        // Customer は多くの Comment を持つ
        return $this->hasMany(Comment::className(), ['customer_id' => 'id'])
    ;
    }
}

// Comment は MongoDb データベースの "comment" コレクションと関連付けられてい
    る
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // Comment は 一つの Customer を持つ
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
    ;
    }
}

$customers = Customer::find()->with('comments')->all();
```

このセクションで説明されたリレーショナル・クエリ機能のほとんどを使用することが出来ます。

> 補足: `yii\db\ActiveQuery::joinWith()` の使用は、データベース間の JOIN クエリをサポートしているデータベースに限定されます。 この理由により、上記の例では `joinWith` メソッドは使用することが出来ません。MongoDB は JOIN をサポートしていないからです。

6.3.13 クエリ・クラスをカスタマイズする

デフォルトでは、全てのアクティブ・レコードのクエリは `yii\db\ActiveQuery` によってサポートされます。 カスタマイズされたクエリ・クラスをアクティブ・レコードで使用するためには、`yii\db\ActiveRecord::find()` メソッドをオーバーライドして、カスタマイズされたクエリ・クラスのインスタンスを返すようにしなければなりません。例えば、

```
// file Comment.php
namespace app\models;

use yii\db\ActiveRecord;
```

```
class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

このようにすると、Comment のクエリを実行したり (例えば find() や findOne() を呼んだり)、Comment とのリレーションを定義したり (例えば hasOne() を定義したり) する際には、いつでも、AcctiveQuery の代りに CommentQuery のインスタンスを使用することになります。

さて、CommentQuery クラスを定義しなければならない訳ですが、このクラスをさまざまな創造的方法でカスタマイズして、あなたのクエリ構築作業を楽しいものにすることが出来ます。例えば、

```
// file CommentQuery.php
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    // デフォルトで条件を追加 省略可()
    public function init()
    {
        $this->andOnCondition(['deleted' => false]);
        parent::init();
    }

    // ... ここにカスタマイズしたクエリ・メソッドを追加 ...

    public function active($state = true)
    {
        return $this->andOnCondition(['active' => $state]);
    }
}
```

> 補足: 新しいクエリ構築メソッドを定義するときには、通常は、既存のどの条件も上書きしないように、 onCondition() ではなく、andOnCondition() または orOnCondition() を呼んで条件を追加しなければなりません。

このようにすると、次のようなクエリ構築のコードを書くことが出来るようになります。

```
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

> ヒント: 大きなプロジェクトでは、アクティブ・レコード・クラスをクリーンに保つことが出来るように、 クエリ関連の

> コードのほとんどをカスタマイズされたクエリ・クラスに保持することが推奨されます。

この新しいクエリ構築メソッドは、Comment に関するリレーションを定義するときや、リレーショナル・クエリを実行するときにも使用することが出来ます。

```
class Customer extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::className(), ['customer_id' => 'id'])
    ->active();
    }
}

$customers = Customer::find()->with('activeComments')->all();
```

```
// あるいは、また
class Customer extends \yii\db\ActiveRecord
{
    public function getComments()
    {
        return $this->hasMany(Comment::className(), ['customer_id' => 'id'])
    ;
    }
}

$customers = Customer::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

> 情報: Yii 1.1 には、スコープ と呼ばれる概念がありました。 Yii 2.0 では、スコープはもはや直接にはサポートされません。 同じ目的を達するためには、カスタマイズされたクエリ・クラスとクエリ・メソッドを使わなければなりません。

6.3.14 追加のフィールドを選択する

アクティブ・レコードのインスタンスにクエリ結果からデータが投入されるときは、 受け取ったデータセットのカラムの値が対応する属性に入れられます。

クエリ結果から追加のカラムや値を取得して、アクティブ・レコードの内部に格納することが出来ます。 例えば、ホテルの客室の情報を含む room という名前のテーブルがあるとしましょう。 そして、全ての客室のデータは length (長さ)、width (幅)、height (高さ) というフィールドを使って、部屋の幾何学的なサイズに関する情報を格納しているとします。 空いている全ての部屋の一覧を容積の降順で取得する必要がある

場合を考えて見てください。 レコードをその値で並べ替える必要があるので、PHP を使って容積を計算することは出来ません。 しかし、同時に、一覧には volume (容積) も表示したいでしょう。 目的を達するためには、Room アクティブ・レコード・クラスにおいて追加のフィールドを宣言し、volume の値を格納する必要があります。

```
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

そして、部屋の容積を計算して並べ替えを実行するクエリを構築しなければなりません。

```
$rooms = Room::find()
    ->select([
        '{{room}}.*', // 全てのカラムを選択
        '([[length]] * [[width]] * [[height]]) AS volume', // 容積を計算
    ])
    ->orderBy('volume DESC') // 並べ替えを適用
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // SQL によって計算された値を含んでいる
}
```

追加のフィールドが選択できることは、集計クエリに対して特に有効に機能します。 注文の数とともに顧客の一覧を表示する必要がある場合を想定してください。 まず初めに、Customer クラスの中で、orders リレーションと、注文数を格納するための追加のフィールドを宣言しなければなりません。

```
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

そして、order を結合して注文数を計算するクエリを構築することが出来ます。

```
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // 顧客の全てのフィールドを選択
        'COUNT({{order}}.id) AS ordersCount' // 注文数を計算
    ])
```

```
    ->joinWith('orders') // テーブルの結合を保証する
    ->groupBy('{{customer}}.id') // 結果をグループ化して、集計関数の動作を保証
    する
    ->all();
```

この方法を使うことの短所の一つは、情報が SQL クエリでロードされていない場合には、それを別途計算しなければならない、ということです。 従って、追加のセレクト文を持たない通常のクエリによって特定のレコードを読み出した場合には、追加のフィールドの実際の値を返すことは不可能になります。 同じことが新しく保存されたレコードでも起こります。

```
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // まだ指定されていないため、この値は `null` になります。
```

__get() と __set() のマジック・メソッドを使用すれば、 プロパティの動作をエミュレートすることが出来ます。

```
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }

    public function getVolume()
    {
        if (empty($this->length) || empty($this->width) || empty($this->
    height)) {
            return null;
        }

        if ($this->_volume === null) {
            $this->setVolume(
                $this->length * $this->width * $this->height
            );
        }

        return $this->_volume;
    }

    // ...
}
```

このようにすると、SELECT クエリによって容積が提供されていない場合に、 モデルの他の属性を使って容積を自動的に計算することが出来ます。

集約フィールドも、同じように、定義されたリレーションを使って計算することが出来ます。

```
class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }

    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // プライマリ・キーが null の場合のリレーショナル・クエリを防止
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount(count($this->orders)); // 要求に応じてリレーションから集約を計算
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

このコードでは、'ordersCount' が 'select' 文に存在する場合は、`Customer::ordersCount` はクエリの結果によって投入されます。 そうでない場合は、要求に応じて、`Customer::orders` リレーションを使って計算されます。

マジック・メソッドによってプロパティをエミュレートする手法は、ある種のリレーショナル・データ、特に集約のショートカットを作成するためにも使用することが出来ます。 例えば、

```
class Customer extends \yii\db\ActiveRecord
{
    /**
     * 読み出し専用の集約データの仮想プロパティを定義
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // プライマリ・キーが null の場合のリレーショナル・クエリを防止
        }
```

```
        return empty($this->ordersAggregation) ? 0 : $this->
    ordersAggregation[0]['counted'];
    }

    /**
     * 通常の 'orders' リレーションを宣言
     */
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }

    /**
     * 集約データを提供する新しいリレーションを 'orders' を元にして宣言
     */
    public function getOrdersAggregation()
    {
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // イーガー・ローディングにより、追加のクエリ
    なしに、リレーションから集約データを出力
}

$customer = Customer::findOne($pk);
$customer->ordersCount; // レイジーロードされたリレーションから集約データを出
    力
```

6.4 データベース・マイグレーション

データベース駆動型のアプリケーションを開発し保守する途上で、ソース・コードが進化するのと同じように、 使用されるデータベースの構造も進化していきます。 例えば、アプリケーションの開発中に、新しいテーブルが必要であることが分ったり、アプリケーションを配備した後に、 クエリのパフォーマンスを向上させるためにインデックスを作成すべきことが発見されたりします。 データベースの構造の変更が何らかのソース・コードの変更を要求する場合はよくありますから、 Yii はいわゆる データベース・マイグレーション 機能を提供して、ソース・コードとともにバージョン管理される データベース・マイグレーション の形式でデータベースの変更を追跡できるようにしています。

下記の一連のステップは、開発中にチームによってデータベース・マ

イグレーションがどのように使用されるかを示す例です。

1. Tim が新しいマイグレーション (例えば、新しいテーブルを作成したり、カラムの定義を変更したりなど) を作る。

2. Tim が新しいマイグレーションをソース・コントロール・システム (例えば Git や Mercurial) にコミットする。

3. Doug がソース・コントロール・システムから自分のレポジトリを更新して新しいマイグレーションを受け取る。

4. Doug がマイグレーションを彼のローカルの開発用データベースに適用して、自分のデータベースの同期を取り、 Tim が行った変更を反映する。

そして、次の一連のステップは、本番環境でデータベース・マイグレーションとともに新しいリリースを配備する方法を示すものです。

1. Scott は新しいデータベース・マイグレーションをいくつか含むプロジェクトのレポジトリにリリース・タグを作成する。

2. Scott は本番サーバでソース・コードをリリース・タグまで更新する。

3. Scott は本番のデータベースに対して累積したデータベース・マイグレーションを全て適用する。

Yii は一連のマイグレーション・コマンドライン・ツールを提供して、以下の機能をサポートします。

- 新しいマイグレーションの作成
- マイグレーションの適用
- マイグレーションの取消
- マイグレーションの再適用
- マイグレーションの履歴と状態の表示

これらのツールは、全て、`yii migrate` コマンドからアクセスすることが出来ます。 このセクションでは、これらのツールを使用して、さまざまなタスクをどうやって達成するかを詳細に説明します。 各ツールの使用方法は、ヘルプコマンド `yii help migrate` によっても知ることが出来ます。

> ヒント: マイグレーションはデータベース・スキーマに影響を及ぼすだけでなく、既存のデータを新しいスキーマに合うように修正したり、RBAC 階層を作成したり、 キャッシュをクリーンアップしたりするために使うことも出来ます。

> 補足: マイグレーションを使ってデータを操作する際に、作成済みのアクティブ・レコード・クラスを使えば 便利かも知れないと気が付くことがあるでしょう。なぜなら、ロジックのいくつかは既にアクティブ・レコードで実装済みなのですから。 しかしながら、永久に不変であり続けることを本質とするマイグレーションにおいて書かれるコードとは対照的に、アプリケーションの ロジックは変化にさらされるものであることに留意しなければなりません。つまり、マイグレーション・コードにアクティブ・レコードを使った場合、 アクティブ・レコードのレイヤにおけるロジックの変更が既存のマイグレーションを偶発的に破壊する可能性があります。この理由により、 マイグレーション・コードは、アクティブ・レコード・クラスなど、他のアプリケーション・ロジックから独立を保つべきです。

6.4.1 マイグレーションを作成する

新しいマイグレーションを作成するためには、次のコマンドを実行します。

```
yii migrate/create <name>
```

要求される `name` パラメータには、マイグレーションの非常に短い説明を指定します。 例えば、マイグレーションが *news* という名前のテーブルを作成するものである場合は、 `create_news_table` という名前を使って、次のようにコマンドを実行すれば良いでしょう。

```
yii migrate/create create_news_table
```

> 補足: この `name` 引数は、生成されるマイグレーション・クラス名の一部として使用されますので、 アルファベット、数字、および/または、アンダースコアだけを含むものでなければなりません。

上記のコマンドは、`m150101_185401_create_news_table.php` という名前の新しい PHP クラス・ファイルを `@app/migrations` ディレクトリに作成します。このファイルは次のようなコードを含み、主として、 スケルトン・コードを持った `m150101_185401_create_news_table` というマイグレーション・クラスを宣言するためのものです。

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
```

```
    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

各データベース・マイグレーションは yii\db\Migration から拡張した PHP クラスとして定義されます。 マイグレーション・クラスの名前は、m<YYMMDD_HHMMSS>_<Name> という形式で自動的に生成されます。ここで、

- <YYMMDD_HHMMSS> は、マイグレーション作成コマンドが実行された UTC 日時を表し、
- <Name> は、あなたがコマンドに与えた name 引数と同じ値になります。

マイグレーション・クラスにおいて、あなたがなすべき事は、データベースの構造に変更を加える up() メソッドにコードを書くことです。また、up() によって加えられた変更を取り消すための down() メソッドにも、コードを書きたいと思うかもしれません。 up() メソッドは、このマイグレーションによってデータベースをアップグレードする際に呼び出され、down() メソッドはデータベースをダウングレードする際に呼び出されます。 下記のコードは、新しい news テーブルを作成するマイグレーション・クラスをどのようにして実装するかを示すものです。

```
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
```

```
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

> 情報: 全てのマイグレーションが取り消し可能な訳ではありません。 例えば、up() メソッドがテーブルからある行を削除するものである場合、down() メソッドでその行を回復することは出来ません。 また、データベース・マイグレーションを取り消すことはあまり一般的ではありませんので、場合によっては、面倒くさいというだけの理由で down() を実装しないこともあるでしょう。 そういう場合は、マイグレーションが取り消し不可能であることを示すために、down() メソッドで false を返さなければなりません。

基底のマイグレーション・クラス yii\db\Migration は、db プロパティによって、 データベース接続にアクセスすることを可能にしています。このデータベース接続によって、データベース・スキーマを扱う で説明されているメソッドを使い、データベース・スキーマを操作することが出来ます。

テーブルやカラムを作成するときは、物理的な型を使うのでなく、抽象型 を使って、 あなたのマイグレーションが特定の DBMS に依存しないようにします。 yii\db\Schema クラスが、サポートされている抽象型を表す一連の定数を定義しています。 これらの定数は TYPE_<Name> という形式の名前を持っています。 例えば、TYPE_PK は、オート・インクリメントのプライマリ・キー型であり、TYPE_STRING は文字列型です。これらの抽象型は、マイグレーションが特定のデータベースに適用されるときに、対応する物理型に翻訳されます。 MySQL の場合は、TYPE_PK は int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY に変換され、TYPE_STRING は varchar(255) となります。

抽象型を使用するときに、付随的な制約を追加することが出来ます。上記の例では、Schema::TYPE_STRING に NOT NULL を追加して、このカラムが null を許容しないことを指定しています。

> 情報: 抽象型と物理型の対応関係は、それぞれの QueryBuilder の具象クラスの $typeMap プロパティによって定義されています。

バージョン 2.0.6 以降は、カラムのスキーマを定義するための更に便利な方法を提供するスキーマビルダが新たに導入されています。 したがって、上記のマイグレーションは次のように書くことが出来ます。

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

カラムの型を定義するために利用できる全てのメソッドのリストは、yii\db\SchemaBuilderTrait の API ドキュメントで参照することが出来ます。

6.4.2 マイグレーションを生成する

バージョン 2.0.7 以降では、マイグレーション・コンソールがマイグレーションを生成する便利な方法を提供しています。

マイグレーションの名前が特別な形式である場合は、生成されるマイグレーション・ファイルに追加のコードが書き込まれます。 例えば、create_xxx_table や drop_xxx_table であれば、テーブルの作成や削除をするコードが追加されます。 以下で、この機能の全ての変種を説明します。

テーブルの作成

```
yii migrate/create create_post_table
```

上記のコマンドは、次のコードを生成します。

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
```

```
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

テーブルのフィールドも直接に生成したい場合は、`--fields` オプションでフィールドを指定します。

```
yii migrate/create create_post_table --fields="title:string,body:text"
```

これは、次のコードを生成します。

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

さらに多くのフィールド・パラメータを指定することも出来ます。

```
yii migrate/create create_post_table --fields="title:string(12):notNull:
    unique,body:text"
```

これは、次のコードを生成します。

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
```

```
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

> 補足: プライマリ・キーが自動的に追加されて、デフォルトでは `id` と名付けられます。 別の名前を使いたい場合は、`--fields="name:primaryKey"` のように、明示的に指定してください。

外部キー　バージョン 2.0.8 からは、`foreignKey` キーワードを使って外部キーを生成することができます。

```
yii migrate/create create_post_table --fields="author_id:integer:notNull:
    foreignKey(user),category_id:integer:defaultValue(1):foreignKey,title:
    string,body:text"
```

これは、次のコードを生成します。

```
/**
 * Handles the creation for table `post`.
 * Has foreign keys to the tables:
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
```

```
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // creates index for column `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // add foreign key for table `user`
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
            'id',
            'CASCADE'
        );

        // creates index for column `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );

        // add foreign key for table `category`
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `user`
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // drops index for column `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
```

```
        );

        // drops foreign key for table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // drops index for column `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
        );

        $this->dropTable('post');
    }
}
```

カラムの記述における `foreignKey` キーワードの位置によって、生成されるコードが変ることはありません。 つまり、

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

これらはすべて同じコードを生成します。

`foreignKey` キーワードは括弧の中にパラメータを取ることが出来て、これが生成される外部キーの関連テーブルの名前になります。 パラメータが渡されなかった場合は、テーブル名はカラム名から推測されます。

上記の例で `author_id:integer:notNull:foreignKey(user)` は、 `user` テーブルへの外部キーを持つ `author_id` という名前のカラムを生成します。 一方、`category_id:integer:defaultValue(1):foreignKey` は、 `category` テーブルへの外部キーを持つ `category_id` というカラムを生成します。

2.0.11 以降では、`foreignKey` キーワードは空白で区切られた第二のパラメータを取ることが出来ます。 これは、生成される外部キーに関連づけられるカラム名を表します。 第二のパラメータが渡されなかった場合は、カラム名はテーブル・スキーマから取得されます。 スキーマが存在しない場合や、プライマリ・キーが設定されていなかったり、複合キーであったりする場合は、デフォルト名として `id` が使用されます。

テーブルを削除する

```
yii migrate/create drop_post_table --fields="title:string(12):notNull:unique
    ,body:text"
```

これは、次のコードを生成します。

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
```

```
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

カラムを追加する

マイグレーションの名前が `add_xxx_column_to_yyy_table` の形式である場合、ファイルの内容は、 必要となる `addColumn` と `dropColumn` を含むことになります。

カラムを追加するためには、次のようにします。

```
yii migrate/create add_position_column_to_post_table --fields="position:
    integer"
```

これが次のコードを生成します。

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

次のようにして複数のカラムを指定することも出来ます。

```
yii migrate/create add_xxx_column_yyy_column_to_zzz_table --fields="xxx:
    integer,yyy:text"
```

カラムを削除する

マイグレーションの名前が `drop_xxx_column_from_yyy_table` の形式である場合、 ファイルの内容は、必要となる `addColumn` と `dropColumn` を含むことになります。

```
yii migrate/create drop_position_column_from_post_table --fields="position:
    integer"
```

これは、次のコードを生成します。

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
```

```
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

中間テーブルを追加する

マイグレーションの名前が `create_junction_table_for_xxx_and_yyy_tables` の形式である場合は、 中間テーブルを作成するのに必要となるコードが生成されます。

```
yii migrate/create create_junction_table_for_post_and_tag_tables --fields="
    created_at:dateTime"
```

これは、次のコードを生成します。

```
/**
 * Handles the creation for table `post_tag`.
 * Has foreign keys to the tables:
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_table_for_post_and_tag_tables extends
    Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'created_at' => $this->dateTime(),
            'PRIMARY KEY(post_id, tag_id)',
        ]);

        // creates index for column `post_id`
        $this->createIndex(
            'idx-post_tag-post_id',
            'post_tag',
            'post_id'
        );

        // add foreign key for table `post`
        $this->addForeignKey(
            'fk-post_tag-post_id',
```

```
            'post_tag',
            'post_id',
            'post',
            'id',
            'CASCADE'
        );

        // creates index for column `tag_id`
        $this->createIndex(
            'idx-post_tag-tag_id',
            'post_tag',
            'tag_id'
        );

        // add foreign key for table `tag`
        $this->addForeignKey(
            'fk-post_tag-tag_id',
            'post_tag',
            'tag_id',
            'tag',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `post`
        $this->dropForeignKey(
            'fk-post_tag-post_id',
            'post_tag'
        );

        // drops index for column `post_id`
        $this->dropIndex(
            'idx-post_tag-post_id',
            'post_tag'
        );

        // drops foreign key for table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // drops index for column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );
```

```
        $this->dropTable('post_tag');
    }
}
```

2.0.11 以降では、中間テーブルの外部キーのカラム名はテーブル・スキーマから取得されます。 スキーマでテーブルが定義されていない場合や、プライマリ・キーが設定されていなかったり複合キーであったりする場合は、デフォルト名 `id` が使われます。

トランザクションを使うマイグレーション

複雑な一連の DB マイグレーションを実行するときは、通常、データベースの一貫性と整合性を保つために、 各マイグレーションが全体として成功または失敗することを保証する必要があります。 この目的を達成するために、各マイグレーションの DB 操作を トランザクション で囲むことが推奨されます。

トランザクションを使うマイグレーションを実装するためのもっと簡単な方法は、マイグレーションのコードを `safeUp()` と `safeDown()` のメソッドに入れることです。 この二つのメソッドが `up()` および `down()` と違う点は、これらが暗黙のうちにトランザクションに囲まれていることです。結果として、これらのメソッドの中で何か操作が失敗した場合は、先行する全ての操作が自動的にロールバックされます。

次の例では、`news` テーブルを作成するだけでなく、このテーブルに初期値となる行を挿入しています。

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

通常、safeUp() で複数の DB 操作を実行する場合は、safeDown() では実行の順序を逆にしなければならないことに注意してください。 上記の例では、safeUp() では、最初にテーブルを作って、次に行を挿入し、safeDown() では、先に行を削除して、次にテーブルを削除しています。

> 補足: 全ての DBMS がトランザクションをサポートしている訳ではありません。また、トランザクションに入れることが出来ない DB クエリもあります。 いくつかの例を 暗黙のコミット[28] で見ることが出来ます。 その場合には、代りに、up() と down() を実装しなければなりません。

データベース・アクセス・メソッド

基底のマイグレーション・クラス yii\db\Migration は、データベースにアクセスして操作するための一連のメソッドを提供しています。 あなたは、これらのメソッドが、yii\db\Command クラスによって提供される DAO メソッド と同じような名前を付けられていることに気付くでしょう。 例えば、yii\db\Migration::createTable() メソッドは、yii\db\Command::createTable() と全く同じように、 新しいテーブルを作成します。

yii\db\Migration によって提供されているメソッドを使うことの利点は、yii\db\Command インスタンスを明示的に作成する必要がないこと、そして、各メソッドを実行すると、 どのようなデータベース操作がどれだけの時間をかけて実行されたかを教えてくれる有益なメッセージが自動的に表示されることです。

以下がそういうデータベース・アクセス・メソッドの一覧です。

- execute(): SQL 文を実行
- insert(): 一行を挿入
- batchInsert(): 複数行を挿入
- update(): 行を更新
- upsert(): 一行を挿入または既に存在していれば更新 (2.0.14 以降)
- delete(): 行を削除
- createTable(): テーブルを作成
- renameTable(): テーブルの名前を変更
- dropTable(): テーブルを削除
- truncateTable(): テーブル中の全ての行を削除
- addColumn(): カラムを追加
- renameColumn(): カラムの名前を変更
- dropColumn(): カラムを削除
- alterColumn(): カラムの定義を変更
- addPrimaryKey(): プライマリ・キーを追加
- dropPrimaryKey(): プライマリ・キーを削除
- addForeignKey(): 外部キーを追加

[28] http://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html

- dropForeignKey(): 外部キーを削除
- createIndex(): インデックスを作成
- dropIndex(): インデックスを削除
- addCommentOnColumn(): カラムにコメントを追加
- dropCommentFromColumn(): カラムからコメントを削除
- addCommentOnTable(): テーブルにコメントを追加
- dropCommentFromTable(): テーブルからコメントを削除

> 情報: yii\db\Migration は、データベース・クエリ・メソッドを提供しません。 これは、通常、データベースからのデータ取得については、メッセージを追加して表示する必要がないからです。 更にまた、複雑なクエリを構築して実行するためには、強力な クエリ・ビルダ を使うことが出来るからです。 マイグレーションの中でクエリ・ビルダを使うと、次のようなコードになります。

```
// 全ユーザについて、status フィールドを更新する
foreach((new Query)->from('users')->each() as $user) {
    $this->update('users', ['status' => 1], ['id' => $user['id'
    ]]);
}
```

6.4.3 マイグレーションを適用する

データベースを最新の構造にアップグレードするためには、利用できる全ての新しいマイグレーションを適用するために、次のコマンドを使わなければなりません。

```
yii migrate
```

コマンドを実行すると、まだ適用されていない全てのマイグレーションが一覧表示されます。 リストされたマイグレーションを適用することをあなたが確認すると、タイムスタンプの値の順に、一つずつ、 すべての新しいマイグレーション・クラスの up() または safeUp() メソッドが実行されます。 マイグレーションのどれかが失敗した場合は、コマンドは残りのマイグレーションを適用せずに終了します。

> ヒント: あなたのサーバでコマンドラインを使用できない場合は web shell[29] エクステンションを使ってみてください。

適用が成功したマイグレーションの一つ一つについて、migration という名前のデータベース・テーブルに行が挿入されて、 マイグレーションの成功が記録されます。この記録によって、マイグレーション・ツールは、どのマイグレーションが適用され、 どのマイグレーションが適用されていないかを特定することが出来ます。

[29] https://github.com/samdark/yii2-webshell

> 情報: マイグレーション・ツールは、コマンドの db オプションで指定されたデータベースに migration テーブルを自動的に作成します。デフォルトでは、このデータベースは db アプリケーション・コンポーネント によって指定されます。

時として、利用できる全てのマイグレーションではなく、一つまたは数個の新しいマイグレーションだけを適用したい場合があります。 コマンドを実行するときに、適用したいマイグレーションの数を指定することによって、そうすることが出来ます。 例えば、次のコマンドは、次の三個の利用できるマイグレーションを適用しようとするものです。

```
yii migrate 3
```

また、そのマイグレーションまでをデータベースに適用するという、特定のマイグレーションを明示的に指定することも出来ます。 そのためには、migrate/to コマンドを、次のどれかの形式で使います。

```
yii migrate/to 150101_185401                      # タイムスタンプを使ってマイグレーションを指定
yii migrate/to "2015-01-01 18:54:01"              # strtotime() によって解釈できる文字列を使用
yii migrate/to m150101_185401_create_news_table   # フルネームを使用
yii migrate/to 1392853618                         # UNIX タイムスタンプを使用
```

指定されたマイグレーションよりも古いものが適用されずに残っている場合は、指定されたものが適用される前に、 すべて適用されます。

指定されたマイグレーションが既に適用済みである場合、それより新しいものが適用されていれば、すべて取り消されます。

6.4.4 マイグレーションを取り消す

適用済みのマイグレーションを一個または複数個取り消したい場合は、下記のコマンドを使うことが出来ます。

```
yii migrate/down     # 最近に適用されたマイグレーション一個を取り消す
yii migrate/down 3   # 最近に適用されたマイグレーション三個を取り消す
```

> 補足: 全てのマイグレーションが取り消せるとは限りません。そのようなマイグレーションを取り消そうとするとエラーとなり、取り消しのプロセス全体が終了させられます。

6.4.5 マイグレーションを再適用する

マイグレーションの再適用とは、指定されたマイグレーションを最初に取り消してから、再度適用することを意味します。 これは次のコマンドによって実行することが出来ます。

```
yii migrate/redo       # 最後に適用された一個のマイグレーションを再適用する
yii migrate/redo 3     # 最後に適用された三個のマイグレーションを再適用する
```

補足: マイグレーションが取り消し不可能な場合は、それを再適用することは出来ません。

6.4.6 マイグレーションをリフレッシュする

Yii 2.0.13 以降、データベースから全てのテーブルと外部キーを削除して、全てのマイグレーションを最初から再適用することが出来ます。

```
yii migrate/fresh       # データベースを削除し、全てのマイグレーションを最初から適用する
```

6.4.7 マイグレーションをリスト表示する

どのマイグレーションが適用済みであり、どのマイグレーションが未適用であるかをリスト表示するために、次のコマンドを使うことが出来ます。

```
yii migrate/history     # 最後に適用された 10 個のマイグレーションを表示
yii migrate/history 5   # 最後に適用された 5 個のマイグレーションを表示
yii migrate/history all # 適用された全てのマイグレーションを表示

yii migrate/new         # 適用可能な最初の 10 個のマイグレーションを表示
yii migrate/new 5       # 適用可能な最初の 5 個のマイグレーションを表示
yii migrate/new all     # 適用可能な全てのマイグレーションを表示
```

6.4.8 マイグレーション履歴を修正する

時として、実際にマイグレーションを適用したり取り消したりするのではなく、 データベースが特定のマイグレーションまでアップグレードされたとマークしたいだけ、という場合があります。 このようなことがよく起るのは、データベースを手作業で特定の状態に変更した後に、その変更のための一つまたは複数のマイグレーションを記録はするが再度適用はしたくない、という場合です。 次のコマンドでこの目的を達することが出来ます。

```
yii migrate/mark 150101_185401                          # タイムスタンプを使ってマイグレーションを指定
yii migrate/mark "2015-01-01 18:54:01"                  # strtotime() によって解釈できる文字列を使用
yii migrate/mark m150101_185401_create_news_table       # フルネームを使用
yii migrate/mark 1392853618                             # UNIX タイムスタンプを使用
```

このコマンドは、一定の行を追加または削除して、`migration` テーブルを修正し、データベースが指定されたものまでマイグレーションが適用済みであることを示します。 このコマンドによってマイグレーションが適用されたり取り消されたりはしません。

6.4.9 マイグレーションをカスタマイズする

マイグレーションコマンドをカスタマイズする方法がいくつかあります。

コマンドライン・オプションを使う

マイグレーション・コマンドには、その動作をカスタマイズするために使うことが出来るコマンドライン・オプションがいくつかあります。

- `interactive`: 真偽値 (デフォルト値は true)。マイグレーションを対話モードで実行するかどうかを指定します。 true である場合は、コマンドが何らかの操作を実行する前に、ユーザは確認を求められます。 コマンドがバックグラウンドのプロセスで使用される場合は、このオプションを false にセットします。
- `migrationPath`: 文字列 (デフォルト値は `@app/migrations`)。 全てのマイグレーション・クラス・ファイルを保存しているディレクトリを指定します。 この値は、ディレクトリ・パスか、パス・エイリアスとして指定することが出来ます。 ディレクトリが存在する必要があり、そうでなければコマンドがエラーを発生させることに注意してください。
- `migrationTable`: 文字列 (デフォルト値は `migration`)。マイグレーション履歴の情報を保存するためのデータベース・テーブル名を指定します。 テーブルが存在しない場合は、コマンドによって自動的に作成されます。 `version varchar(255) primary key, apply_time integer` という構造のテーブルを手作業で作成しても構いません。
- `db`: 文字列 (デフォルト値は `db`)。データベース アプリケーション・コンポーネント の ID を指定します。 このコマンドによってマイグレーションを適用されるデータベースを表します。
- `templateFile`: 文字列 (デフォルト値は `@yii/views/migration.php`)。 スケルトンのマイグレーション・クラス・ファイルを生成するために使用されるテンプレート・ファイルのパスを指定します。 この値は、ファイル・パスか、パス エイリアス として指定することが出来ます。 テンプレート・ファイルは PHP スクリプトであり、その中で、マイグレーション・クラスの名前を取得するための `$className` という事前定義された変数を使うことが出来ます。
- `generatorTemplateFiles`: 配列 (デフォルト値は ‘[

```
'create_table' => '@yii/views/createTableMigration.php',
'drop_table' => '@yii/views/dropTableMigration.php',
'add_column' => '@yii/views/addColumnMigration.php',
'drop_column' => '@yii/views/dropColumnMigration.php',
'create_junction' => '@yii/views/createTableMigration.php'
```

  ]‘)。マイグレーション・コードを生成するためのテンプレート・ファイルを指定します。 詳細は “マイグレーションを生成する“ を参照してください。

- fields: マイグレーション・コードを生成するためのカラム定義文字列の配列。 デフォルト値は []。個々の定義の書式は COLUMN_NAME:COLUMN_TYPE:COLUMN_DECORATOR です。 例えば、--fields=name:string(12):notNull は、サイズが 12 の null でない文字列カラムを作成します。

次の例は、これらのオプションの使い方を示すものです。

例えば、forum モジュールにマイグレーションを適用しようとしており、 そのマイグレーション・ファイルがモジュールの migrations ディレクトリに配置されている場合、 次のコマンドを使うことが出来ます。

```
# forum モジュールのマイグレーションを非対話的に適用する
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

コマンドをグローバルに構成する

マイグレーション・コマンドを実行するたびに同じオプションの値を入力する代りに、次のように、 アプリケーションの構成情報でコマンドを一度だけ構成して済ませることが出来ます。

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

上記のように構成しておくと、migrate コマンドを実行するたびに、backend_migration テーブルがマイグレーション履歴を記録するために使われるようになります。 もう、migrationTable のコマンドライン・オプションを使ってテーブルを指定する必要はなくなります。

名前空間を持つマイグレーション

2.0.10 以降では、マイグレーションのクラスに名前空間を適用することが出来ます。 マイグレーションの名前空間のリストをを migrationNamespaces によって指定することが出来ます。 マイグレーションのクラスに名前空間を使うと、マイグレーションのソースについて、複数の配置場所を使用することが出来ます。例えば、

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => null, // app\migrations が下記にあげられている
    場合に、名前空間に属さないマイグレーションを無効化する
            'migrationNamespaces' => [
                'app\migrations', // アプリケーション全体のための共通のマイグ
    レーション
```

```
                'module\migrations', // プロジェクトの特定のモジュールのための
    マイグレーション
                'some\extension\migrations', // 特定のエクステンションのため
    のマイグレーション
            ],
        ],
    ],
];
```

> 補足: 異なる名前空間に属するマイグレーションを適用しても、単一の マイグレーション履歴が生成されます。 つまり、特定の名前空間に属するマイグレーションだけを適用したり元に戻したりすることは出来ません。

名前空間を持つマイグレーションを操作するときは、新規作成時も、元に戻すときも、マイグレーション名の前にフルパスの名前空間を指定しなければなりません。 バック・スラッシュ (\) のシンボルは、通常、シェルでは特殊文字として扱われますので、シェルのエラーや誤った動作を防止するために、 適切にエスケープしなければならないことに注意して下さい。例えば、

```
yii migrate/create app\\migrations\\createUserTable
```

> 補足: `migrationPath` によって指定されたマイグレーションは、 名前空間を持つことが出来ません。 名前空間を持つマイグレーションは `yii\console\controllers\MigrateController::$migrationNamespaces` プロパティを通じてのみ適用可能です。

バージョン 2.0.12 以降は `migrationPath` プロパティは 名前空間を持たないマイグレーションを含む複数のディレクトリを指定した配列を受け入れるようになりました。 この機能追加は、主として、いろんな場所にあるマイグレーションを使っている既存のプロジェクトによって使われることを意図しています。 これらのマイグレーションは、主として、他の開発者による Yii エクステンションなど、外部ソースに由来するものであり、 新しい手法を使い始めようとしても、名前空間を使うように変更することが簡単には出来ないものだからです。

名前空間を持つマイグレーションを生成する　名前空間を持つマイグレーションは “CamelCase” の命名規則 `M<YYMMDDHHMMSS><Name>` (例えば `M190720100234CreateUserTable`) を持ちます。 このようなマイグレーションを生成するときは、テーブル名が “CamenCase” 形式から “アンダースコア” 形式に変換されることを 覚えておいて下さい。例えば、

```
yii migrate/create app\\migrations\\DropGreenHotelTable
```

上記のコマンドは、名前空間 app\migrations の中で、green_hotel というテーブルを削除するマイグレーションを生成します。そして、

```
yii migrate/create app\\migrations\\CreateBANANATable
```

というコマンドは、名前空間 app\migrations の中で b_a_n_a_n_a というテーブルを作成するマイグレーションを生成します。

テーブル名が大文字と小文字を含む（例えば studentsExam）ときは、名前の先頭にアンダースコアを付ける必要があります。

```
yii migrate/create app\\migrations\\Create_studentsExamTable
```

このコマンドは、名前空間 app\migrations の中で studentsExam というテーブルを作成するマイグレーションを生成します。

分離されたマイグレーション

プロジェクトのマイグレーション全体に単一のマイグレーション履歴を使用することが望ましくない場合もあります。 例えば、完全に独立した機能性とそれ自身のためのマイグレーションを持つような 'blog' エクステンションをインストールする場合には、 メインのプロジェクトの機能専用のマイグレーションに影響を与えたくないでしょう。

これらをお互いに完全に分離して適用かつ追跡したい場合は、別々の名前空間とマイグレーション履歴テーブルを使う 複数のマイグレーションコマンドを構成することが出来ます。

```
return [
    'controllerMap' => [
        // アプリケーション全体のための共通のマイグレーション
        'migrate-app' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['app\migrations'],
            'migrationTable' => 'migration_app',
            'migrationPath' => null,
        ],
        // 特定のモジュールのためのマイグレーション
        'migrate-module' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['module\migrations'],
            'migrationTable' => 'migration_module',
            'migrationPath' => null,
        ],
        // 特定のエクステンションのためのマイグレーション
        'migrate-rbac' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => '@yii/rbac/migrations',
            'migrationTable' => 'migration_rbac',
        ],
    ],
];
```

データベースを同期するためには、一つではなく複数のコマンドを実行しなければならなくなることに注意してください。

```
yii migrate-app
yii migrate-module
yii migrate-rbac
```

6.4.10 複数のデータベースにマイグレーションを適用する

デフォルトでは、マイグレーションは db アプリケーション・コンポーネント によって指定された同じデータベースに対して適用されます。 マイグレーションを別のデータベースに適用したい場合は、次のように、db コマンドライン・オプションを指定することが出来ます。

```
yii migrate --db=db2
```

上記のコマンドはマイグレーションを db2 データベースに適用します。

場合によっては、いくつかの マイグレーションはあるデータベースに適用し、別のいくつかの マイグレーションはもう一つのデータベースに適用したい、ということがあります。 この目的を達するためには、マイグレーション・クラスを実装する時に、そのマイグレーションが使用する DB コンポーネントの ID を明示的に指定しなければなりません。 例えば、次のようにします。

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

上記のマイグレーションは、db コマンドライン・オプションで別のデータベースを指定した場合でも、db2 に対して適用されます。 ただし、マイグレーション履歴は、db コマンドライン・オプションで指定されたデータベースに記録されることに注意してください。

同じデータベースを使う複数のマイグレーションがある場合は、上記の init() コードを持つ基底のマイグレーション・クラスを作成することを推奨します。 そうすれば、個々のマイグレーション・クラスは、その基底クラスから拡張することが出来ます。

> ヒント: 異なるデータベースを操作するためには、db プロパティを設定する以外にも、 マイグレーション・クラスの中で新しいデータベース接続を作成するという方法があります。 そうすれば、そのデータベース接続で DAO メソッド を使って、違うデータベースを操作することが出来ます。

複数のデータベースに対してマイグレーションを適用するために採用できるもう一つの戦略としては、異なるデータベースに対するマイグレーションは異なるマイグレーションパスに保持する、というものがあります。 そうすれば、次のように、異なるデータベースのマイグレーションを別々のコマンドで適用することが出来ます。

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

最初のコマンドは `@app/migrations/db1` にあるマイグレーションを `db1` データベースに適用し、 第二のコマンドは `@app/migrations/db2` にあるマイグレーションを `db2` データベースに適用する、という具合です。

Chapter 7

ユーザからのデータ取得

7.1 フォームを作成する

7.1.1 アクティブ・レコードに基づくフォーム： ActiveForm

Yii においてフォームを使用するときは、主として `yii\widgets\ActiveForm` による方法を使います。 フォームがモデルに基づくものである場合はこの方法を選ぶべきです。 これに加えて、`yii\helpers\Html` にはいくつかの有用なメソッドがあり、どんなフォームでも、ボタンやヘルプ・テキストを追加するのには、通常、それらのメソッドを使います。

フォームは、クライアント・サイドで表示されるものですが、たいていの場合、対応する モデル を持ち、それを使ってサーバ・サイドでフォームの入力を検証します (入力の検証の詳細については、入力を検証する のセクションを参照してください)。 モデルに基づくフォームを作成する場合、最初のステップは、モデルそのものを定義することです。 モデルは、データベースの何らかのデータを表現するために アクティブ・レコード から派生させたクラスか、 あるいは、任意の入力、例えばログイン・フォームの入力を保持するための (`yii\base\Model` から派生させた) 汎用的な Model クラスか、どちらかにすることが出来ます。

> ヒント: フォームのフィールドがデータベースのカラムと異なっていたり、そのフォーム特有のフォーマット形式やロジックがあったりする場合は、 `yii\base\Model` を拡張した独自のモデルを作るほうを選んで下さい。

以下の例においては、ログイン・フォームのために汎用的なモデルを使う方法を示します。

```
<?php

class LoginForm extends \yii\base\Model
{
    public $username;
    public $password;
```

```
    public function rules()
    {
        return [
            // 検証規則をここで定義
        ];
    }
```

コントローラにおいて、このモデルのインスタンスをビューに渡し、ビューでは ActiveForm ウィジェットがフォームを表示するのに使われます。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <div class="col-lg-offset-1 col-lg-11">
            <?= Html::submitButton('ログイ
    ン', ['class' => 'btn btn-primary']) ?>
        </div>
    </div>
<?php ActiveForm::end() ?>
```

`begin()` と `end()` で囲む

上記のコードでは、`ActiveForm::begin()` がフォームのインスタンスを作成するとともに、フォームの開始をマークしています。 `ActiveForm::begin()` と `ActiveForm::end()` の間に置かれた全てのコンテントが HTML の `<form>` タグによって囲まれます。 どのウィジェットでも同じですが、ウィジェットをどのように構成すべきかに関するオプションを指定するために、`begin` メソッドに配列を渡すことが出来ます。 この例では、追加の CSS クラスと要素を特定するための ID が渡されて、`<form>` の開始タグに適用されています。 利用できるオプションの全ては `yii\widgets\ActiveForm` の API ドキュメントに記されていますので参照してください。

ActiveField

フォームの中では、フォームの要素を作成するために、ActiveForm ウィジェットの `ActiveForm::field()` メソッドが呼ばれています。 このメソッドは、フォームの要素だけでなく、そのラベルも作成し、適用でき

る JavaScript の検証メソッドがあれば、それも追加します。ActiveForm ::field() メソッドは、 yii\widgets\ActiveField のインスタンスを返します。このメソッドの呼び出し結果を直接にエコーすると、結果は通常の (text の) インプットになります。 このメソッドの呼び出しに追加の ActiveField のメソッドをチェーンして、出力結果をカスタマイズすることが出来ます。

```
// パスワードのインプット
<?= $form->field($model, 'password')->passwordInput() ?>
// ヒントとカスタマイズしたラベルを追加
<?= $form->field($model, 'username')->textInput()->hint('お名前を入力してくださ
    い')->label('お名前') ?>
// HTML5 のメール・インプット要素を作成
<?= $form->field($model, 'email')->input('email') ?>
```

これで、フォームのフィールドによって定義された テンプレート に従って、<label>、<input> など、全てのタグが生成されます。 インプット・フィールドの名前は、モデルの フォーム名 と属性から自動的に決定されます。 例えば、上記の例における username 属性のインプット・フィールドの名前は LoginForm[username] となります。 この命名規則の結果として、ログイン・フォームの全ての属性が配列として、サーバ・サイドにおいては $_POST['LoginForm'] に格納されて利用できることになります。

> ヒント: 一つのフォームに一つのモデルだけがある場合、インプットの名前を単純化したいときは、 モデルの formName() メソッドをオーバーライドして空文字列を返すようにして、配列の部分をスキップすることが出来ます。 この方法を使えば、GridView で使われるフィルター・モデルで、もっと見栄えの良い URL を生成させることが出来ます。

モデルの属性を指定するために、もっと洗練された方法を使うことも出来ます。 例えば、複数のファイルをアップロードしたり、複数の項目を選択したりする場合に、属性の名前に [] を付けて、 属性が配列の値を取り得ることを指定することが出来ます。

```
// 複数のファイルのアップロードを許可する
echo $form->field($model, 'uploadFile[]')->fileInput(['multiple'=>'multiple'
    ]);

// 複数の項目をチェックすることを許可する
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' =>
    'Item B', 'c' => 'Item C']);
```

送信ボタンなどのフォーム要素に名前をつけるときには注意が必要です。 jQuery ドキュメント[1] によれば、衝突を生じさせ得る予約された名前がいくつかあります。

[1] https://api.jquery.com/submit/

> フォームおよびフォームの子要素は、フォームのプロパティと衝突するインプット名や id、たとえば `submit`、`length`、`method` などを使ってはなりません。 名前の衝突は訳の分らない失敗を生じさせることがあります。 命名規則の完全なリストを知り、この問題についてあなたのマークアップをチェックするためには、DOMLint[2] を参照してください。

フォームに HTML タグを追加するためには、素の HTML を使うか、または、上記の例の `Html::submitButton()` のように、 `Html` ヘルパ・クラスのメソッドを使うことが出来ます。

> ヒント: あなたのアプリケーションで Twitter Bootstrap CSS を使っている場合は、`yii\widgets\ActiveForm` の代りに `yii\bootstrap\ActiveForm` を使うのが良いでしょう。 後者は前者の拡張であり、bootstrap CSS フレームワークで使用するための追加のスタイルをサポートしています。
>
> ヒント: 必須フィールドをアスタリスク付きのスタイルにするために、次の CSS を使うことが出来ます。

```
div.required label.control-label:after {
    content: " *";
    color: red;
}
```

7.1.2 リストを作る

三種類のリストがあります:

- ドロップダウン・リスト
- ラジオ・リスト
- チェックボックス・リスト

リストを作るためには、項目の配列を準備しなければなりません。これは、手作業でやることも出来ます。

```
$items = [
    1 => '項目 1',
    2 => '項目 2'
]
```

または、DB から取得することも出来ます。

```
$items = Category::find()
        ->select(['label'])
        ->indexBy('id')
        ->column();
```

このような `$items` が、いろんなリスト・ウィジェットによって処理されるべきものとなります。 フォームのフィールドの値(および現在アクティブな項目)は、 `$model` の属性の現在の値に従って自動的に設定されます。

[2] http://kangax.github.io/domlint/

ドロップダウン・リストを作る　ActiveField の yii\widgets\ActiveField ::dropDownList() メソッドを使って、ドロップダウン・リストを作ることが出来ます。

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->dropdownList([
        1 => '項目 1',
        2 => '項目 2'
    ],
    ['prompt'=>'カテゴリーを選択してください']
);
```

ラジオ・リストを作る　ActiveField の yii\widgets\ActiveField::radioList() メソッドを使ってラジオ・リストを作ることが出来ます。

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->radioList([
    1 => 'ラジオ 1',
    2 => 'ラジオ 2'
]);
```

チェックボックス・リストを作る　ActiveField の yii\widgets\ActiveField ::checkboxList() メソッドを使ってチェックボックス・リストを作ることが出来ます。

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->checkboxList([
    1 => 'チェックボックス 1',
    2 => 'チェックボックス 2'
]);
```

7.1.3 Pjax を使う

Pjax ウィジェットを使うと、ページ全体をリロードせずに、 ページの一部分だけを更新することが出来ます。 これを使うと、送信後にフォームだけを更新して、その中身を入れ替えることが出来ます。

$formSelector を構成すると、どのフォームの送信が pjax を起動するかを指定することが出来ます。 それが指定されていない場合は、Pjax に囲まれたコンテントの中にあって data-pjax 属性を持つすべてのフォームが pjax リクエストを起動することになります。

```
use yii\widgets\Pjax;
use yii\widgets\ActiveForm;

Pjax::begin([
    // Pjax のオプション
```

```
]);
    $form = ActiveForm::begin([
        'options' => ['data' => ['pjax' => true]],
        // ActiveForm の追加のオプション
    ]);

        // ActiveForm のコンテント

    ActiveForm::end();
Pjax::end();
```

> ヒント: Pjax ウィジェット内部のリンクに注意してください。 と言うのは、リンクに対するレスポンスもウィジェット内部でレンダリングされるからです。 これを防ぐためには、`data-pjax="0"` という HTML 属性を使用します。

送信ボタンの値とファイルのアップロード `jQuery.serializeArray()` については、 ファイル[3] および 送信ボタンの値[4] を扱うときに問題があることが知られています。 この問題は解決される見込みがなく、関数自体も HTML5 で導入された `FormData` クラスによって置き換えられるべきものとして、廃止予定となっています。

このことは、すなわち、ajax または Pjax ウィジェットを使う場合、ファイルと送信ボタンの値に対する唯一の公式なサポートは、 `FormData` クラスに対する ブラウザのサポート[5] に依存しているということを意味します。

7.1.4 さらに読むべき文書

次のセクション 入力を検証する は、送信されたフォームデータのサーバ・サイドでの検証と、ajax 検証およびクライアント・サイドでの検証を扱います。

フォームのもっと複雑な使用方法については、以下のセクションを読んで下さい。

- 表形式インプットのデータ収集 - 同じ種類の複数のモデルのデータを収集する。
- 複数のモデルのデータを取得する - 同じフォームの中で複数の異なるモデルを扱う。
- ファイルをアップロードする - フォームを使ってファイルをアップロードする方法。

[3] https://github.com/jquery/jquery/issues/2321
[4] https://github.com/jquery/jquery/issues/2321
[5] https://developer.mozilla.org/en-US/docs/Web/API/FormData#Browser_compatibility

7.2 入力を検証する

経験則として言えることは、エンド・ユーザから受信したデータは決して信用せず、 利用する前に検証しなければならない、ということです。

モデル にユーザの入力が投入されたら、モデルの yii\base\Model::validate() メソッドを呼んで入力を検証することが出来ます。 このメソッドは検証が成功したか否かを示す真偽値を返します。 検証が失敗した場合は、yii\base\Model::$errors プロパティからエラー・メッセージを取得することが出来ます。例えば、

```
$model = new \app\models\ContactForm();

// モデルの属性にユーザ入力を投入する
$model->load(\Yii::$app->request->post());
// これは次と等価
// $model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // 全ての入力が有効
} else {
    // 検証が失敗。$errors はエラー・メッセージを含む配列
    $errors = $model->errors;
}
```

7.2.1 規則を宣言する

validate() を現実に動作させるためには、検証する予定の属性に対して検証規則を宣言しなければなりません。 規則は yii\base\Model::rules() メソッドをオーバーライドすることで宣言します。 次の例は、ContactForm モデルに対して検証規則を宣言する方法を示すものです。

```
public function rules()
{
    return [
        // 名前、メール・アドレス、主題、本文が必須項目
        [['name', 'email', 'subject', 'body'], 'required'],

        // email 属性は有効なメール・アドレスでなければならない
        ['email', 'email'],
    ];
}
```

rules() メソッドは規則の配列を返すべきものですが、 その配列の各要素は次の形式の配列でなければなりません。

```
[
    // 必須。この規則によって検証されるべき属性を指定する。
    // 属性が一つだけの場合は、属性の名前を直接に書いてもよい。
    // 配列の中に入れずに()
```

```
    ['属性1', '属性2', ...],

    // 必須。この規則のタイプを指定する。
    // クラス名、バリデータのエイリアス、または、バリデーション・メソッドの名
    前。
    'バリデータ',

    // オプション。この規則が適用されるべき一つまたは複数のシナリオを指定する。
    // 指定しない場合は、この規則が全てのシナリオに適用されることを意味する。
    // "except" オプションを構成して、列挙したシナリオを除く全てのシナリオに
    // この規則が適用されるべきことを指定してもよい。
    'on' => ['シナリオ1', 'シナリオ2', ...],

    // オプション。バリデータ・オブジェクトに対する追加の構成情報を指定する。
    'プロパティ1' => '値1', 'プロパティ2' => '値2', ...
]
```

各規則について、最低限、規則がどの属性に適用されるか、そして、規則がどのタイプであるかを指定しなければなりません。 規則のタイプは、次に挙げる形式のどれか一つを選ぶことが出来ます。

- コア・バリデータのエイリアス。例えば、`required`、`in`、`date`、等々。コア・バリデータの完全なリストは コア・バリデータ を参照してください。
- モデル・クラス内のバリデーション・メソッドの名前、または無名関数。 詳細は、インライン・バリデータ の項を参照してください。
- 完全修飾のバリデータ・クラス名。詳細は スタンドアロン・バリデータ の項を参照してください。

一つの規則は、一つまたは複数の属性を検証するために使用することが出来ます。そして、一つの属性は、一つまたは複数の規則によって検証され得ます。 `on` オプションを指定することで、規則を特定の シナリオ においてのみ適用することが出来ます。 `on` オプションを指定しない場合は、規則が全てのシナリオに適用されることになります。

`validate()` メソッドが呼ばれると、次のステップを踏んで検証が実行されます。

1. 現在の シナリオ を使って `yii\base\Model::scenarios()` から属性のリストを取得し、どの属性が検証されるべきかを決定します。 検証されるべき属性が アクティブな属性 と呼ばれます。

2. 現在の シナリオ を使って `yii\base\Model::rules()` から規則のリストを取得し、どの検証規則が使用されるべきかを決定します。 使用されるべき規則が アクティブな規則 と呼ばれます。

3. 全てのアクティブな規則を一つずつ使って、その規則に関連付けられた全てのアクティブな属性を一つずつ検証します。 検証規則はリストに挙げられている順に評価されます。

属性は、上記の検証のステップに従って、scenarios() でアクティブな属性であると宣言されており、 かつ、rules() で宣言された一つまたは複数のアクティブな規則と関連付けられている場合に、 また、その場合に限って、検証されます。

補足: 規則に名前を付けると便利です。すなわち、

```
public function rules()
{
    return [
        // ...
        'password' => [['password'], 'string', 'max' => 60],
    ];
}
```

これを子のモデルで使うことが出来ます。

```
public function rules()
{
    $rules = parent::rules();
    unset($rules['password']);
    return $rules;
}
```

エラー・メッセージをカスタマイズする

たいていのバリデータはデフォルトのエラー・メッセージを持っていて、属性の検証が失敗した場合にそれを検証の対象であるモデルに追加します。 例えば、required バリデータは、このバリデータを使って username 属性を検証したとき、 規則に合致しない場合は「ユーザ名は空ではいけません。」というエラー・メッセージをモデルに追加します。

規則のエラー・メッセージは、次に示すように、規則を宣言するときに message プロパティを指定することによってカスタマイズすることが出来ます。

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'ユーザ名を選んでください。'],
    ];
}
```

バリデータの中には、検証を失敗させたさまざまな原因をより詳しく説明するための追加のエラー・メッセージをサポートしているものがあります。 例えば、number バリデータは、検証される値が大きすぎたり小さすぎたりしたときに、 検証の失敗を説明するために、それぞれ、tooBig および tooSmall のメッセージをサポートしています。 これらのエラー・メッセージも、バリデータの他のプロパティと同様、検証規則の中で構成することが出来ます。

検証のイベント

yii\base\Model::validate() は、呼び出されると、 検証のプロセスをカスタマイズするためにオーバーライドできる二つのメソッドを呼び出します。

- yii\base\Model::beforeValidate(): デフォルトの実装は yii\base\Model::EVENT_BEFORE_VALIDATE イベントをトリガするものです。このメソッドをオーバーライドするか、または、イベントに反応して、検証が実行される前に、 何らかの前処理 (例えば入力されたデータの正規化) をすることが出来ます。 このメソッドは、検証を続行すべきか否かを示す真偽値を返さなくてはなりません。
- yii\base\Model::afterValidate(): デフォルトの実装は yii\base\Model::EVENT_AFTER_VALIDATE イベントをトリガするものです。このメソッドをオーバーライドするか、または、イベントに反応して、検証が完了した後に、 何らかの後処理をすることが出来ます。

条件付きの検証

特定の条件が満たされる場合に限って属性を検証したい場合、例えば、ある属性の検証が他の属性の値に依存する場合には、 when プロパティを使って、そのような条件を定義することが出来ます。 例えば、

```
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }],
```

when プロパティは、次のシグニチャを持つ PHP コーラブルを値として取ります。

```
/**
 * @param Model $model 検証されるモデル
 * @param string $attribute 検証される属性
 * @return bool 規則が適用されるか否か
 */
function ($model, $attribute)
```

クライアント・サイドでも条件付きの検証をサポートする必要がある場合は、whenClient プロパティを構成しなければなりません。 このプロパティは、規則を適用すべきか否かを返す JavaScript 関数を表す文字列を値として取ります。 例えば、

```
    ['state', 'required', 'when' => function ($model) {
        return $model->country == 'USA';
    }, 'whenClient' => "function (attribute, value) {
        return $('#country').val() == 'USA';
    }"]
```

データのフィルタリング

ユーザ入力をフィルタまたは前処理する必要があることがよくありま

す。 例えば、username の入力値の前後にある空白を除去したいというような場合です。この目的を達するために検証規則を使うことが出来ます。

次の例では、入力値の前後にある空白を除去して、空の入力値を null に変換することを、 trim および default のコア・バリデータで行っています。

```
return [
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
];
```

もっと汎用的な filter バリデータを使って、 もっと複雑なデータ・フィルタリングをすることも出来ます。

お分かりのように、これらの検証規則は実際には入力を検証しません。 そうではなくて、検証される属性の値を処理して書き戻すのです。

ユーザ入力の完全な処理を次のサンプル・コードで示します。 これは、ある属性に整数の値だけが保存されるように保証しようとするものです。

```
['age', 'trim'],
['age', 'default', 'value' => null],
['age', 'integer', 'min' => 0],
['age', 'filter', 'filter' => 'intval', 'skipOnEmpty' => true],
```

上記のコードは入力に対して以下の操作を実行します。

1. 入力値から先頭と末尾のホワイト・スペースをトリムします。
2. 空の入力値がデータベースで null として保存されることを保証します。 “not set(未設定)” という値と、実際の値である 0 は区別します。null が許されない時は、ここで別のデフォルト値を設定することが出来ます。
3. 空でない場合は、値は 0 以上の整数であることを検証します。 通常のバリデータでは $skipOnEmpty が true に設定されています。
4. 例えば、文字列 '42' は、整数 42 にキャストして、値が整数型になることを保証します。 デフォルトでは false である filter バリデータの $skipOnEmpty をtrue に設定しています。

空の入力値を扱う

HTML フォームから入力データが送信されたとき、入力値が空である場合には何らかのデフォルト値を割り当てなければならないことがよくあります。 default バリデータを使ってそうすることが出来ます。例えば、

```
return [
    // 空の時は "username" と "email" を null にする
    [['username', 'email'], 'default'],
```

```
    // 空の時は "level" を 1 にする
    ['level', 'default', 'value' => 1],
];
```

デフォルトでは、入力値が空であると見なされるのは、それが、空文字列であるか、空配列であるか、null であるときです。 空を検知するこのデフォルトのロジックは、yii\validators\Validator::isEmpty() プロパティを PHP コーラブルで構成することによって、 カスタマイズすることが出来ます。例えば、

```
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }]
```

> 補足: たいていのバリデータは、yii\validators\Validator::$skipOnEmpty プロパティがデフォルト値 true を取っている場合は、 空の入力値を処理しません。そのようなバリデータは、関連付けられた属性が空の入力値を受け取ったときは、検証の過程ではスキップされるだけになります。コア・バリデータ の中では、 captcha、default、filter、required、そして trim だけが空の入力値を処理します。

7.2.2 その場限りの検証

時として、何らかのモデルに結び付けられていない値に対する その場限りの検証 を実行しなければならない場合があります。

実行する必要がある検証が一種類 (例えば、メール・アドレスの検証) だけである場合は、 使いたいバリデータの validate() メソッドを次のように呼び出すことが出来ます。

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'メール・アドレスは有効。';
} else {
    echo $error;
}
```

> 補足: 全てのバリデータがこの種の検証をサポートしている訳ではありません。 その一例が unique コア・バリデータであり、これはモデルとともに使用されることだけを前提にして設計されています。

> 補足: yii\base\Validator::skipOnEmpty プロパティは yii\base\Model の検証の場合にのみ使用されます。モデル無しで使っても効果はありません。

いくつかの値に対して複数の検証を実行する必要がある場合は、属性と規則の両方をその場で宣言することが出来る yii\base\DynamicModel を使うことが出来ます。 これは、次のような使い方をします。

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(['name' => $name, 'email' => $email
    ], [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // 検証が失敗
    } else {
        // 検証が成功
    }
}
```

yii\base\DynamicModel::validateData() メソッドは DynamicModel のインスタンスを作成し、 与えられた値 (この例では name と email) を使って属性を定義し、そして、与えられた規則で yii\base\Model::validate() を呼び出します。

別の選択肢として、次のように、もっと「クラシック」な構文を使って、その場限りのデータ検証を実行することも出来ます。

```
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));
    $model->addRule(['name', 'email'], 'string', ['max' => 128])
        ->addRule('email', 'email')
        ->validate();

    if ($model->hasErrors()) {
        // 検証が失敗
    } else {
        // 検証が成功
    }
}
```

検証を実行した後は、通常のモデルで行うのと同様に、検証が成功したか否かを hasErrors() メソッドを呼んでチェックして、 errors プロパティから検証エラーを取得することが出来ます。 また、このモデルのインスタンスによって定義された動的な属性に対しても、例えば $model->name や $model->email のようにして、アクセスすることが出来ます。

7.2.3 バリデータを作成する

Yii のリリースに含まれている コア・バリデータ を使う以外に、あなた自身のバリデータを作成することも出来ます。 インライン・バリデータとスタンドアロン・バリデータを作ることが出来ます。

インライン・バリデータ

インライン・バリデータは、モデルのメソッドまたは無名関数として定義されるバリデータです。 メソッド/関数 のシグニチャは、

```
/**
 * @param string $attribute 現在検証されている属性
 * @param mixed $params 規則に与えられる "params" の値
 * @param \yii\validators\InlineValidator $validator 関係す
    る InlineValidator のインスタンス。
 * このパラメータは、バージョン 2.0.11 以降で利用可能。
 * @param mixed $current 現在検証されている属性の値
 * このパラメータは、バージョン 2.0.36 以降で利用可能。
 */
function ($attribute, $params, $validator, $current)
```

属性が検証に失敗した場合は、メソッド/関数 は yii\base\Model::addError() を呼んでエラー・メッセージをモデルに保存し、 後で読み出してエンド・ユーザに表示することが出来るようにしなければなりません。

下記にいくつかの例を示します。

```
use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;

    public function rules()
    {
        return [
            // モデル・メソッド validateCountry() として定義されるインライン・
    バリデータ
            ['country', 'validateCountry'],

            // 無名関数として定義されるインライン・バリデータ
            ['token', function ($attribute, $params, $validator) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'トークンは英数字で構成しなけ
    ればなりません。');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params, $validator)
    {
        if (!in_array($this->$attribute, ['USA', 'Indonesia'])) {
            $this->addError($attribute, '国は "USA" または "Indonesia" でな
    ければなりません。');
        }
    }
```

```
}
```

補足: バージョン 2.0.11 以降では、代わりに、yii\validators\InlineValidator::addError() を使ってエラー・メッセージを追加することが出来ます。 そうすれば、エラー・メッセージはそのまま yii\i18n\I18N::format() を使ってフォーマットされます。 属性のラベルと値を参照するためには、それぞれ、{attribute} と {value} を使ってください(手作業で取得する必要はありません)。

```
$validator->addError($this, $attribute, 'The value "{value}" is
    not acceptable for {attribute}.');
```

補足: デフォルトでは、インライン・バリデータは、関連付けられている属性が空の入力値を受け取ったり、 既に何らかの検証規則に失敗したりしている場合には、適用されません。 規則が常に適用されることを保証したい場合は、 規則の宣言において skipOnEmpty および/または skipOnError のプロパティを false に設定することが出来ます。例えば、

```
[
    ['country', 'validateCountry', 'skipOnEmpty' => false, '
    skipOnError' => false],
]
```

スタンドアロン・バリデータ

スタンドアロン・バリデータは、yii\validators\Validator またはその子クラスを拡張するクラスです。 yii\validators\Validator::validateAttribute() メソッドをオーバーライドすることによって、その検証ロジックを実装することが出来ます。 インライン・バリデータでするのと同じように、属性が検証に失敗した場合は、 yii\base\Model::addError() を呼んでエラー・メッセージをモデルに保存します。

例えば、上記のインライン・バリデータは、新しい [[components/validators/CountryValidator]] クラスに作りかえることが出来ます。 この場合、yii\validators\Validator::addError() を使って特製のメッセージをモデルに設定することが出来ます。

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Indonesia'])) {
```

```
            $this->addError($model, $attribute, '国は "{coutry1}" また
    は "{coutry2}" でなければなりませ
    ん。', ['country1' => 'USA', 'country2' => 'Indonesia']);
        }
    }
}
```

あなたのバリデータで、モデルを使わない値の検証をサポートしたい場合は、`yii\validators\Validator::validate()` もオーバーライドしなければなりません。または、`validateAttribute()` と `validate()` の代りに、`yii\validators\Validator::validateValue()` をオーバーライドしても構いません。 と言うのは、前の二つは、デフォルトでは、`validateValue()` を呼び出すことによって実装されているからです。

次の例は、上記のバリデータ・クラスをあなたのモデルの中でどのように使用することが出来るかを示すものです。

```
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::className()],
            ['email', 'email'],
        ];
    }
}
```

7.2.4 複数の属性の検証

時として、バリデータが複数の属性に関係する場合があります。次のようなフォームを考えてみてください。

```
class MigrationForm extends \yii\base\Model
{
    /**
     * 成人一人のための最低限の生活費
     */
    const MIN_ADULT_FUNDS = 3000;
    /**
     * こども一人のための最低限の生活費
     */
```

```
    const MIN_CHILD_FUNDS = 1500;

    public $personalSalary;    // 給与
    public $spouseSalary;      // 配偶者の給与
    public $childrenCount;     // こどもの数
    public $description;

    public function rules()
    {
        return [
            [['personalSalary', 'description'], 'required'],
            [['personalSalary', 'spouseSalary'], 'integer', 'min' => self::
    MIN_ADULT_FUNDS],
            ['childrenCount', 'integer', 'min' => 0, 'max' => 5],
            [['spouseSalary', 'childrenCount'], 'default', 'value' => 0],
            ['description', 'string'],
        ];
    }
}
```

バリデータを作成する

家族の収入が子ども達のために十分であるかどうかをチェックする必要があるとしましょう。 そのためには、`childrenCount` が 1 以上である場合にのみ実行される `validateChildrenFunds` というインライン・バリデータを作れば良いわけです。

検証されるすべての属性 (`['personalSalary', 'spouseSalary', 'childrenCount']`) にこのバリデータをアタッチすることは出来ない、ということに注意してください。 そのようにすると、同じバリデータが属性ごとに (合計で三回) 走ることになりますが、 属性のセット全体に対してこのバリデータを一度だけ走らせれば十分なのです。

これらの属性のどれを使っても構いません (あるいは、もっとも関係が深いと思うものを使ってください)。

```
['childrenCount', 'validateChildrenFunds', 'when' => function ($model) {
    return $model->childrenCount > 0;
}],
```

`validateChildrenFunds` の実装は次のようにすることが出来ます。

```
public function validateChildrenFunds($attribute, $params)
{
    $totalSalary = $this->personalSalary + $this->spouseSalary;
    // 配偶者の給与が指定されているときは、成人の最低生活費を倍にする
    $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 : self
    ::MIN_ADULT_FUNDS;
    $childFunds = $totalSalary - $minAdultFunds;
    if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
        $this->addError('childrenCount', '子どもの数に対して給与が不足してい
    ます。');
    }
```

```
}
```

この検証は属性一つだけに関係するものではないので、`$attribute` のパラメータは無視することが出来ます。

エラー・メッセージを追加する

複数の属性の場合のエラー・メッセージの追加は、フォームをどのように設計するかによって異なってきます。

- もっとも関係が深いとあなたが思うフィールドを選んで、その属性にエラー・メッセージを追加する。

```
$this->addError('childrenCount', '子どもの数に対して給与が不足しています。');
```

- 重要な複数の属性、または、すべての属性を選んで、同じエラー・メッセージを追加する。 メッセージを独立した変数に格納してから `addError` に渡せば、コードを DRY に保つことが出来ます。

```
$message = '子どもの数に対して給与が不足しています。';
$this->addError('personalSalary', $message);
$this->addError('wifeSalary', $message);
$this->addError('childrenCount', $message);
```

あるいは、ループを使います。

```
$attributes = ['personalSalary', 'wifeSalary', 'childrenCount'];
foreach ($attributes as $attribute) {
    $this->addError($attribute, '子どもの数に対して給与が不足しています。');
}
```

- (特定の属性に結び付かない) 共通のエラー・メッセージを追加する。 その時点では属性の存在はチェックされませんので、存在しない属性の名前、例えば `*` を使ってエラー・メッセージを追加することが出来ます。

```
$this->addError('*', '子どもの数に対して給与が不足しています。');
```

結果として、フォームのフィールドの近くにはこのエラー・メッセージは表示されません。これを表示するためには、ビューにエラー・サマリを含めます。

```
<?= $form->errorSummary($model) ?>
```

> 補足: 複数の属性を一度に検証するバリデータを作成する方法が community cookbook[6] で分り易く解説されています。.

[6] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-validator-multiple-attributes.md

7.2.5 クライアント・サイドでの検証

エンド・ユーザが HTML フォームで値を入力する際には、JavaScript に基づくクライアント・サイドでの検証を提供することが望まれます。 というのは、クライアント・サイドでの検証は、ユーザが入力のエラーを早く見つけることが出来るようにすることによって、より良いユーザ体験を提供するものだからです。 あなたも、サーバ・サイドでの検証 に加えて クライアント・サイドでの検証をサポートするバリデータを使用したり実装したりすることが出来ます。

> 情報: クライアント・サイドでの検証は望ましいものですが、不可欠なものではありません。 その主たる目的は、ユーザにより良い体験を提供することにあります。 エンド・ユーザから来る入力値と同じように、クライアント・サイドでの検証を決して信用してはいけません。 この理由により、これまでの項で説明したように、常に `yii\base\Model::validate()` を呼び出してサーバ・サイドでの検証を実行しなければなりません。

クライアント・サイドでの検証を使う

多くの コア・バリデータ は、そのままで、クライアント・サイドでの検証をサポートしています。 あなたがする必要があるのは、`yii\widgets\ActiveForm` を使って HTML フォームを作るということだけです。 例えば、下の `LoginForm` は二つの規則を宣言しています。 一つは、required コア・バリデータを使っていますが、これはクライアント・サイドとサーバ・サイドの両方でサポートされています。 もう一つは `validatePassword` インライン・バリデータを使っていますが、こちらはサーバ・サイドでのみサポートされています。

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // username と password はともに必須
            [['username', 'password'], 'required'],

            // password は validatePassword() によって検証される
            ['password', 'validatePassword'],
        ];
```

```
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'ユーザ名またはパスワードが違いま
    す。');
        }
    }
}
```

次のコードによって構築される HTML フォームは、username と password の二つの入力フィールドを含みます。 何も入力せずにこのフォームを送信すると、何かを入力するように要求するエラー・メッセージが、 サーバと少しも交信することなく、ただちに表示されることに気付くでしょう。

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('ログイン') ?>
<?php yii\widgets\ActiveForm::end(); ?>
```

舞台裏では、yii\widgets\ActiveForm がモデルで宣言されている検証規則を読んで、 クライアント・サイドの検証をサポートするバリデータのために、適切な JavaScript コードを生成します。 ユーザが入力フィールドの値を変更したりフォームを送信したりすると、クライアント・サイドの検証の JavaScript が起動されます。

クライアント・サイドの検証を完全に無効にしたい場合は、yii\widgets\ActiveForm::$enableClientValidation プロパティを false に設定することが出来ます。 また、個々の入力フィールドごとにクライアント・サイドの検証を無効にしたい場合には、 入力フィールドの yii\widgets\ActiveField::$enableClientValidation プロパティを false に設定することが出来ます。 eanbleClientValidation が入力フィールドのレベルとフォームのレベルの両方で構成されている場合は前者が優先されます。

> 情報: バージョン 2.0.11 以降、yii\validators\Validator を拡張する全てのバリデータは、クライアント・サイドのオプションを 独立したメソッド - yii\validators\Validator::getClientOptions() から受け取るようになりました。これを使うと、次のことが可能になります。
>
> - 独自のクライアント・サイド検証を実装しながら、 サーバ・サイド検証のオプションとの同期はそのまま残す
> - 特殊な要求に合うように拡張またはカスタマイズする
>
> ```
> public function getClientOptions($model, $attribute)
> ```

```
{
    $options = parent::getClientOptions($model, $attribute);
    // ここで $options を修正

    return $options;
}
```

クライアント・サイドの検証を実装する

クライアント・サイドの検証をサポートするバリデータを作成するためには、 クライアント・サイドでの検証を実行する JavaScript コードを返す `yii\validators\Validator::clientValidateAttribute()` メソッドを実装しなければなりません。 その JavaScript の中では、次の事前定義された変数を使用することが出来ます。

- `attribute`: 検証される属性の名前。
- `value`: 検証される値。
- `messages`: 属性に対する検証のエラー・メッセージを保持するために使用される配列。
- `deferred`: Deferred オブジェクトをプッシュして入れることが出来る配列 (次の項で説明します)。

次の例では、入力された値が既存のステータスのデータに含まれる有効なステータス値であるかどうかを検証する `StatusValidator` を作成します。 このバリデータは、サーバ・サイドとクライアント・サイドの両方の検証をサポートします。

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = '無効なステータスが入力されました。';
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;
        if (!Status::find()->where(['id' => $value])->exists()) {
            $model->addError($attribute, $this->message);
        }
    }

    public function clientValidateAttribute($model, $attribute, $view)
    {
        $statuses = json_encode(Status::find()->select('id')->asArray()->
    column());
```

```
        $message = json_encode($this->message, JSON_UNESCAPED_SLASHES |
    JSON_UNESCAPED_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

> ヒント: 上記のコード例の主たる目的は、クライアント・サイドの検証をサポートする方法を説明することにあります。 実際の仕事では、in コア・バリデータを使って、同じ目的を達することが出来ます。 次のように検証規則を書けばよいのです。
>
> ```
> [
>     ['status', 'in', 'range' => Status::find()->select('id')->
>     asArray()->column()],
> ]
> ```

> ヒント: クライアント・サイドの検証を手動で操作する必要がある場合、すなわち、動的にフィールドを追加したり、何か特殊な UI ロジックを実装する場合は、 Yii 2.0 Cookbook の Working with ActiveForm via JavaScript[7] を参照してください。

Deferred 検証

非同期のクライアント・サイドの検証をサポートする必要がある場合は、Defered オブジェクト[8] を作成することが出来ます。 例えば、AJAX によるカスタム検証を実行するために、次のコードを使うことが出来ます。

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

[7] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-activeform-js.md

[8] http://api.jquery.com/category/deferred-object/

上のコードにおいて、deferred は Yii が提供する変数で、Deferred オブジェクトの配列です。 jQuery の $.get() メソッドによって作成された Deferred オブジェクトが deferred 配列にプッシュされています。

Deferred オブジェクトを明示的に作成して、非同期のコールバックが呼ばれたときに、Deferred オブジェクトの resolve() メソッドを呼ぶことも出来ます。 次の例は、アップロードされる画像ファイルの大きさをクライアント・サイドで検証する方法を示すものです。

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('画像の幅が大きすぎます。');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> 補足: 属性が検証された後に、resolve() メソッドを呼び出さなければなりません。 そうしないと、主たるフォームの検証が完了しません。

簡潔に記述できるように、deferred 配列はショートカット・メソッド add() を装備しており、このメソッドを使うと、自動的に Deferred オブジェクトを作成して deferred 配列に追加することが出来ます。このメソッドを使えば、上記の例は次のように簡潔に記すことが出来ます。

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                  messages.push('画像の幅が大きすぎます。');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
```

```
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

7.2.6 AJAX 検証

場合によっては、サーバだけが必要な情報を持っているために、サーバ・サイドでしか検証が実行できないことがあります。 例えば、ユーザ名がユニークであるか否かを検証するためには、サーバ・サイドで user テーブルを調べることが必要になります。 このような場合には、AJAX ベースの検証を使うことが出来ます。 AJAX 検証は、通常のクライアント・サイドでの検証と同じユーザ体験を保ちながら、入力値を検証するためにバックグラウンドで AJAX リクエストを発行します。

単一のインプット・フィールドに対して AJAX 検証を有効にするためには、そのフィールドの `enableAjaxValidation` プロパティを true に設定し、フォームに一意の `id` を指定します。

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

フォームの全てのインプットに対して AJAX 検証を有効にするためには、フォームのレベルで `enableAjaxValidation` を true に設定します。

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> 補足: `enableAjaxValidation` プロパティがインプット・フィールドのレベルとフォームのレベルの両方で構成された場合は、前者が優先されます。

また、サーバ・サイドでは、AJAX 検証のリクエストを処理できるように準備しておく必要があります。 これは、コントローラのアクションにおいて、次のようなコード断片を使用することで達成できます。

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
    {
```

```
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

上記のコードは、現在のリクエストが AJAX であるかどうかをチェックします。 もし AJAX であるなら、リクエストに応えて検証を実行し、エラーを JSON 形式で返します。

> 情報: AJAX 検証を実行するためには、Deferred 検証 を使うことも出来ます。 しかし、ここで説明された AJAX 検証の機能の方がより体系化されており、コーディングの労力も少なくて済みます。

`enableClientValidation` と `enableAjaxValidation` が両方とも `true` に設定されているときは、 クライアント検証が成功した後でだけ AJAX 検証のリクエストが起動されます。 `validateOnChange`, `validateOnBlur` または `validateOnType` が `true` に設定されている単一のフィールドを検証する場合、 当該フィールドが単独でクライアント検証を通ったら AJAX リクエストが送信されることに注意して下さい。

7.3 ファイルをアップロードする

Yii におけるファイルのアップロードは、通常、アップロードされる個々のファイルを `UploadedFile` としてカプセル化する `yii\web\UploadedFile` の助けを借りて実行されます。これを `yii\widgets\ActiveForm` および モデル と組み合わせることで、安全なファイル・アップロード・メカニズムを簡単に実装することが出来ます。

7.3.1 モデルを作成する

プレーンなテキスト・インプットを扱うのと同じように、一つのファイルをアップロードするためには、モデル・クラスを作成して、 そのモデルの一つの属性を使ってアップロードされるファイルのインスタンスを保持します。 また、ファイルのアップロードを検証するために、検証規則も宣言しなければなりません。例えば、

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;
```

```
    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
    'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
     . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

上記のコードにおいては、imageFile 属性がアップロードされたファイルのインスタンスを保持するのに使われます。 この属性が関連付けられている file 検証規則は、yii\validators\FileValidator を使って、png または jpg の拡張子を持つファイルがアップロードされることを保証しています。 upload() メソッドは検証を実行して、アップロードされたファイルをサーバに保存します。

file バリデータによって、ファイル拡張子、サイズ、MIME タイプなどをチェックすることが出来ます。 詳細については、コア・バリデータのセクションを参照してください。

> ヒント: 画像をアップロードしようとする場合は、image バリデータを代りに使うことを考慮しても構いません。 image バリデータは yii\validators\ImageValidator によって実装されており、属性が有効な画像、すなわち、 保存したり Imagine エクステンション[9] を使って処理したりすることが可能な有効な画像を、受け取ったかどうかを検証します。

7.3.2 ファイル・インプットをレンダリングする

次に、ビューでファイル・インプットを作成します。

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-
    data']]) ?>

    <?= $form->field($model, 'imageFile')->fileInput() ?>
```

[9] https://github.com/yiisoft/yii2-imagine

```
    <button送信></button>

<?php ActiveForm::end() ?>
```

ファイルが正しくアップロードされるように、フォームに `enctype` オプションを追加することを憶えておくのは重要なことです。 `fileInput()` を呼ぶと `<input type="file">` のタグがレンダリングされて、ユーザがアップロードするファイルを選ぶことが出来るようになります。

> ヒント: バージョン 2.0.8 以降では、ファイル・インプットのフィールドが使われているときは、 `fileInput` がフォームに `enctype` オプションを自動的に追加します。

7.3.3 繋ぎ合せる

そして、コントローラ・アクションの中で、モデルとビューを繋ぎ合せるコードを書いて、ファイルのアップロードを実装します。

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model, 'imageFile
    ');
            if ($model->upload()) {
                // ファイルのアップロードが成功
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

上記のコードでは、フォームが送信されると `yii\web\UploadedFile::getInstance()` メソッドが呼ばれて、 アップロードされたファイルが `UploadedFile` のインスタンスとして表現されます。 そして、次に、モデルの検証によってアップロードされたファイルが有効なものであることを確かめ、サーバにファイルを保存します。

7.3.4 複数のファイルをアップロードする

ここまでの項で示したコードに若干の修正を加えれば、複数のファイルを一度にアップロードすることも出来ます。

最初に、モデル・クラスを修正して、`file` 検証規則に `maxFiles` オプションを追加して、アップロードを許可されるファイルの最大数を制限しなければなりません。 `maxFiles` を `0` に設定することは、同時にアップロード出来るファイル数に制限がないことを意味します。 同時にアップロードすることを許されるファイルの数は、また、PHP のディレクティブ `max_file_uploads`[10] (デフォルト値は 20) によっても制限されます。 `upload()` メソッドも、アップロードされた複数のファイルを一つずつ保存するように修正しなければなりません。

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
     'png, jpg', 'maxFiles' => 4],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' . $file->
    extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

ビュー・ファイルでは、`fileInput()` の呼び出しに `multiple` オプションを追加して、ファイル・アップロードのフィールドが複数のファイルを受け取ることが出来るようにしなければなりません。 また、`imageFiles` を

[10] https://secure.php.net/manual/ja/ini.core.php#ini.max-file-uploads

imageFiles[] に変更して、属性の値が配列として送信されるようにする必要があります。

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-
    data']]) ?>

    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' => true,
     'accept' => 'image/*']) ?>

    <button送信></button>

<?php ActiveForm::end() ?>
```

そして、最後に、コントローラ・アクションの中では、UploadedFile::getInstance() の代りに UploadedFile::getInstances() を呼んで、 UploadedFile インスタンスの配列を UploadForm::imageFiles に代入しなければなりません。

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFiles = UploadedFile::getInstances($model, '
    imageFiles');
            if ($model->upload()) {
                // ファイルのアップロードが成功
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

7.4 表形式インプットでデータを収集する

時として、一つのフォームで同じ種類の複数のモデルを扱わなければならないことがあります。 例えば、それぞれが「名前-値」の形で保存さ

れ、Setting アクティブ・レコード モデルとして表される複数の設定項目を扱うフォームです。 この種のフォームは「表形式インプット」と呼ばれることもよくあります。 これとは対照的な、異なる種類のさまざまなモデルを扱うことについては、複数のモデルを持つ複雑なフォーム のセクションで扱います。

以下に、表形式インプットを Yii で実装する方法を示します。

カバーすべき三つの異なる状況があり、それぞれ少しずつ異なる処理をしなければなりません。

- 特定の数のデータベース・レコードを更新する
- 不特定の数の新しいレコードを作成する
- 一つのページでレコードを更新、作成、および、削除する

前に説明した単一モデルのフォームとは対照的に、モデルの配列を扱うことになります。 この配列がビューに渡されて、各モデルのためのインプット・フィールドが表のような形式で表示されます。 そして、複数のモデルを一度にロードしたり検証したりするために yii\base\Model のヘルパ・メソッドを使用します。

- Model::loadMultiple() - 送信されたデータをモデルの配列にロードします。
- Model::validateMultiple() - モデルの配列を検証します。

特定の数のレコードを更新する

コントローラのアクションから始めましょう。

```
<?php

namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use app\models\Setting;

class SettingsController extends Controller
{
    // ...

    public function actionUpdate()
    {
        $settings = Setting::find()->indexBy('id')->all();

        if (Model::loadMultiple($settings, Yii::$app->request->post()) &&
    Model::validateMultiple($settings)) {
            foreach ($settings as $setting) {
                $setting->save(false);
            }
            return $this->redirect('index');
        }

        return $this->render('update', ['settings' => $settings]);
```

```
    }
}
```

上記のコードでは、データベースからモデルを読み出すときに indexBy() を使って、 モデルのプライマリ・キーでインデックスされた配列にデータを投入しています。このインデックスが、後で、 フォーム・フィールドを特定するために使われます。Model::loadMultiple() が POST から来るフォーム・データを複数のモデルに代入し、Model::validateMultiple() が全てのモデルを一度に検証します。 保存するときには、validateMultiple() を使ってモデルの検証を済ませていますので、save() のパラメータに false を渡して、二度目の検証を実行しないようにしています。

次に、update ビューの中にあるフォームです。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo $form->field($setting, "[$index]value")->label($setting->name);
}

ActiveForm::end();
```

ここで全ての設定項目について、それぞれ、項目名を示すラベルと、項目の値を入れたインプットをレンダリングしています。 インプットの名前に適切なインデックスを追加することが肝腎です。というのは、loadMultiple がそれを見て、どのモデルにどの値を代入するかを決定するからです。

不特定の数の新しいレコードを動的に作成する

新しいレコードを作成するのは、モデルのインスタンスを作成する部分を除いて、更新の場合と同じです。

```
public function actionCreate()
{
    $count = count(Yii::$app->request->post('Setting', []));
    $settings = [new Setting()];
    for($i = 1; $i < $count; $i++) {
        $settings[] = new Setting();
    }

    // ...
}
```

ここでは、デフォルトで一個のモデルを含む $settings 配列を初期値として作成し、少なくとも一個のテキスト・フィールドが常にビューに表示されるようにしています。 そして、受信したインプットの行数に合せて、配列にモデルを追加しています。

ビューでは javascript を使ってインプットの行を動的に追加することが出来ます。

更新、作成、削除を一つのページに組み合わせる

> 補足: このセクションはまだ執筆中です。
>
> まだ内容がありません。

(未定)

7.5 複数のモデルのデータを取得する

複雑なデータを扱う場合には、複数の異なるモデルを使用してユーザの入力を収集する必要があることがあり得ます。 例えば、ユーザのログイン情報は `user` テーブルに保存されているけれども、ユーザのプロファイル情報は `profile` テーブルに保存されているという場合を考えて見ると、ユーザに関して入力されたデータを `User` モデルと `Profile` モデルによって収集しなければならないでしょう。 Yii のモデルとフォームのサポートを使えば、単一のモデルを扱うのとそれほど違いのない方法によってこの問題を解決することが出来ます。

下記において、`User` と `Profile` の二つのモデルのデータを収集することが出来るフォームをどのようにして作成することが 出来るかを示します。

最初に、ユーザとプロファイルのデータを収集するためのコントローラ・アクションは、次のように書くことが出来ます。

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
            throw new NotFoundHttpException("ユーザが見つかりませんでし
    た。");
        }

        $profile = Profile::findOne($id);

        if (!$profile) {
```

```
            throw new NotFoundHttpException("ユーザのプロファイルがありませ
    ん。");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) && $profile->load(Yii::
    $app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}
```

この update アクションでは、最初に、更新の対象になる $user と $profile のモデルをデータベースからロードします。 次に yii\base\Model::load() を呼んで、これら二つのモデルにユーザ入力を代入します。 代入が成功すれば、二つのモデルを検証して保存します。 — モデルの中では、ユーザの入力データは既に検証済みであるため、過剰な検証を避けるために save(false) を使っていることに注意して下さい。 そうでない場合は、次の内容を持つ update ビューをレンダリングします。

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...other input fields...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('更新', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

ご覧のように、update ビューでは、二つのモデル、すなわち $user と $profile を使ってインプット・フィールドをレンダリングすることになります。

7.6 クライアント・サイドで ActiveForm を拡張する

yii\widgets\ActiveForm ウィジェットは、クライアント・サイドの検証に使う一連の JavaScript メソッドを備えています。 その実装は非常に柔軟で、様々な方法で拡張することが可能になっています。 下記でそれについて解説します。

7.6.1 ActiveForm イベント

ActiveForm は、一連の専用のイベントを発生させます。 次のようなコードを使って、これらのイベントを購読して処理することが出来ます。

```
$('#contact-form').on('beforeSubmit', function (e) {
  if (!confirm("全てオーケー。送信しますか?")) {
    return false;
  }
  return true;
});
```

以下、利用できるイベントを見ていきましょう。

beforeValidate

beforeValidate は、フォーム全体を検証する前にトリガされます。
イベント・ハンドラのシグニチャは以下の通り:

```
function (event, messages, deferreds)
```

引数は以下の通り:

- event: イベントのオブジェクト。
- messages: 連想配列で、キーは属性の ID、 値は対応する属性のエラー・メッセージの配列です。
- deferreds: Deferred オブジェクトの配列。deferreds.add(callback) を使って、 新しい deferrd な検証を追加することが出来ます。

ハンドラが真偽値 false を返すと、このイベントに続くフォームの検証は中止されます。 その結果、afterValidate イベントもトリガされません。

afterValidate

afterValidate イベントは、フォーム全体を検証した後でトリガされます。
イベント・ハンドラのシグニチャは以下の通り:

```
function (event, messages, errorAttributes)
```

引数は以下の通り:

- event: イベントのオブジェクト。
- messages: 連想配列で、キーは属性の ID、 値は対応する属性のエラー・メッセージの配列です。
- errorAttributes: 検証エラーがある属性の配列。 この引数の構造については attributeDefaults を参照して下さい。

beforeValidateAttribute

beforeValidateAttribute イベントは、属性を検証する前にトリガされます。 イベント・ハンドラのシグニチャは以下の通り:

```
function (event, attribute, messages, deferreds)
```

引数は以下の通り:

- event: イベントのオブジェクト。
- attribute: 検証される属性。 この引数の構造については attributeDefaults を参照して下さい。
- messages: 指定された属性に対する検証エラー・メッセージを追加することが出来る配列。
- deferreds: Deferred オブジェクトの配列。 deferreds.add(callback) を使って、新しい deferrd な検証を追加することが出来ます。

ハンドラが真偽値 false を返すと、指定された属性の検証は中止されます。 その結果、afterValidateAttribute イベントもトリガされません。

afterValidateAttribute

afterValidateAttribute イベントは、フォーム全体および各属性の検証の後にトリガされます。

イベント・ハンドラのシグニチャは以下の通り:

```
function (event, attribute, messages)
```

引数は以下の通り:

- event: イベントのオブジェクト。
- attribute: 検証される属性。 この引数の構造については attributeDefaults を参照して下さい。
- messages: 指定された属性に対する追加の検証エラー・メッセージを追加することが 出来る配列。

beforeSubmit

beforeSubmit イベントは、全ての検証が通った後、フォームを送信する前にトリガされます。

イベント・ハンドラのシグニチャは以下の通り:

```
function (event)
```

ここで、event は、イベントのオブジェクトです。

ハンドラが真偽値 false を返すと、フォームの送信は中止されます。

ajaxBeforeSend

ajaxBeforeSend イベントは、AJAX ベースの検証のための AJAX リクエストを送信する前にトリガされます。

イベント・ハンドラのシグニチャは以下の通り:

```
function (event, jqXHR, settings)
```

引数は以下の通り:

- `event`: イベントのオブジェクト。
- `jqXHR`: jqXHR のオブジェクト。
- `settings`: AJAX リクエストの設定。

`ajaxComplete`

`ajaxComplete` イベントはAJAX ベースの検証のための AJAX リクエストが完了した後にトリガされます。

イベント・ハンドラのシグニチャは以下の通り:

```
function (event, jqXHR, textStatus)
```

引数は以下の通り:

- `event`: イベントのオブジェクト。
- `jqXHR`: jqXHR のオブジェクト。
- `textStatus`: リクエストの状態 ("success", "notmodified", "error", "timeout", "abort", または "parsererror")。

7.6.2 AJAX でフォームを送信する

検証(バリデーション)は、クライアント・サイドまたは AJAX リクエストによって行うことが出来ますが、 フォームの送信そのものはデフォルトでは通常のリクエストとして実行されます。 フォームを AJAX で送信したい場合は、次のように、フォームの `beforeSubmit` イベントを処理することによって達成することが出来ます。

```
var $form = $('#formId');
$form.on('beforeSubmit', function() {
    var data = $form.serialize();
    $.ajax({
        url: $form.attr('action'),
        type: 'POST',
        data: data,
        success: function (data) {
            // 成功したときの実装
        },
        error: function(jqXHR, errMsg) {
            alert(errMsg);
        }
     });
     return false; // デフォルトの送信を抑止
});
```

jQuery の `ajax()` 関数について更に学習するためには、jQuery documentation[11] を参照して下さい。

[11] https://api.jquery.com/jQuery.ajax/

7.6.3 フィールドを動的に追加する

現在のウェブ・アプリケーションでは、ユーザに対して表示した後でフォームを変更する必要がある場合がよくあります。 例えば、”追加”アイコンをクリックするとフィールドが追加される場合などです。 このようなフィールドに対するクライアント・サイドの検証を有効にするためには、フィールドを ActiveForm JavaScript プラグインに登録しなければなりません。

フィールドそのものを追加して、そして、検証のリストに追加しなければなりません。

```
$('#contact-form').yiiActiveForm('add', {
    id: 'address',
    name: 'address',
    container: '.field-address',
    input: '#address',
    error: '.help-block',
    validate:  function (attribute, value, messages, deferred, $form) {
        yii.validation.required(value, messages, {message: "Validation
    Message Here"});
    }
});
```

フィールドを検証のリストから削除して検証されないようにするためには、次のようにします。

```
$('#contact-form').yiiActiveForm('remove', 'address');
```

Chapter 8

データの表示

8.1 データのフォーマット

ユーザにとってより読みやすい形式でデータを表示するために、formatter アプリケーション・コンポーネント を使ってデータをフォーマットすることが出来ます。デフォルトでは、フォーマッタは yii\i18n\Formatter によって実装されており、 これが、日付/時刻、数字、通貨、その他のよく使われる形式にデータをフォーマットする一連のメソッドを提供します。このフォーマッタは次のようにして使うことが出来ます。

```
$formatter = \Yii::$app->formatter;

// 出力: January 1, 2014
echo $formatter->asDate('2014-01-01', 'long');

// 出力: 12.50%
echo $formatter->asPercent(0.125, 2);

// 出力: <a href="mailto:cebe@example.com">cebe@example.com</a>
echo $formatter->asEmail('cebe@example.com');

// 出力: Yes
echo $formatter->asBoolean(true);
// it also handles display of null values:

// 出力: (not set)
echo $formatter->asDate(null);
```

ご覧のように、これらのメソッドは全て asXyz() という名前を付けられており、Xyz がサポートされている形式を表しています。 別の方法として、汎用メソッド format() を使ってデータをフォーマットすることも出来ます。 この方法を使うと望む形式をプログラム的に制御することが可能になりますので、yii\grid\GridView や yii\widgets\DetailView などのウィジェットでは、こちらがよく使われています。例えば、

```
// 出力: January 1, 2014
```

```
echo Yii::$app->formatter->format('2014-01-01', 'date');

// 配列を使ってフォーマット・メソッドのパラメータを指定することも出来ます。
// `2` は asPercent() メソッドの $decimals パラメータの値です。
// 出力: 12.50%
echo Yii::$app->formatter->format(0.125, ['percent', 2]);
```

> 補足: フォーマッタ・コンポーネントは、エンド・ユーザへの表示用に値をフォーマットすることを目的に設計されています。 ユーザの入力を機械が読み取れる形式にフォーマットしたい場合、また、日付を機械が読み取れる形式にフォーマットしたいだけ、という場合には、 フォーマッタは適切なツールではありません。 日付と数値についてユーザ入力を変換するためには、それぞれ、yii\validators\DateValidator と yii\validators\NumberValidator を使うことが出来ます。機械が読み取れる日付と時刻のフォーマットの単純な相互変換には、PHP の date()[1] 関数で十分です。

8.1.1 フォーマッタを構成する

アプリケーションの構成情報 の中で formatter コンポーネントを構成して、 フォーマットの規則をカスタマイズすることが出来ます。例えば、

```
return [
    'components' => [
        'formatter' => [
            'dateFormat' => 'dd.MM.yyyy',
            'decimalSeparator' => ',',
            'thousandSeparator' => ' ',
            'currencyCode' => 'EUR',
        ],
    ],
];
```

構成可能なプロパティについては、yii\i18n\Formatter を参照してください。

8.1.2 日付と時刻の値をフォーマットする

フォーマッタは日付と時刻に関連した下記の出力形式をサポートしています。

- date - 値は日付としてフォーマットされます。例えば January 01, 2014。
- time - 値は時刻としてフォーマットされます。例えば 14:23。
- datetime - 値は日付および時刻としてフォーマットされます。例えば January 01, 2014 14:23。

[1] https://secure.php.net/manual/ja/function.date.php

- `timestamp` - 値は unix タイムスタンプ[2] としてフォーマットされます。例えば `1412609982`。
- `relativeTime` - 値は、その日時と現在との間隔として、人間に分かりやすい言葉でフォーマットされます。 例えば `1 hour ago`。
- `duration` - 値は継続時間として、人間に分かりやすい言葉でフォーマットされます。例えば `1 day, 2 minutes`。

`date`、`time`、`datetime` メソッドに使われるデフォルトの日時書式は、フォーマッタの `$dateFormat`、 `$timeFormat`、`$datetimeFormat` を構成することで、 グローバルにカスタマイズすることが出来ます。

日付と時刻のフォーマットは、ICU 構文[3] によって指定することが出来ます。 また、ICU 構文と区別するために `php:` という接頭辞を付けて、PHP の date() 構文[4] を使うことも出来ます。例えば、

```
// ICU 形式
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06

// PHP date() 形式
echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

> 情報: PHP 形式の書式の文字の中には ICU でサポートされておらず、従って PHP intl エクステンションでもサポートされていないため、 Yii のフォーマッタで使用できないものがあります。それらの文字のほとんどのもの (`w`, `t`, `L`, `B`, `u`, `I`, `Z`) は、日付の書式としては大して有用ではなく、 むしろ日数の計算をするのに使われるものです。しかし、`S` と `U` は有用かも知れません。これらの動作は次のようにして達成することが出来ます。
>
> - `S` は、月の何日目かを示す英語の序数接尾詞序詞 (すなわち st, nd, rd または th) ですが、これの代りに以下のような代替手段が使用出来ます。
>
>   ```
>   $f = Yii::$app->formatter;
>   $d = $f->asOrdinal($f->asDate('2017-05-15', 'php:j'));
>   echo "On the $d day of the month.";  // "On the 15th day of
>       the month." と表
>   ```
>
>   示
>
> - `U`、すなわち Unix エポックに対しては、`timestamp` 形式を使うことが出来ます。

複数の言語をサポートする必要があるアプリケーションを扱う場合には、ロケールごとに異なる日付と時刻のフォーマットを指定しなければならないことがよくあります。 この仕事を単純化するためには、(`long`、`short` などの) フォーマットのショートカットを代りに使うことが出

[2] http://en.wikipedia.org/wiki/Unix_time
[3] http://userguide.icu-project.org/formatparse/datetime
[4] https://secure.php.net/manual/ja/function.date.php

来ます。 フォーマッタは、現在アクティブな locale に従って、フォーマットのショートカットを適切なフォーマットに変換します。 フォーマットのショートカットとして、次のものがサポートされています (例は en_GB がアクティブなロケールであると仮定したものです)。

- short: 日付は 06/10/2014、時刻は 15:58 を出力
- medium: 6 Oct 2014 と 15:58:42 を出力
- long: 6 October 2014 と 15:58:42 GMT を出力
- full: Monday, 6 October 2014 と 15:58:42 GMT を出力

> 情報: ja_JP ロケールでは、次のようになります。
>
> short: 2014/10/06 と 15:58 medium: 2014/10/06 と 15:58:42 long: 年月日2014106 と 15:58:42 JST full: 年月日月曜日2014106 と 時分秒155842 日本標準時

バージョン 2.0.7 以降では、さまざまな暦法に従って日付をフォーマットすることが可能です。 通常とは異なる暦法を使用する方法については、フォーマッタの $calendar プロパティの API ドキュメントを参照して下さい。

タイム・ゾーン

日時の値をフォーマットするときに、Yii はその値をターゲット・タイム・ゾーン に変換します。 フォーマットされる日付の値は、タイム・ゾーンが明示的に指定されるか、yii\i18n\Formatter::$defaultTimeZone が構成されるかしていない限り、 UTC であると見なされます。

次の例では、ターゲット・タイム・ゾーン が Europe/Berlin に設定されているものとします。

```
// UNIX タイムスタンプを時刻としてフォーマット
echo Yii::$app->formatter->asTime(1412599260); // 14:41:00

// UTC の日付時刻文字列を時刻としてフォーマット
echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00

// CEST の日付時刻文字列を時刻としてフォーマット
echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

> Info: ターゲット・タイム・ゾーン が Asia/Tokyo である場合は、次のようになります。
>
> ```
> echo Yii::$app->formatter->asTime(1412599260); // 21:41:00
> echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); //
>     21:41:00
> echo Yii::$app->formatter->asTime('2014-10-06 21:41:00 JST'); //
>      21:41:00
> ```

フォーマッタ・コンポーネントに対して タイム・ゾーン が明示的に設定されていない場合は、 アプリケーションで設定されたタイム・ゾーン (PHP の構成で設定されたタイム・ゾーンと同じ) が使用されます。

> 補足: タイム・ゾーンは世界中のさまざまな政府によって作られる規則に従うものであり、頻繁に変更されるものであるため、 あなたのシステムにインストールされたタイム・ゾーンのデータベースが最新の情報を持っていない可能性が大いにあります。 タイム・ゾーン・データベースの更新についての詳細は、 ICU マニュアル[5] で参照することが出来ます。 PHP 環境を国際化のために設定する も参照してください。

8.1.3 数値をフォーマットする

フォーマッタは、数値に関連した下記の出力フォーマットをサポートしています。

- `integer` - 値は整数としてフォーマットされます。例えば `42`。
- `decimal` - 値は小数点と三桁ごとの区切りを考慮して十進数としてフォーマットされます。 例えば `2,542.123` または `2.542,123`。
- `percent` - 値は百分率としてフォーマットされます。例えば `42%`。
- `scientific` - 値は科学記法による数値としてフォーマットされます。例えば `4.2E4`。
- `currency` - 値は通貨の値としてフォーマットされます。例えば `£420.00`。 この関数が正しく働くためには、`en_GB` や `en_US` のように、ロケールが国コードを含んでいる必要があります。 なぜなら、この場合は言語だけでは曖昧になるからです。
- `size` - バイト数である値が人間にとって読みやすいサイズとしてフォーマットされます。例えば `410 キビバイト`。
- `shortSize` - `size` の短いバージョンです。例えば `410 KiB`。

数値のフォーマットに使われる書式は、デフォルトではロケールに従って設定される `decimalSeparator` と `thousandSeparator` を使って調整することが出来ます。

更に高度な設定のためには、`yii\i18n\Formatter::$numberFormatterOptions` と `yii\i18n\Formatter::$numberFormatterTextOptions` を使って、内部的に使用される NumberFormatter クラス[6] を構成することが出来ます。例えば、小数部の最大桁数と最小桁数を調整するためには、次のように `yii\i18n\Formatter::$numberFormatterOptions` プロパティを構成します。

```
'numberFormatterOptions' => [
    NumberFormatter::MIN_FRACTION_DIGITS => 0,
    NumberFormatter::MAX_FRACTION_DIGITS => 2,
]
```

[5] http://userguide.icu-project.org/datetime/timezone#TOC-Updating-the-Time-Zone-Data

[6] https://secure.php.net/manual/ja/class.numberformatter.php

8.1.4 その他のフォーマット

日付/時刻と数値のフォーマット以外にも、Yii はよく使われるフォーマットをサポートしています。その中には、次のものが含まれます。

- raw - 値はそのまま出力されます。 null 値が nullDisplay を使ってフォーマットされる以外は、何の効果もない擬似フォーマッタです。
- text - 値は HTML エンコードされます。 これは GridView DataColumn で使われるデフォルトのフォーマットです。
- ntext - 値は HTML エンコードされ、 改行文字が強制改行に変換された平文テキストとしてフォーマットされます。
- paragraphs - 値は HTML エンコードされ、 <p> タグに囲まれた段落としてフォーマットされます。
- html - 値は XSS 攻撃を避けるために HtmlPurifier を使って浄化されます。 ['html', ['Attr.AllowedFrameTargets' => ['_blank']]] のような追加のオプションを渡すことが出来ます。
- email - 値は mailto リンクとしてフォーマットされます。
- image - 値は image タグとしてフォーマットされます。
- url - 値はハイパーリンクとしてフォーマットされます。
- boolean - 値は真偽値としてフォーマットされます。 デフォルトでは、true は Yes、false は No とレンダリングされ、現在のアプリケーションの言語に翻訳されます。 この動作は yii\i18n\Formatter::$booleanFormat プロパティを構成して調整できます。

8.1.5 null 値

Null 値は特殊な方法でフォーマットされます。空文字列を表示する代りに、フォーマッタは null 値を事前定義された文字列 (そのデフォルト値は (not set) です) に変換し、それを現在のアプリケーションの言語に翻訳します。 この文字列は nullDisplay プロパティを構成してカスタマイズすることが出来ます。

8.1.6 データのフォーマットをローカライズする

既に述べたように、フォーマッタは現在のアクティブな locale を使って、 ターゲットの国/地域にふさわしい値のフォーマットを決定することが出来ます。 例えば、同じ日時の値でも、ロケールによって異なる書式にフォーマットされます。

```
Yii::$app->formatter->locale = 'en-US';
echo Yii::$app->formatter->asDate('2014-01-01'); // 出力: January 1, 2014

Yii::$app->formatter->locale = 'de-DE';
echo Yii::$app->formatter->asDate('2014-01-01'); // 出力: 1. Januar 2014

Yii::$app->formatter->locale = 'ru-RU';
echo Yii::$app->formatter->asDate('2014-01-01'); // 出力: 1 января 2014 г.
```

```
Yii::$app->formatter->locale = 'ja-JP';
echo Yii::$app->formatter->asDate('2014-01-01'); // 出力: 2014/01/01
```

デフォルトでは、現在のアクティブな `locale` は `yii\base\Application::$language` の値によって決定されます。 これは `yii\i18n\Formatter::$locale` プロパティを明示的に指定することによってオーバーライドすることが出来ます。

> 補足: Yii のフォーマッタは、PHP intl 拡張[7] に依存してデータのフォーマットの ローカライズをサポートしています。PHP にコンパイルされた ICU ライブラリのバージョンによってフォーマットの結果が異なる場合がありますので、 あなたの全ての環境で、同じ ICU バージョンを使うことが推奨されます。 詳細については、PHP 環境を国際化のために設定する を参照してください。
>
> intl 拡張がインストールされていない場合は、データはローカライズされません。
>
> 1901年より前、または、2038年より後の日時の値は、たとえ intl 拡張がインストールされていても、32-bit システムではローカライズされないことに注意してください。 これは、この場合、ICU ライブラリが日時の値に対して 32-bit の UNIX タイムスタンプを使用しているのが原因です。

8.2 ページネーション

一つのページに表示するにはデータの数が多すぎるという場合に、データを複数のページに分割して、それぞれのページでは一部分だけを表示する、 という戦略がよく使われます。この戦略が ページネーション として知られるものです。

Yii は `yii\data\Pagination` オブジェクトを使って、ページネーションのスキームに関する情報を表します。具体的に言えば、

- `totalCount` データ・アイテムの総数を指定します。 通常、データ・アイテムの総数は、一つのページを表示することが可能なデータ・アイテムの数より、ずっと大きなものになることに注意してください。
- `pageSize` 各ページが含むアイテムの数を指定します。 デフォルト値は 20 です。
- `page` 現在のページ番号 (0 から始まる) を示します。デフォルト値は 0 であり、最初のページを意味します。

[7] https://secure.php.net/manual/ja/book.intl.php

これらの情報を全て定義した yii\data\Pagination オブジェクトを使って、データの一部分を取得して表示することが出来ます。 例えば、データ・プロバイダからデータを取得する場合であれば、ページネーションによって提供される値によって、それに対応する OFFSET と LIMIT の句を DB クエリに指定することが出来ます。下記に例を挙げます。

```
use yii\data\Pagination;

// status = 1 である全ての記事を取得する DB クエリを構築する
$query = Article::find()->where(['status' => 1]);

// 記事の総数を取得する ただし、記事のデータはまだ取得しない()
$count = $query->count();

// 記事の総数を使ってページネーション・オブジェクトを作成する
$pagination = new Pagination(['totalCount' => $count]);

// ページネーションを使ってクエリの OFFSET と LIMIT を修正して記事を取得する
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

上記の例で返される記事のページ番号はどうなるでしょう? それは page という名前のクエリ・パラメータがリクエストに含まれるかどうかによって決ります。 デフォルトでは、ページネーション・オブジェクトは page に page パラメータの値をセットしようと試みます。 そして、このパラメータが提供されていない場合には、デフォルト値である 0 が使用されます。

ページネーションをサポートする UI 要素の構築を容易にするために、Yii はページ・ボタンのリストを表示する yii\widgets\LinkPager ウィジェットを提供しています。 これは、ユーザがページ・ボタンをクリックして、どのページを表示すべきかを指示することが出来るものです。 このウィジェットは、ページネーション・オブジェクトを受け取って、現在のページ番号が何であるかを知り、何個のページ・ボタンを表示すべきかを知ります。 例えば、

```
use yii\widgets\LinkPager;

echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

UI 要素を手動で構築したい場合は、yii\data\Pagination::createUrl() を使って、いろんなページに跳ぶ URL を作成することが出来ます。 このメソッドは page パラメータを要求し、その page パラメータを含む正しくフォーマットされた URL を作成します。 例えば、

```
// 作成される URL が使用すべきルートを指定する
// 指定しない場合は、現在リクエストされているルートが使用される
$pagination->route = 'article/index';
```

```
// /index.php?r=article%2Findex&page=100 を表示
echo $pagination->createUrl(100);

// /index.php?r=article%2Findex&page=101 を表示
echo $pagination->createUrl(101);
```

> ヒント: page クエリ・パラメータの名前をカスタマイズするためには、ページネーション・オブジェクトを作成する際に pageParam プロパティを構成します。

8.3 並べ替え

複数のデータ行を表示する際に、エンド・ユーザによって指定されるカラムに従ってデータを並べ替えなければならないことがよくあります。Yii は yii\data\Sort オブジェクトを使って並べ替えのスキーマに関する情報を表します。 具体的に言えば、

- attributes データの並べ替えに使用できる 属性 を指定します。 単純で良ければ、モデルの属性 をこの属性とすることが出来ます。また、複数のモデル属性や DB のカラムを結合した合成的な属性を指定することも出来ます。詳細については後述します。
- attributeOrders 各属性について、 現在リクエストされている並べ替えの方向を指定します。
- orders 並べ替えの方向をカラムを使う低レベルな形式で示します。

yii\data\Sort を使用するためには、最初にどの属性が並べ替え可能であるかを宣言します。 次に、現在リクエストされている並べ替え情報を attributeOrders または orders から取得して、データのクエリをカスタマイズします。例えば、

```
use yii\data\Sort;

$sort = new Sort([
    'attributes' => [
        'age',
        'name' => [
            'asc' => ['first_name' => SORT_ASC, 'last_name' => SORT_ASC],
            'desc' => ['first_name' => SORT_DESC, 'last_name' => SORT_DESC],
            'default' => SORT_DESC,
            'label' => '氏名',
        ],
    ],
]);

$articles = Article::find()
    ->where(['status' => 1])
    ->orderBy($sort->orders)
    ->all();
```

上記の例では、Sort オブジェクトに対して二つの属性が宣言されています。すなわち、age と name です。

age 属性は Article アクティブ・レコード・クラスの age 属性に対応する 単純な 属性です。 これは、次の宣言と等価です。

```
'age' => [
    'asc' => ['age' => SORT_ASC],
    'desc' => ['age' => SORT_DESC],
    'default' => SORT_ASC,
    'label' => Inflector::camel2words('age'),
]
```

name 属性は Article の first_name と last_name によって定義される 合成的な 属性です。 これは次のような配列構造を使って宣言されています。

- asc および desc の要素は、それぞれ、この属性を昇順および降順に並べ替える方法を指定します。 この値が、データの並べ替えに使用されるべき実際のカラムと方向を表します。 一つまたは複数のカラムを指定して、単純な並べ替えや合成的な並べ替えを示すことが出来ます。
- default 要素は、最初にリクエストされたときの属性の並べ替えに使用されるべき方向を指定します。 デフォルト値は昇順です。 つまり、以前に並べ替えられたことがない状態でこの属性による並べ替えをリクエストすると、この属性の昇順に従ってデータが並べ替えられることになります。
- label 要素は、並べ替えのリンクを作成するために yii\data\Sort::link() を呼んだときに、どういうラベルを使用すべきかを指定するものです。 設定されていない場合は、yii\helpers\Inflector::camel2words() が呼ばれて、属性名からラベルが生成されます。 ラベルは HTML エンコードされないことに注意してください。

  情報: orders の値をデータベースのクエリに直接に供給して、 ORDER BY 句を構築することが出来ます。 データベースのクエリが認識できない合成的な属性が入っている場合があるため、 attributeOrders を使ってはいけません。

yii\data\Sort::link() を呼んでハイパーリンクを生成すれば、それをクリックして、指定した属性によるデータの並べ替えをリクエストすることが出来るようになります。 yii\data\Sort::createUrl() を呼んで並べ替えを実行する URL を生成することも出来ます。 例えば、

```
// 生成される URL が使用すべきルートを指定する
// これを指定しない場合は、現在リクエストされているルートが使用される
$sort->route = 'article/index';

// 氏名による並べ替えと年齢による並べ替えを実行するリンクを表示
echo $sort->link('name') . ' | ' . $sort->link('age');

// /index.php?r=article%2Findex&sort=age を表示
echo $sort->createUrl('age');
```

yii\data\Sort は、リクエストの sort クエリ・パラメータをチェックして、どの属性による並べ替えがリクエストされたかを判断します。 このクエリ・パラメータが存在しない場合のデフォルトの並べ替え方法は yii\data\Sort::$defaultOrder によって指定することが出来ます。 また、sortParam プロパティを構成して、このクエリ・パラメータの名前をカスタマイズすることも出来ます。

8.4 データ・プロバイダ

ページネーション と 並べ替え のセクションにおいて 、エンド・ユーザが特定のページのデータを選んで表示し、いずれかのカラムによってデータを並べ替えることが出来るようにする方法を説明しました。 データのページネーションと並べ替えは非常によくあるタスクですから、Yii はこれをカプセル化した一連の データ・プロバイダ を提供しています。

データ・プロバイダは yii\data\DataProviderInterface を実装するクラスであり、主として、ページ分割され並べ替えられたデータの取得をサポートするものです。 通常は、データ・ウィジェット と共に使用して、 エンド・ユーザが対話的にデータのページネーションと並べ替えをすることが出来るようにします。

Yii のリリースには次のデータ・プロバイダのクラスが含まれています。

- yii\data\ActiveDataProvider: yii\db\Query または yii\db\ActiveQuery を使ってデータベースからデータを取得して、 配列または アクティブ・レコード・インスタンスの形式でデータを返します。
- yii\data\SqlDataProvider: SQL 文を実行して、データベースのデータを配列として返します。
- yii\data\ArrayDataProvider: 大きな配列を受け取り、ページネーションと並べ替えの指定に基づいて、 一部分を切り出して返します。

これら全てのデータ・プロバイダの使用方法は、次の共通のパターンを持っています。

```
// ページネーションと並べ替えのプロパティを構成してデータ・プロバイダを作成する
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// ページ分割されて並べ替えられたデータを取得する
$models = $provider->getModels();

// 現在のページにあるデータ・アイテムの数を取得する
$count = $provider->getCount();

// 全ページ分のデータ・アイテムの総数を取得する
$totalCount = $provider->getTotalCount();
```

データ・プロバイダのページネーションと並べ替えの振る舞いを指定するためには、その pagination と sort のプロパティを構成します。 二つのプロパティは、それぞれ、yii\data\Pagination と yii\data\Sort の構成情報に対応します。 これらを false に設定して、ページネーションや並べ替えの機能を無効にすることも出来ます。

データ・ウィジェット、例えば yii\grid\GridView は、dataProvider という名前のプロパティを持っており、 これにデータ・プロバイダのインスタンスを受け取らせて、それが提供するデータを表示させることが出来ます。例えば、

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

これらのデータ・プロバイダの主たる相異点は、データソースがどのように指定されるかという点にあります。 次に続く項において、各データ・プロバイダの詳細な使用方法を説明します。

8.4.1 アクティブ・データ・プロバイダ

yii\data\ActiveDataProvider を使用するためには、その query プロパティを構成しなければなりません。 これは、yii\db\Query または yii\db\ActiveQuery のオブジェクトを取ることが出来ます。 前者であれば、返されるデータは配列になります。後者であれば、返されるデータは配列または アクティブ・レコード インスタンスとすることが出来ます。例えば、

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// Post オブジェクトの配列を返す
$posts = $provider->getModels();
```

上記の例における $query が次のコードによって作成される場合は、提供されるデータは生の配列になります。

```
use yii\db\Query;
```

```
$query = (new Query())->from('post')->where(['status' => 1]);
```

> 補足: クエリが既に orderBy 句を指定しているものである場合、(sort の構成を通して) エンド・ユーザによって与えられる並べ替えの指定は、 既存の orderBy 句に追加されます。一方、limit と offset の句が存在している場合は、 (pagenation の構成を通して) エンド・ユーザによって指定されるページネーションのリクエストによって上書きされます。

デフォルトでは、yii\data\ActiveDataProvider はデータベース接続として db アプリケーション・コンポーネントを使用します。 yii\data\ActiveDataProvider::$db プロパティを構成すれば、別のデータベース接続を使用することが出来ます。

8.4.2 SQL データ・プロバイダ

yii\data\SqlDataProvider は、生の SQL 文を使用して、必要なデータを取得します。 このデータ・プロバイダは、sort と pagination の指定に基づいて、SQL 文の ORDER BY と OFFSET/LIMIT の句を修正し、 指定された順序に並べ替えられたデータを要求されたページの分だけ取得します。

yii\data\SqlDataProvider を使用するためには、sql プロパティだけでなく、 totalCount プロパティを指定しなければなりません。例えば、

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// データ行の配列を返す
$models = $provider->getModels();
```

> 情報: totalCount プロパティは、データにページネーションを適用しなければならない 場合にだけ要求されます。これは、sql によって指定される SQL 文は、 現在要求されているページのデータだけを返すように、データ・プロバイダによって修正されてしまうからです。 データ・プロバイダは、総ページ数を正しく計算するためには、データ・アイテムの総数を知る必要があります。

8.4.3 配列データ・プロバイダ

yii\data\ArrayDataProvider は、一つの大きな配列を扱う場合に最も適しています。 このデータ・プロバイダによって、一つまたは複数のカラムで並べ替えた配列データの 1 ページ分を返すことが出来ます。 yii\data\ArrayDataProvider を使用するためには、全体の大きな配列として allModels プロパティを指定しなければなりません。 この大きな配列の要素は、連想配列 (例えば DAO のクエリ結果) またはオブジェクト (例えば アクティブ・レコード インスタンス) とすることが出来ます。例えば、

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];

$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
    ],
]);

// 現在リクエストされているページの行を返す
$rows = $provider->getModels();
```

> 補足: アクティブ・データ・プロバイダ および SQL データ・プロバイダ と比較すると、 配列データ・プロバイダは効率の面では劣ります。何故なら、全ての データをメモリにロードしなければならないからです。

8.4.4 データのキーを扱う

データ・プロバイダによって返されたデータ・アイテムを使用する場

合、各データ・アイテムを一意のキーで 特定しなければならないことがよくあります。例えば、データ・アイテムが顧客情報を表す場合、顧客 ID を各顧客データのキーとして使用したいでしょう。データ・プロバイダは、yii\data\DataProviderInterface::getModels() によって返されたデータ・アイテムに対応するそのようなキーのリストを返すことが出来ます。例えば、

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
]);

// Post オブジェクトの配列を返す
$posts = $provider->getModels();

// $post に対応するプライマリ・キーの値を返す
$ids = $provider->getKeys();
```

上記の例では、yii\data\ActiveDataProvider に対して yii\db\ActiveQuery オブジェクトを供給していますから、 キーとしてプライマリ・キーの値を返すのが理にかなっています。キーの値の計算方法を明示的に指定するために、 yii\data\ActiveDataProvider::$key にカラム名を設定したり、キーの値を計算するコーラブルを設定したりすることも出来ます。例えば、

```
// "slug" カラムをキーの値として使用する
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// md5(id) の結果をキーの値として使用する
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.4.5 カスタム・データ・プロバイダを作成する

あなた自身のカスタム・データ・プロバイダ・クラスを作成するためには、yii\data\DataProviderInterface を実装しなければなりません。yii\data\BaseDataProvider を拡張するのが比較的簡単な方法です。そうすれば、データ・プロバイダのコアのロジックに集中することが出来ます。 具体的に言えば、実装する必要があるのは、主として次のメソッドです。

- prepareModels(): 現在のページで利用できるデータ・モデルを準備して、 それを配列として返します。
- prepareKeys(): 現在利用できるデータ・モデルの配列を受け取って、 それと関連付けられるキーの配列を返します。
- prepareTotalCount: データ・プロバイダにある データ・モデルの総数を示す値を返します。

下記は、CSV ファイルを効率的に読み出すデータ・プロバイダのサンプルです。

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string 読み出す CSV ファイルの名前
     */
    public $filename;

    /**
     * @var string|callable キーカラムの名前またはそれを返すコーラブル
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject; // ファイルの特定の行までシークするの
    に SplFileObject が非常に便利


    /**
     * {@inheritdoc}
     */
    public function init()
    {
        parent::init();

        // ファイルを開く
        $this->fileObject = new SplFileObject($this->filename);
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareModels()
    {
        $models = [];
        $pagination = $this->getPagination();

        if ($pagination === false) {
            // ページネーションが無い場合、全ての行を読む
```

```
            while (!$this->fileObject->eof()) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        } else {
            // ページネーションがある場合、一つのページだけを読む
            $pagination->totalCount = $this->getTotalCount();
            $this->fileObject->seek($pagination->getOffset());
            $limit = $pagination->getLimit();

            for ($count = 0; $count < $limit; ++$count) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        }

        return $models;
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareKeys($models)
    {
        if ($this->key !== null) {
            $keys = [];

            foreach ($models as $model) {
                if (is_string($this->key)) {
                    $keys[] = $model[$this->key];
                } else {
                    $keys[] = call_user_func($this->key, $model);
                }
            }

            return $keys;
        } else {
            return array_keys($models);
        }
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareTotalCount()
    {
        $count = 0;

        while (!$this->fileObject->eof()) {
            $this->fileObject->next();
            ++$count;
        }

        return $count;
```

```
    }
}
```

8.4.6 データ・フィルタを使ってデータ・プロバイダをフィルタリングする

データをフィルタリングする や 独立したフィルタ・フォーム で述べられているように、 アクティブ・データ・プロバイダの条件を手作業で構築することも可能ですが、 柔軟なフィルタ条件を必要とする場合には、Yii が持っているデータ・フィルタが非常に役に立ちます。 データ・フィルタは次のようにして使うことが出来ます。

```
$filter = new ActiveDataFilter([
    'searchModel' => 'app\models\PostSearch'
]);

$filterCondition = null;

// どのようなソースからでもフィルタをロードすることが出来ます。
// 例えば、リクエスト・ボディの JSON からロードしたい場合は、
// 下記のように Yii::$app->request->getBodyParams() を使います。
if ($filter->load(\Yii::$app->request->get())) {
    $filterCondition = $filter->build();
    if ($filterCondition === false) {
        // シリアライザがエラーを抽出するだろう
        return $filter;
    }
}

$query = Post::find();
if ($filterCondition !== null) {
    $query->andWhere($filterCondition);
}

return new ActiveDataProvider([
    'query' => $query,
]);
```

PostSearch モデルは、どういうプロパティと値がフィルタリングに使用できるかを定義するという目的のために使用されています。

```
use yii\base\Model;

class PostSearch extends Model
{
    public $id;
    public $title;

    public function rules()
    {
        return [
            ['id', 'integer'],
```

```
            ['title', 'string', 'min' => 2, 'max' => 200],
        ];
    }
}
```

データ・フィルタは極めて柔軟で、どのようにして条件が構築されるか、また、どんな演算子が許容されるかをカスタマイズすることが可能です。 詳細は API リファレンスで `yii\data\DataFilter` を参照して下さい。

8.5 データ・ウィジェット

Yii はデータを表示するために使うことが出来る一連の ウィジェット を提供しています。 DetailView は、単一のレコードのデータを表示するのに使うことが出来ます。 それに対して、ListView と GridView は、複数のデータ・レコードをリストまたはテーブルで表示することが出来るもので、 ページネーション、並べ替え、フィルタリングなどの機能を提供するものです。

8.5.1 DetailView

DetailView は単一のデータ モデル の詳細を表示します。

モデルを標準的な書式で表示する場合 (例えば、全てのモデル属性をそれぞれテーブルの一行として表示する場合) に最も適しています。 モデルは `yii\base\Model` またはそのサブ・クラス、例えば アクティブ・レコード のインスタンスか、連想配列かのどちらかにすることが出来ます。

DetailView は `yii\widgets\DetailView::$attributes` プロパティを使って、モデルのどの属性が表示されるべきか、また、どういうフォーマットで表示されるべきかを決定します。 利用できるフォーマットのオプションについては、フォーマッタのセクション を参照してください。

次に DetailView の典型的な用例を示します。

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',                                                    // title 属性 平
    文テキストで()
        'description:html',                                         // description 属
    性は HTML としてフォーマットされる
        [                                                           // モデルの所有者
    の名前
            'label' => '所有者',
            'value' => $model->owner->name,
            'contentOptions' => ['class' => 'bg-red'],      // 値のタグをカス
    タマイズする HTML 属性
            'captionOptions' => ['tooltip' => 'Tooltip'], // ラベルのタグを
    カスタマイズする HTML 属性
```

```
        ],
        'created_at:datetime',                                    // 作成日時
    は datetime としてフォーマットされる
    ],
]);
```

yii\widgets\GridView が一組のモデルを処理するのとは異なって、DetailView は一つのモデルしか処理しないということを覚えておいてください。 表示すべきモデルはビューの変数としてアクセスできる $model 一つだけですから、たいていの場合、クロージャを使用する必要はありません。

しかし、クロージャが役に立つ場合もあります。例えば、visible が指定されており、それが false と評価される場合には value の計算を避けたい場合です。

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        [
            'attribute' => 'owner',
            'value' => function ($model) {
                return $model->owner->name;
            },
            'visible' => \Yii::$app->user->can('posts.owner.view'),
        ],
    ],
]);
```

8.5.2 ListView

ListView ウィジェットは、データ・プロバイダ からのデータを表示するのに使用されます。 各データ・モデルは指定された ビュー・ファイル を使って表示されます。 ListView は、特に何もしなくても、ページネーション、並べ替え、フィルタリングなどの機能を提供してくれますので、 エンド・ユーザに情報を表示するためにも、データ管理 UI を作成するためにも、非常に便利なウィジェットです。

典型的な使用方法は以下の通りです。

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);
```

_post ビューは次のような内容を含むことが出来ます。

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="post">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>
```

上記のビュー・ファイルでは、現在のデータ・モデルを $model としてアクセスすることが出来ます。追加で次のものを利用することも出来ます。

- $key: mixed - データ・アイテムと関連付けられたキーの値。
- $index: integer - データ・プロバイダによって返されるアイテムの配列の 0 から始まるインデックス。
- $widget: ListView - ウィジェットのインスタンス。

追加のデータを各ビューに渡す必要がある場合は、次のように、$viewParams を使って「キー・値」のペアを渡すことが出来ます。

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

このようにすると、これらをビューで変数として利用できるようになります。

8.5.3 GridView

データ・グリッドすなわち GridView は Yii の最も強力なウィジェットの一つです。これは、システムの管理セクションを素速く作らねばならない時に、 この上なく便利なものです。このウィジェットは データ・プロバイダ からデータを受けて、 テーブルの形式で、行ごとに一組の カラム を使ってデータを表示します。

テーブルの各行が一つのデータ・アイテムを表します。そして、一つのカラムは通常はアイテムの一属性を表します (カラムの中に、複数の属性を組み合わせた複雑な式に対応するものや、静的なテキストを表すものを含めることも出来ます)。

GridView を使うために必要な最小限のコードは次のようなものです。

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;
```

```
$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

上記のコードは、最初にデータ・プロバイダを作成し、次に GridView を使って、データ・プロバイダから受け取る全ての行の全ての属性を表示するものです。 表示されるテーブルには、特に何も設定しなくても、並べ替えとページネーションの機能が装備されます。

グリッドのカラム

グリッドのテーブルのカラムは yii\grid\Column クラスとして表現され、GridView の構成情報の columns プロパティで構成されます。カラムは、タイプや設定の違いに応じて、 データをさまざまな形で表現することが出来ます。デフォルトのクラスは yii\grid\DataColumn です。 これは、モデルの一つの属性を表現し、その属性による並べ替えとフィルタリングを可能にするものです。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // $dataProvider に含まれるデータによって定義される単純なカラム
        // モデルのカラムのデータが使われる
        'id',
        'username',
        // 複雑なカラム定義
        [
            'class' => 'yii\grid\DataColumn', // 省略可。これがデフォルト値。
            'value' => function ($data) {
                return $data->name; // 配列データの場合は $data['name。例えば、']SqlDataProvider を使う場合。
            },
        ],
    ],
]);
```

構成情報の columns の部分が指定されない場合は、Yii は、 データ・プロバイダのモデルの表示可能な全てのカラムを表示しようとすることに注意してください。

カラム・クラス

グリッドのカラムは、いろいろなカラム・クラスを使うことでカスタマ

イズすることが出来ます。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- ここ
            // ここで追加のプロパティを構成することが出来ます
        ],
```

Yii によって提供されるカラム・クラスを以下で見ていきますが、それらに加えて、あなた自身のカラム・クラスを作成することも出来ます。

全てのカラム・クラスは yii\grid\Column を拡張するものですので、グリッドのカラムを構成するときに設定できる 共通のオプションがいくつかあります。

- header によって、ヘッダ行のコンテントを設定することが出来ます。
- footer によって、フッタ行のコンテントを設定することが出来ます。
- visible はカラムの可視性を定義します。
- content によって、行のデータを返す有効な PHP コールバックを渡すことが出来ます。書式は以下の通りです。

```
function ($model, $key, $index, $column) {
    return '文字列';
}
```

下記のオプションに配列を渡して、コンテナ要素のさまざまな HTML オプションを指定することが出来ます。

- headerOptions
- footerOptions
- filterOptions
- contentOptions

データ・カラム　データ・カラム は、データの表示と並べ替えに使用されます。 これがデフォルトのカラムタイプですので、これを使用するときはクラスの指定を省略することが出来ます。

データ・カラムの主要な設定項目は、その format プロパティです。その値が、デフォルトでは Formatter である formatter アプリケーション・コンポーネント のメソッドに対応します。

```
echo GridView::widget([
    'columns' => [
        [
            'attribute' => 'name',
            'format' => 'text'
        ],
        [
            'attribute' => 'birthday',
            'format' => ['date', 'php:Y-m-d']
```

```
        ],
        'created_at:datetime', // shortcut format
        [
            'label' => '教育',
            'attribute' => 'education',
            'filter' => ['0' => '初等教育', '1' => '中等教育', '2' => '高等教
    育'],
            'filterInputOptions' => ['prompt' => '全ての教
    育', 'class' => 'form-control', 'id' => null]
        ],
    ],
]);
```

上記において、text は yii\i18n\Formatter::asText() に対応し、カラムの値が最初の引数として渡されます。 二番目のカラムの定義では、date が yii\i18n\Formatter::asDate() に対応します。 カラムの値が、ここでも、最初の引数として渡され、'php:Y-m-d' が二番目の引数の値として渡されます。

利用できるフォーマッタの一覧については、データのフォーマット のセクションを参照してください。

データカラムを構成するためには、ショートカット形式を使うことも出来ます。 それについては、columns の API ドキュメントで説明されています。

フィルタ・インプットの HTML を制御するためには、filter と filterInputOptions を使用して下さい。

デフォルトでは、カラム・ヘッダは yii\data\Sort::link() によってレンダリングされますが、yii\grid\Column::$header を使って調整することが出来ます。 ヘッダのテキストを変更するには、上の例のように、yii\grid\DataColumn::$label を設定しなければなりません。 デフォルトでは、ラベルはデータ・モデルによって設定されます。詳細は yii\grid\DataColumn::getHeaderCellLabel() を参照して下さい。

アクション・カラム　アクション・カラム は、各行について、更新や削除などのアクション・ボタンを表示します。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\ActionColumn',
            // ここで追加のプロパティを構成することが出来ます
        ],
```

構成が可能なプロパティは、以下の通りです。

- controller は、アクションを処理すべきコントローラの ID です。設定されていない場合は、現在アクティブなコントローラが使われます。
- template は、アクション・カラムの各セルを構成するのに使用されるテンプレートを定義します。 波括弧に囲まれたトークンは、

コントローラのアクション ID として扱われます (アクション・カラムのコンテキストでは ボタンの名前 とも呼ばれます)。 これらは、buttons によって定義される、対応するボタン表示コールバックによって置き換えられます。 例えば、{view} というトークンは、buttons['view'] のコールバックの結果によって置き換えられます。 コールバックが見つからない場合は、トークンは空文字列によって置き換えられます。デフォルトのテンプレートは {view} {update} {delete} です。

- buttons はボタン表示コールバックの配列です。 配列のキーはボタンの名前 (波括弧を除く) であり、値は対応するボタン表示コールバックです。コールバックは下記のシグニチャを使わなければなりません。

```
function ($url, $model, $key) {
    // ボタンの HTML コードを返す
}
```

上記のコードで、$url はカラムがボタンのために生成する URL、$model は現在の行に表示されるモデル・オブジェクト、 そして $key はデータ・プロバイダの配列の中にあるモデルのキーです。

- urlCreator は、指定されたモデルの情報を使って、ボタンの URL を生成するコールバックです。 コールバックのシグニチャは yii\grid\ActionColumn::createUrl() のそれと同じでなければなりません。 このプロパティが設定されていないときは、ボタンの URL は yii\grid\ActionColumn::createUrl() を使って生成されます。
- visibleButtons は、各ボタンの可視性の条件を定義する配列です。 配列のキーはボタンの名前 (波括弧を除く) であり、値は真偽値 true/false または無名関数です。 ボタンの名前がこの配列の中で指定されていない場合は、デフォルトで、ボタンが表示されます。コールバックは次のシグニチャを使わなければなりません。

```
function ($model, $key, $index) {
    return $model->status === 'editable';
}
```

または、真偽値を渡すことも出来ます。

```
[
    'update' => \Yii::$app->user->can('update')
]
```

チェックボックス・カラム　チェックボックス・カラム はチェックボックスのカラムを表示します。

GridView に CheckboxColumn を追加するためには、以下のようにして、columns 構成情報にカラムを追加します。

```
echo GridView::widget([
    'id' => 'grid',
    'dataProvider' => $dataProvider,
```

```
    'columns' => [
        // ...
        [
            'class' => 'yii\grid\CheckboxColumn',
            // ここで追加のプロパティを構成することが出来ます
        ],
    ],
```

ユーザはチェックボックスをクリックして、グリッドの行を選択することが出来ます。 選択された行は、次の JavaScript コードを呼んで取得することが出来ます。

```
var keys = $('#grid').yiiGridView('getSelectedRows');
// keys は選択された行と関連付けられたキーの配列
```

シリアル・カラム　シリアルカラム は、1 から始まる行番号を表示します。

使い方は、次のように、とても簡単です。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'], // <-- ここ
        // ...
```

データを並べ替える

補足: このセクションはまだ執筆中です。

- https://github.com/yiisoft/yii2/issues/1576

データをフィルタリングする

データをフィルタリングするためには、GridView は検索基準を表す モデル を必要とします。 検索基準は、通常は、グリッド・ビューのテーブルのフィルタのフィールドから取得されます。 アクティブ・レコード を使用している場合は、必要な機能を提供する検索用のモデル・クラスを 作成するのが一般的なプラクティスです (あなたに代って Gii が生成してくれます)。 このクラスが、グリッド・ビューのテーブルに表示されるフィルタ・コントロールのための検証規則を定義し、 検索基準に従って修正されたクエリを持つデータ・プロバイダを返す `search()` メソッドを提供します。

`Post` モデルに対して検索機能を追加するために、次の例のようにして、`PostSearch` モデルを作成することが出来ます。

```
<?php

namespace app\models;

use Yii;
```

```
use yii\base\Model;
use yii\data\ActiveDataProvider;

class PostSearch extends Post
{
    public function rules()
    {
        // rules() にあるフィールドだけが検索可能
        return [
            [['id'], 'integer'],
            [['title', 'creation_date'], 'safe'],
        ];
    }

    public function scenarios()
    {
        // 親クラスの scenarios() の実装をバイパスする
        return Model::scenarios();
    }

    public function search($params)
    {
        $query = Post::find();

        $dataProvider = new ActiveDataProvider([
            'query' => $query,
        ]);

        // 検索フォームのデータをロードして検証する
        if (!($this->load($params) && $this->validate())) {
            return $dataProvider;
        }

        // フィルタを追加してクエリを修正する
        $query->andFilterWhere(['id' => $this->id]);
        $query->andFilterWhere(['like', 'title', $this->title])
              ->andFilterWhere(['like', 'creation_date', $this->
    creation_date]);

        return $dataProvider;
    }
}
```

> ヒント: フィルタのクエリを構築する方法を学ぶためには、クエリ・ビルダ、 中でも特に フィルタ条件 を参照してください。

この `search()` メソッドをコントローラで使用して、GridView のためのデータ・プロバイダを取得することが出来ます。

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search(Yii::$app->request->get());
```

```
return $this->render('myview', [
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
]);
```

そしてビューでは、$dataProvider と $searchModel を GridView に与えます。

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        // ...
    ],
]);
```

独立したフィルタ・フォーム

たいていの場合はグリッド・ビューのヘッダのフィルタで十分でしょう。しかし、独立したフィルタのフォームが必要な場合でも、 簡単に追加することができます。まず、以下の内容を持つパーシャル・ビュー _search.php を作成します。

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model app\models\PostSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="post-search">
    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

    <?= $form->field($model, 'title') ?>

    <?= $form->field($model, 'creation_date') ?>

    <div class="form-group">
        <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
        <?= Html::submitButton('Reset', ['class' => 'btn btn-default']) ?>
    </div>

    <?php ActiveForm::end(); ?>
</div>
```

そして、これを以下のように index.php ビューにインクルードします。

```
<?= $this->render('_search', ['model' => $searchModel]) ?>
```

> 補足: Gii を使って CRUD コードを生成する場合、デフォルトで、独立したフィルタ・フォーム (_search.php) が生成されます。 ただし、index.php ビューの中ではコメント・アウトされています。コメントを外せば、すぐに使うことが出来ます。

独立したフィルタ・フォームは、グリッド・ビューに表示されないフィールドによってフィルタをかけたり、 または日付の範囲のような特殊なフィルタ条件を使う必要があったりする場合に便利です。 日付の範囲によってフィルタする場合は、DB には存在しない createdFrom と createdTo という属性を検索用のモデルに追加すると良いでしょう。

```
class PostSearch extends Post
{
    /**
     * @var string
     */
    public $createdFrom;

    /**
     * @var string
     */
    public $createdTo;
}
```

そして、search() メソッドでクエリの条件を次のように拡張します。

```
$query->andFilterWhere(['>=', 'creation_date', $this->createdFrom])
      ->andFilterWhere(['<=', 'creation_date', $this->createdTo]);
```

そして、フィルタ・フォームに、日付の範囲を示すフィールドを追加します。

```
<?= $form->field($model, 'creationFrom') ?>

<?= $form->field($model, 'creationTo') ?>
```

モデルのリレーションを扱う

GridView でアクティブ・レコードを表示するときに、例えば、単に投稿者の id ではなく、 投稿者の名前のような関連するカラムの値を表示するという場合に遭遇するかも知れません。 Post モデルが author という名前のリレーションを持っていて、その投稿者のモデルが name という属性を持っているなら、 yii\grid\GridView::$columns の属性名を author.name と定義します。 そうすれば、GridView が投稿者の名前を表示するようになります。ただし、並べ替えとフィルタリングは、デフォルトでは有効になりません。 これらの機能を追加するためには、前の項で導入した PostSearch モデルを修正しなければなりません。

リレーションのカラムによる並べ替えを有効にするためには、リレーションのテーブルを結合し、 データ・プロバイダの Sort コンポーネントに並べ替えの規則を追加します。

```
$query = Post::find();
$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);

// リレーション 'author' を結合します。これはテーブル 'users' に対するリレーションであり、
// テーブル・エイリアスを 'author' とします。
$query->joinWith(['author' => function($query) { $query->from(['author' => 'users']); }]);
// バージョン 2.0.7 以降では、上の行は $query->joinWith('author AS author'); として単純化することが出来ます。
// リレーションのカラムによる並べ替えを有効にします。
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...
```

フィルタリングも上記と同じ joinWith の呼び出しを必要とします。また、次のように、attributes と rules の中で、検索可能なカラムを追加で定義する必要があります。

```
public function attributes()
{
    // 検索可能な属性にリレーションのフィールドを追加する
    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],
        [['title', 'creation_date', 'author.name'], 'safe'],
    ];
}
```

search() メソッドでは、次のように、もう一つのフィルタ条件を追加するだけです。

```
$query->andFilterWhere(['LIKE', 'author.name', $this->getAttribute('author.name')]);
```

> 情報: 上の例では、リレーション名とテーブル・エイリアスに同じ文字列を使用しています。 しかし、エイリアスとリレーション名が異なる場合は、どこでエイリアスを使い、どこでリレーション名を使うかに注意を払わなければなりません。これに関する簡単な規則は、データベース・クエリを構築す

るために使われる全ての場所でエイリアスを使い、 attributes() や rules() など、その他の全ての定義においてリレーション名を使う、というものです。

例えば、投稿者のリレーションテーブルに au というエイリアスを使う場合は、joinWith の文は以下のようになります。

```
$query->joinWith(['author au']);
```

リレーションの定義においてエイリアスが定義されている場合は、単に $query->joinWith(['author']); として呼び出すことも可能です。

フィルタ条件においてはエイリアスが使われなければなりませんが、属性の名前はリレーション名のままで変りません。

```
$query->andFilterWhere(['LIKE', 'au.name', $this->getAttribute('
    author.name')]);
```

並べ替えの定義についても同じことです。

```
$dataProvider->sort->attributes['author.name'] = [
     'asc' => ['au.name' => SORT_ASC],
     'desc' => ['au.name' => SORT_DESC],
];
```

さらに、並べ替えの defaultOrder を指定するときも、 エイリアスではなくリレーション名を使う必要があります。

```
$dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
```

情報: joinWith およびバックグラウンドで実行されるクエリの詳細については、 アクティブ・レコード - リレーションを使ってテーブルを結合する を参照してください。

SQL ビューを使って、データのフィルタリング・並べ替え・表示をする

もう一つ別に、もっと高速で便利な手法があります。SQL ビューです。例えば、ユーザとユーザのプロファイルを一緒にグリッド・ビューに表示する必要がある場合、次のような SQL ビューを作成することが出来ます。

```
CREATE OR REPLACE VIEW vw_user_info AS
    SELECT user.*, user_profile.lastname, user_profile.firstname
    FROM user, user_profile
    WHERE user.id = user_profile.user_id
```

そして、このビューを表す ActiveRecord を作成します。

```
namespace app\models\views\grid;

use yii\db\ActiveRecord;
```

```
class UserView extends ActiveRecord
{

    /**
     * {@inheritdoc}
     */
    public static function tableName()
    {
        return 'vw_user_info';
    }

    public static function primaryKey()
    {
        return ['id'];
    }

    /**
     * {@inheritdoc}
     */
    public function rules()
    {
        return [
            // ここで規則を定義
        ];
    }

    /**
     * {@inheritdoc}
     */
    public function attributeLabels()
    {
        return [
            // ここで属性のラベルを定義
        ];
    }


}
```

このようにした後は、この UserView アクティブ・レコードを検索用のモデルとともに使うことが出来ます。並べ替えやフィルタリングの属性を追加で定義する必要はありません。 全ての属性がそのままで動作します。この手法にはいくつかの長所と短所があることに注意してください。

- 並べ替えとフィルタリングの条件をいろいろと定義する必要はありません。全てそのままで動きます。
- データサイズが小さく、実行される SQL クエリの数が少ない (通常なら全てのリレーションについて一つずつ必要になる追加のクエリが要らない) ため、非常に高速になり得ます。
- これは SQL ビューにかぶせた単純な UI に過ぎないもので、エンティティに含まれるドメイン・ロジックを欠いています。 従っ

て、isActive や isDeleted などのような UI に影響するメソッドがある場合は、それらをこのクラスの中に複製する必要があります。

一つのページに複数のグリッド・ビュー

一つのページで二つ以上のグリッド・ビューを使うことが出来ますが、お互いが干渉しないように、追加の構成がいくつか必要になります。 グリッド・ビューの複数のインスタンスを使う場合は、並べ替えとページネーションのリンクが違うパラメータ名を持って生成されるように構成して、 それぞれのグリッド・ビューが独立した並べ替えとページネーションを持つことが出来るようにしなければなりません。 そのためには、データ・プロバイダの sort と pagination インスタンスの sortParam と pageParam を設定します。

Post と User のリストを表示するために、二つのプロバイダ、$userProvider と $postProvider を準備済みであると仮定します。

```
use yii\grid\GridView;

$userProvider->pagination->pageParam = 'user-page';
$userProvider->sort->sortParam = 'user-sort';

$postProvider->pagination->pageParam = 'post-page';
$postProvider->sort->sortParam = 'post-sort';
```

```
echo '<h1ユーザ></h1>';
echo GridView::widget([
    'dataProvider' => $userProvider,
]);
```

```
echo '<h1投稿></h1>';
echo GridView::widget([
    'dataProvider' => $postProvider,
]);
```

GridView を Pjax とともに使う

Pjax ウィジェットを使うと、ページ全体をリロードせずに、 ページの一部分だけを更新することが出来ます。 これを使うと、フィルタを使うときに、GridView の中身だけを更新することが出来ます。

```
use yii\widgets\Pjax;
use yii\grid\GridView;

Pjax::begin([
    // PJax のオプション
]);
    Gridview::widget([
        // GridView のオプション
    ]);
Pjax::end();
```

Pjax は、Pjax ウィジェットの内側にあるリンク、および、Pjax::$linkSelector で指定されているリンクに対しても動作します。 しかし、これは ActionColumn のリンクに対しては問題となり得ます。 この問題を防止するためには、ActionColumn::$buttons プロパティを編集して data-pjax="0" という HTML 属性を追加します。

Gii における Pjax を伴う GridView/ListView バージョン 2.0.5 以降、Gii の CRUD ジェネレータでは $enablePjax というオプションがウェブ・インタフェイスまたはコマンドラインで使用可能になっています。

```
yii gii/crud --controllerClass="backend\\controllers\PostController" \
  --modelClass="common\\models\\Post" \
  --enablePjax=1
```

これによって、GridView または ListView を囲む Pjax ウィジェットが生成されます。

8.5.4 さらに読むべき文書

- Arno Slatius による Rendering Data in Yii 2 with GridView and ListView[8]。

8.6 クライアント・スクリプトを扱う

今日のウェブ・アプリケーションでは、静的な HTML ページがレンダリングされてブラウザに送信されるだけでなく、 JavaScript によって、既存の要素を操作したり、新しいコンテントを AJAX でロードしたりして、 ブラウザに表示されるページを修正します。 このセクションでは、JavaScript と CSS をウェブ・サイトに追加したり、 それらを動的に調整するために Yii によって提供されているメソッドを説明します。

8.6.1 スクリプトを登録する

yii\web\View オブジェクトを扱う際には、フロントエンド・スクリプトを動的に登録することが出来ます。 このための専用のメソッドが二つあります。

- インライン・スクリプトのための registerJs()
- 外部スクリプトのための registerJsFile()

インライン・スクリプトを登録する

インライン・スクリプトは、設定のためのコード、動的に生成されるコード、および、ウィジェット に含まれる再利用可能なフロントエンド・コードが生成する コード断片などです。インライン・スクリプトを

[8] http://www.sitepoint.com/rendering-data-in-yii-2-with-gridview-and-listview/

追加するためのメソッド registerJs() は、次のようにして使うことが出来ます。

```
$this->registerJs(
    "$('#myButton').on('click', function() { alertボタンがクリックされました(''); });",
    View::POS_READY,
    'my-button-handler'
);
```

最初の引数は、ページに挿入したい実際の JS コードです。これが <script> タグに包まれて挿入されます。 二番目の引数は、スクリプトがページのどの位置に挿入されるべきかを決定します。 取りうる値は以下のとおりです。

- View::POS_HEAD - head セクション。
- View::POS_BEGIN - 開始の <body> の直後。
- View::POS_END - 終了の </body> の直前。
- View::POS_READY - ドキュメントの ready イベント[9] でコードを実行するための指定。 これを指定すると、jQuery が自動的に登録され、コードは適切な jQuery コードの中に包まれます。これがデフォルトの位置指定です。
- View::POS_LOAD - ドキュメントの load イベント[10] でコードを実行するための指定。 上記と同じく、これを指定すると、jQuery が自動的に登録されます。

最後の引数は、スクリプトのコード・ブロックを一意に特定するために使われるスクリプトのユニークな ID です。同じ ID のスクリプトが既にある場合は、新しいものを追加するのでなく、 それを置き換えます。ID を指定しない場合は、JS コードそれ自身が ID として扱われます。この ID によって、同じコードが複数回登録されるのを防止します。

スクリプト・ファイルを登録する

registerJsFile() の引数は、 registerCssFile() の引数と同様なものです。 以下に示す例では、main.js ファイルを、yii\web\JqueryAsset への依存関係とともに、登録します。 これは、main.js ファイルは jquery.js の後に追加される、ということを意味します。 このような依存関係の仕様が無ければ、main.js と jquery.js の間の相対的な順序は未定義となり、コードは動作しなくなるでしょう。

外部スクリプトは次のようにして追加することが出来ます。

```
$this->registerJsFile(
    '@web/js/main.js',
    ['depends' => [\yii\web\JqueryAsset::className()]]
);
```

[9] http://learn.jquery.com/using-jquery-core/document-ready/
[10] http://learn.jquery.com/using-jquery-core/document-ready/

これによって、アプリケーションの base URL の下に配置されている /js/main.js スクリプトを読み込むタグが追加されます。

ただし、外部 JS ファイルを登録するのには、 registerJsFile() を使わずに、アセット・バンドル を使うことが強く推奨されます。 なぜなら、そうする方が、柔軟性も高く、依存関係の構成も粒度を細かく出来るからです。また、アセット・バンドルを使えば、複数の JS ファイルを結合して圧縮すること (アクセスの多いウェブ・サイトではそうすることが望まれます) が可能になります。

8.6.2 CSS を登録する

Javascript と同様に、registerCss() または registerCssFile() を使って CSS を登録することが出来ます。 前者は CSS のコードブロックを登録し、後者は外部 CSS ファイルを登録するものです。

インライン CSS を登録する

```
$this->registerCss("body { background: #f00; }");
```

上記のコードによって、結果として、下記の出力がページの <head> セクションに追加されます。

```
<style>
body { background: #f00; }
</style>
```

style タグに追加の属性を指定したい場合は、名前-値 の配列を二番目の引数として渡します。最後の引数は、スタイルのブロックを一意に特定するために使われるユニークな ID です。 同じスタイルがコードの別の箇所で重複して登録されたとしても、このスタイルのブロックが一度だけ追加されることを保証するものです。

CSS ファイルを登録する

CSS ファイルは次のようにして登録することが出来ます。

```
$this->registerCssFile("@web/css/themes/black-and-white.css", [
    'depends' => [\yii\bootstrap\BootstrapAsset::className()],
    'media' => 'print',
], 'css-print-theme');
```

上記のコードは /css/themes/black-and-white.css という CSS ファイルに対するリンクをページの <head> セクションに追加します。

- 最初の引数が、登録される CSS ファイルを指定します。 この例における @web in this example is an アプリケーションのベース URL に対するエイリアス です。
- 二番目の引数は、結果として出力される <link> タグの HTML 属性を指定するものです。 ただし、depends というオプションは特別な処理を受けます。これは、この CSS ファイルが依存するアセット・バンドルを指定するものです。 この例の場合は、yii\bootstrap

\BootstrapAsset が依存するアセット・バンドルです。 これは、この CSS ファイルが yii\bootstrap\BootstrapAsset に属する CSS ファイルの後に追加されることを意味します。
- 最後の引数はこの CSS ファイルを特定する ID を指定するものです。 省略された場合は、CSS ファイルの URL が代りに ID として使用されます。

外部 CSS ファイルを登録するのには、 registerCssFile() を使わずに、アセット・バンドル を使うことが強く推奨されます。アセット・バンドルを使えば、複数の CSS ファイルを結合して圧縮すること (アクセスの多いウェブ・サイトではそうすることが望まれます) が可能になります。また、アプリケーションの全てのアセットの依存関係を一ヶ所で構成することが出来るため、より大きな柔軟性を得ることが出来ます。

8.6.3 アセット・バンドルを登録する

既に述べたように、CSS ファイルと JavaScript ファイルを直接に登録する代りにアセット・バンドルを使うことが推奨されます。 アセット・バンドルを定義する方法の詳細は、ガイドの アセット のセクションで知ることが出来ます。 既に定義されているアセット・バンドルの使い方は、次のように非常に単純明快です。

```
\frontend\assets\AppAsset::register($this);
```

上記のコードでは、ビュー・ファイルのコンテキストにおいて、AppAsset バンドルが ($this で表される) 現在のビューに対して登録されています。 ウィジェットの中からアセット・バンドルを登録するときは、ウィジェットの $view を代りに渡します ($this->view)。

8.6.4 動的な Javascript を生成する

ビュー・ファイルでは、HTML コードが直接に書き出されのではなく、ビューの変数に依存して、PHP のコードによって生成されることがよくあります。 生成された HTML を Javascript によって操作するためには、JS コードも同様に動的な部分を含まなければなりません。 例えば、jQuery セレクタの ID などがそうです。

PHP の変数を JS コードに挿入するためには、変数の値を適切にエスケープする必要があります。 JS コードを専用の JS ファイルの中に置くのではなく、HTML に挿入する場合は特にそうです。 Yii は、この目的のために、Json ヘルパの htmlEncode() メソッドを提供しています。 その使用方法は、以下の例の中で示されています。

グローバルな JavaScript の構成情報を登録する

この例では、配列を使って、グローバルな構成情報のパラメータをアプリケーションの PHP のパートから JS のフロントエンド・コードに渡します。

```
$options = [
    'appName' => Yii::$app->name,
    'baseUrl' => Yii::$app->request->baseUrl,
    'language' => Yii::$app->language,
    // ...
];
$this->registerJs(
    "var yiiOptions = ".\yii\helpers\Json::htmlEncode($options).";",
    View::POS_HEAD,
    'yiiOptions'
);
```

上記のコードは、次のような JavaScript の変数定義を含む `<script>` タグを登録します。 例えば、

```
var yiiOptions = {"appName":"My Yii Application","baseUrl":"/basic/web","
    language":"en"};
```

このようにすれば、あなたの Javascript コードで、これらの構成情報に `yiiOptions.baseUrl` や `yiiOptions.language` のようにしてアクセスすることが出来るようになります。.

翻訳されたメッセージを渡す

あなたの JavaScript が何らかのイベントに反応してメッセージを表示する必要がある、という状況に遭遇するかも知れません。 複数の言語で動作するアプリケーションでは、この文字列は、現在のアプリケーシの言語に翻訳されなければなりません。 これを達成する一つの方法は、Yii によって提供されている [メッセージ翻訳機能] (tutorial-i18n.md#message-translation) を使って、その結果を JavaScript コードに渡すことです。

```
$message = \yii\helpers\Json::htmlEncode(
    \Yii::t('app', 'Button clicked!')
);
$this->registerJs(<<<JS
    $('#myButton').on('click', function() { alert( $message ); });
JS
);
```

上記のサンプル・コードは、可読性を高めるために、PHP の ヒアドキュメント構文[11] を使っています。 また、ヒアドキュメントは、たいていの IDE で、より良い構文ハイライトが可能になるので、 インライン JavaScript、特に一行に収まらないものを書くときに推奨される方法です。 変数 `$message` は PHP で生成され、`Json::htmlEncode` のおかげで、適切な JS 構文の文字列を含むものになります。 それを JavaScript コードに挿入して、`alert()` の関数呼び出しに動的な文字列を渡すことが出来ます。

[11] https://secure.php.net/manual/ja/language.types.string.php#language.types.string.syntax.heredoc

補足: ヒアドキュメントを使う場合は、JS コード中の変数名に注意してください。 `$` で始まる変数は、PHP の変数として解釈され、 その値によって置き換えられる可能性があります。 ただし、`$(` または `$.` という形式の jQuery 関数は PHP 変数として解釈される心配は無く、安全に使うことが出来ます。

8.6.5 `yii.js` スクリプト

補足: このセクションはまだ書かれていません。このセクションは、`yii.js` によって提供される以下の機能についての説明を含むはずのものです。

- Yii JavaScript モジュール
- CSRF パラメータの処理
- `data-confirm` ハンドラ
- `data-method` ハンドラ
- スクリプトのフィルタリング
- リダイレクトの処理

8.7 テーマ

テーマは、元のビュー・レンダリングのコードに触れる必要なしに、ビューのセットを別のセットに置き換えるための方法です。 テーマを使うとアプリケーションのルック・アンド・フィールを体系的に変更することが出来ます。

テーマを使うためには、`view` アプリケーション・コンポーネントの `theme` プロパティを構成しなければなりません。 このプロパティが、ビュー・ファイルが置換される方法を管理する `yii\base\Theme` オブジェクトを構成します。 指定しなければならない `yii\base\Theme` のプロパティは主として以下のものです。

- `yii\base\Theme::$basePath`: テーマのリソース (CSS、JS、画像など) を含むベース・ディレクトリを指定します。
- `yii\base\Theme::$baseUrl`: テーマのリソースのベース URL を指定します。
- `yii\base\Theme::$pathMap`: ビュー・ファイルの置換の規則を指定します。 詳細は後述する項で説明します。

例えば、`SiteController` で `$this->render('about')` を呼び出すと、ビュー・ファイル `@app/views/site/about.php` をレンダリングすることになります。しかし、下記のようにアプリケーション構成情報でテーマを有効にすると、 代りに、ビュー・ファイル `@app/themes/basic/site/about.php` がレンダリングされます。

```
return [
```

```
    'components' => [
        'view' => [
            'theme' => [
                'basePath' => '@app/themes/basic',
                'baseUrl' => '@web/themes/basic',
                'pathMap' => [
                    '@app/views' => '@app/themes/basic',
                ],
            ],
        ],
    ],
];
```

> 情報: テーマではパス・エイリアスがサポートされています。ビューの置換を行う際に、パス・エイリアスは実際のファイル・パスまたは URL に変換されます。

yii\base\View::$theme プロパティを通じて yii\base\Theme オブジェクトにアクセスすることが出来ます。 例えば、ビュー・ファイルの中では $this がビュー・オブジェクトを指すので、次のようなコードを書くことが出来ます。

```
$theme = $this->theme;

// $theme->baseUrl . '/img/logo.gif' を返す
$url = $theme->getUrl('img/logo.gif');

// $theme->basePath . '/img/logo.gif' を返す
$file = $theme->getPath('img/logo.gif');
```

yii\base\Theme::$pathMap プロパティが、ビュー・ファイルがどのように置換されるべきかを制御します。 このプロパティは「キー・値」ペアの配列を取ります。 キーは置き換えられる元のビューのパスであり、値は対応するテーマのビューのパスです。 置換は部分一致に基づいて行われます。 あるビューのパスが pathMap 配列のキーのどれかで始っていると、その一致している部分が対応する配列の値によって置き換えられます。 上記の構成例を使う場合、@app/views/site/about.php は @app/views というキーに部分一致するため、@app/themes/basic/site/about.php に置き換えられることになります。

モジュールにテーマを適用する

モジュールにテーマを適用するためには、yii\base\Theme::$pathMap を次のように構成します。

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/modules' => '@app/themes/basic/modules', // <-- !!!
],
```

これによって、@app/modules/blog/views/comment/index.php に @app/themes/basic/modules/blog/views/comment/index.php というテーマを適用することが出来ます。

ウィジェットにテーマを適用する

ウィジェットにテーマを適用するためには、yii\base\Theme::$pathMap を次のように構成します。

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/widgets' => '@app/themes/basic/widgets', // <-- !!!
],
```

これによって、@app/widgets/currency/views/index.php に @app/themes/basic/widgets/currency/views/index.php というテーマを適用することが出来ます。

8.7.1 テーマの継承

場合によっては、基本的なルック・アンド・フィールを含むアプリケーションの基本テーマを定義しておいて、 現在の祝日に基づいてルック・アンド・フィールを少し変更したい、ということがあるかもしれません。 テーマの継承を使ってこの目的を達することが出来ます。テーマの継承は、一つのビュー・パスを複数のターゲットに割り付けることによって設定することが出来ます。例えば、

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
]
```

この場合、ビュー @app/views/site/index.php には、どちらのテーマ・ファイルが存在するかに従って、 @app/themes/christmas/site/index.php か @app/themes/basic/site/index.php か、どちらかのテーマが適用されます。 テーマ・ファイルが両方とも存在する場合は、最初のものが優先されます。実際の場面では、ほとんどのテーマ・ビュー・ファイルを @app/themes/basic に保管し、その中のいくつかを @app/themes/christmas でカスタマイズすることになるでしょう。

Chapter 9

セキュリティ

9.1 セキュリティ

十分なセキュリティは、すべてのアプリケーションの健全さと成功のために欠くことが出来ないものです。 不幸なことに、理解が不足しているためか、実装の難易度が高すぎるためか、セキュリティのことになると手を抜く開発者がたくさんいます。 Yii によって駆動されるあなたのアプリケーションを可能な限り安全にするために、Yii はいくつかの優秀な使いやすいセキュリティ機能を内蔵しています。

- 認証
- 権限付与
- パスワードを扱う
- 暗号化
- ビューのセキュリティ
- 認証クライアント[1]
- ベスト・プラクティス
- 信頼できるプロキシとヘッダ

9.2 認証

認証は、ユーザが誰であるかを確認するプロセスです。 通常は、識別子 (ユーザ名やメール・アドレスなど) と秘密のトークン (パスワードやアクセス・トークンなど) を使って、ユーザがそうであると主張する通りのユーザであるか否かを判断します。 認証がログイン機能の基礎となります。

Yii はさまざまなコンポーネントを結び付けてログインをサポートする認証フレームワークを提供しています。 このフレームワークを使用するために、あなたは主として次の仕事をする必要があります。

- `user` アプリケーション・コンポーネントを構成する。

[1] https://github.com/yiisoft/yii2-authclient/blob/master/docs/guide-ja/README.md

- yii\web\IdentityInterface インタフェイスを実装するクラスを作成する。

9.2.1 yii\web\User を構成する

user アプリケーション・コンポーネントがユーザの認証状態を管理します。 実際の認証ロジックを含む ユーザ識別情報クラス は、あなたが指定しなければなりません。 下記のアプリケーション構成情報においては、user の ユーザ識別情報クラス は app\models\User であると構成されています。 app\models\User の実装については、次の項で説明します。

```
return [
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
        ],
    ],
];
```

9.2.2 yii\web\IdentityInterface を実装する

ユーザ識別情報クラス が実装しなければならない yii\web\IdentityInterface は次のメソッドを含んでいます。

- findIdentity(): 指定されたユーザ ID を使ってユーザ識別情報クラスのインスタンスを探します。 セッションを通じてログイン状態を保持する必要がある場合に、このメソッドが使用されます。
- findIdentityByAccessToken(): 指定されたアクセス・トークンを使ってユーザ識別情報クラスのインスタンスを探します。 単一の秘密のトークンでユーザを認証する必要がある場合 (ステートレスな RESTful アプリケーションなどの場合) に、このメソッドが使用されます。
- getId(): ユーザ識別情報クラスのインスタンスによって表されるユーザの ID を返します。
- getAuthKey(): クッキー・ベースのログインを検証するのに使用されるキーを返します。 このキーがログイン・クッキーに保存され、後でサーバ・サイドのキーと比較されて、 ログイン・クッキーが有効であることが確認されます。
- validateAuthKey(): クッキー・ベースのログイン・キーを検証するロジックを実装します。

特定のメソッドが必要でない場合は、中身を空にして実装しても構いません。例えば、あなたのアプリケーションが純粋なステート・レス RESTful アプリケーションであるなら、 実装する必要があるのは findIdentityByAccessToken() と getId() だけであり、他のメソッドは全て中身を空にしておくことが出来ます。

次の例では、ユーザ識別情報クラス は、user データベース・テーブルと関連付けられた アクティブ・レコード クラスとして実装されていま

す。

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * 与えられた ID によってユーザ識別情報を探す
     *
     * @param string|int $id 探すための ID
     * @return IdentityInterface|null 与えられた ID に合致する Identity オブ
    ジェクト
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * 与えられたトークンによってユーザ識別情報を探す
     *
     * @param string $token 探すためのトークン
     * @return IdentityInterface|null 与えられたトークンに合致す
    る Identity オブジェクト
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string 現在のユーザの ID
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string 現在のユーザの認証キー
     */
    public function getAuthKey()
    {
        return $this->auth_key;
    }
```

```
    /**
     * @param string $authKey
     * @return bool 認証キーが現在のユーザに対して有効か否か
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}
```

前述のように、getAuthKey() と validateAuthKey() は、 あなたのアプリケーションがクッキー・ベースのログイン機能を使用する場合にのみ実装する必要があります。 この場合、次のコードを使って、各ユーザに対して認証キーを生成して、user テーブルに保存しておくことが出来ます。

```
class User extends ActiveRecord implements IdentityInterface
{
    ......

    public function beforeSave($insert)
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
                $this->auth_key = \Yii::$app->security->generateRandomString
    ();
            }
            return true;
        }
        return false;
    }
}
```

> 補足: ユーザ識別情報クラスである User と yii\web\User を混同してはいけません。 前者は認証のロジックを実装するクラスであり、普通は、ユーザの認証情報を保存する何らかの持続的ストレージと関連付けられた アクティブ・レコード クラスとして実装されます。 後者はユーザの認証状態の管理に責任を持つアプリケーション・コンポーネントです。

9.2.3 yii\web\User を使う

yii\web\User は、主として、user アプリケーション・コンポーネントの形で使います。

現在のユーザの識別情報は、Yii::$app->user->identity という式を使って取得することが出来ます。 これは、現在ログインしているユーザのユーザ識別情報クラス のインスタンスを返すか、現在のユーザが認証されていない (つまりゲストである) 場合は null を返します。 次のコードは、yii\web\User からその他の認証関連の情報を取得する方法を示すものです。

```
// 現在のユーザの識別情報。ユーザが認証されていない場合は null
$identity = Yii::$app->user->identity;

// 現在のユーザの 。ユーザが認証されていない場合はID null
$id = Yii::$app->user->id;

// 現在のユーザがゲストである 認証されていない() かどうか
$isGuest = Yii::$app->user->isGuest;
```

ユーザをログインさせるためには、次のコードを使うことが出来ます。

```
// 指定された username を持つユーザ識別情報を探す
// 必要ならパスワードをチェックしてもよいことに注意
$identity = User::findOne(['username' => $username]);

// ユーザをログインさせる
Yii::$app->user->login($identity);
```

yii\web\User::login() メソッドは現在のユーザの識別情報を yii\web\User にセットします。 セッションが 有効 にされている場合は、 ユーザの認証状態がセッション全体を通じて保持されるように、ユーザ識別情報がセッションに保管されます。 クッキー・ベースのログイン (つまり “remember me”、「次回は自動ログイン」) が 有効 にされている場合は、 ユーザ識別情報をクッキーにも保存して、クッキーが有効である限りは、ユーザの認証状態をクッキーから復元することが可能になります。

クッキー・ベースのログインを有効にするためには、アプリケーションの構成情報で yii\web\User::$enableAutoLogin を true に構成する必要があります。 また、yii\web\User::login() メソッドを呼ぶときには、有効期間のパラメータを与える必要があります。

ユーザをログアウトさせるためには、単に次のように logout() を呼びます。

```
Yii::$app->user->logout();
```

ユーザのログアウトはセッションが有効にされている場合にだけ意味があることに注意してください。 logout() メソッドは、ユーザ認証状態をメモリとセッションの両方から消去します。そして、デフォルトでは、ユーザのセッションデータの 全て を破壊します。 セッション・データを保持したい場合は、代りに、Yii::$app->user->logout(false) を呼ばなければなりません。

9.2.4 認証のイベント

yii\web\User クラスは、ログインとログアウトのプロセスで、いくつかのイベントを発生させます。

- EVENT_BEFORE_LOGIN: yii\web\User::login() の開始時に発生します。 イベント・ハンドラがイベントの isValid プロパティを false にセットした場合は、 ログインのプロセスがキャンセルされます。

- EVENT_AFTER_LOGIN: ログインが成功した時に発生します。
- EVENT_BEFORE_LOGOUT: yii\web\User::logout() の開始時に発生します。 イベント・ハンドラがイベントの isValid プロパティを false にセットした場合は、 ログアウトのプロセスがキャンセルされます。
- EVENT_AFTER_LOGOUT: ログアウトが成功した時に発生します。

これらのイベントに反応して、ログイン監査、オンライン・ユーザ統計などの機能を実装することが出来ます。 例えば、EVENT_AFTER_LOGIN のハンドラの中で、 user テーブルにログインの日時と IP アドレスを記録することが出来ます。

9.3 権限付与

権限付与は、ユーザが何かをするのに十分な許可を有しているか否かを確認するプロセスです。 Yii は二つの権限付与の方法を提供しています。すなわち、アクセス制御フィルタ (ACF) と、ロール・ベース・アクセス制御 (RBAC) です。

9.3.1 アクセス制御フィルタ (ACF)

アクセス制御フィルタ (ACF) は、yii\filters\AccessControl として実装される単純な権限付与の方法であり、 何らかの単純なアクセス制御だけを必要とするアプリケーションで使うのに最も適したものです。 その名前が示すように、ACF は、コントローラまたはモジュールで使用することが出来るアクション フィルタ です。 ACF は、ユーザがアクションの実行をリクエストしたときに、一連の アクセス規則 をチェックして、現在のユーザがそのアクションにアクセスする許可を持つかどうかを決定します。

下記のコードは、site コントローラで ACF を使う方法を示すものです。

```
use yii\web\Controller;
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
```

```
                ],
                [
                    'allow' => true,
                    'actions' => ['logout'],
                    'roles' => ['@'],
                ],
            ],
        ],
    ];
}
// ...
}
```

上記のコードにおいて、ACF は site コントローラにビヘイビアとしてアタッチされています。これがアクション・フィルタを使用する典型的な方法です。 only オプションは、ACF が login、logout、signup のアクションにのみ適用されるべきであることを指定しています。 site コントローラの他の全てのアクションには ACF の影響は及びません。 rules オプションは アクセス規則 を指定するものであり、以下のように読むことが出来ます。

- 全てのゲスト・ユーザ (まだ認証されていないユーザ) に、login と singup のアクションにアクセスすることを許可します。 roles オプションに疑問符 ? が含まれていますが、これは「ゲスト」を表す特殊なトークンです。
- 認証されたユーザに、logout アクションにアクセスすることを許可します。 @ という文字はもう一つの特殊なトークンで、「認証されたユーザ」を表すものです。

ACF による権限付与のプロセスにおいては、現在の実行コンテキストに合致する規則が見つかるまで、 アクセス規則が上から下へと一つずつ調べられます。 そして、合致したアクセス規則の allow の値が、ユーザが権限を有するか否かを決定するのに使われます。 合致する規則が一つもなかった場合は、ユーザが権限をもたないことを意味し、ACF はアクションの継続を中止します。

ユーザが現在のアクションにアクセスする権限を持っていないと判定した場合は、デフォルトでは、ACF は以下の手段を取ります。

- ユーザがゲストである場合は、yii\web\User::loginRequired() を呼び出して、ユーザのブラウザをログイン・ページにリダイレクトします。
- ユーザが既に認証されている場合は、yii\web\ForbiddenHttpException を投げます。

この動作は、次のように、yii\filters\AccessControl::$denyCallback プロパティを構成することによって、カスタマイズすることが出来ます。

```
[
    'class' => AccessControl::className(),
    ...
    'denyCallback' => function ($rule, $action) {
```

```
            throw new \Exception('このページにアクセスする権限がありません。');
        }
    ]
```

アクセス規則 は多くのオプションをサポートしています。以下はサポートされているオプションの要約です。 yii\filters\AccessRule を拡張して、あなた自身のカスタマイズしたアクセス規則のクラスを作ることも出来ます。

- allow: これが「許可」の規則であるか、「禁止」の規則であるかを指定します。
- actions: どのアクションにこの規則が適用されるかを指定します。これはアクション ID の配列でなければなりません。比較は大文字と小文字を区別します。 このオプションが空であるか指定されていない場合は、規則が全てのアクションに適用されることを意味します。
- controllers: どのコントローラにこの規則が適用されるかを指定します。これはコントローラ ID の配列でなければなりません。 コントローラがモジュールに属する場合は、モジュール ID をコントローラ ID の前に付けます。比較は大文字と小文字を区別します。このオプションが空であるか指定されていない場合は、規則が全てのコントローラに適用されることを意味します。
- roles: どのユーザ・ロールにこの規則が適用されるかを指定します。 二つの特別なロールが認識されます。これらは、yii\web\User::$isGuest によって判断されます。
  - ?: ゲスト・ユーザ (まだ認証されていないユーザ) を意味します。
  - @: 認証されたユーザを意味します。

  その他のロール名を使うと、yii\web\User::can() の呼び出しが惹起されますが、そのためには、RBAC (次のセクションで説明します) を有効にする必要があります。 このオプションが空であるか指定されていない場合は、規則が全てのロールに適用されることを意味します。
- roleParams: yii\web\User::can() に渡されるパラメータを指定します。 パラメータがどのように使われるかは、RBAC 規則を説明する後のセクションを参照して下さい。このオプションが空であるか設定されていない場合は、パラメータは渡されません。
- ips: どの クライアントの IP アドレス にこの規則が適用されるかを指定します。 IP アドレスは、最後にワイルドカード * を含むことが出来て、同じプレフィクスを持つ IP アドレスに合致させることが出来ます。 例えば、'192.168.*' は、'192.168.' のセグメントに属する全ての IP アドレスに合致します。 このオプションが空であるか指定されていない場合は、規則が全ての IP アドレスに適用されることを意味します。
- verbs: どのリクエスト・メソッド (HTTP 動詞、例えば GET や POST) にこの規則が適用されるかを指定します。 比較は大文字と小文字を

区別しません。

- matchCallback: この規則が適用されるべきか否かを決定するために呼び出されるべき PHP コーラブルを指定します。
- denyCallback: この規則がアクセスを禁止する場合に呼び出されるべき PHP コーラブルを指定します。

下記は、matchCallback オプションを利用する方法を示す例です。 このオプションによって、任意のアクセス制御ロジックを書くことが可能になります。

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['special-callback'],
                'rules' => [
                    [
                        'actions' => ['special-callback'],
                        'allow' => true,
                        'matchCallback' => function ($rule, $action) {
                            return date('d-m') === '31-10';
                        }
                    ],
                ],
            ],
        ];
    }

    // matchCallback が呼ばれる。このページは毎年月日だけアクセス出来ま
    す。1031
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

9.3.2 ロール・ベース・アクセス制御 (RBAC)

ロール・ベース・アクセス制御 (RBAC) は、単純でありながら強力な集中型のアクセス制御を提供します。 RBAC と他のもっと伝統的なアクセス制御スキーマとの比較に関する詳細については、 Wiki 記事[2] を参照してください。

Yii は、NIST RBAC モデル[3] に従って、一般的階層型 RBAC を実装

[2] http://ja.wikipedia.org/wiki/%E3%83%AD%E3%83%BC%E3%83%AB%E3%83%99%E3%83%BC%E3%82%B9%E3%82%A2%E3%82%AF%E3%82%BB%E3%82%B9%E5%88%B6%E5%BE%A1

[3] http://csrc.nist.gov/rbac/sandhu-ferraiolo-kuhn-00.pdf

しています。 RBAC の機能は、authManager アプリケーション・コンポーネント を通じて提供されます。

RBAC を使用することには、二つの作業が含まれます。 最初の作業は、RBAC 権限付与データを作り上げることであり、第二の作業は、権限付与データを使って必要とされる場所でアクセス・チェックを実行することです。

説明を容易にするために、まず、いくつかの基本的な RBAC の概念を導入します。

基本的な概念

ロール (役割) は、許可 (例えば、記事を作成する、記事を更新するなど) のコレクションです。 一つのロールを一人または複数のユーザに割り当てることが出来ます。 ユーザが特定の許可を有しているか否かをチェックするためには、その許可を含むロールがユーザに割り当てられているか否かをチェックすればよいのです。

各ロールまたは許可に関連付けられた 規則 が存在し得ます。 規則とは、アクセス・チェックの際に、対応するロールや許可が現在のユーザに適用されるか否かを決定するために実行されるコード断片のことです。 例えば、「記事更新」の許可は、現在のユーザが記事の作成者であるかどうかをチェックする規則を持つことが出来ます。 そして、アクセス・チェックのときに、ユーザが記事の作成者でない場合は、彼/彼女は「記事更新」の許可を持っていないと見なすことが出来ます。

ロールおよび許可は、ともに、階層的に構成することが出来ます。具体的に言えば、一つのロールは他のロールと許可を含むことが出来、 許可は他の許可を含むことが出来ます。Yii は、一般的な 半順序 階層を実装していますが、これはその特殊形として 木 階層を含むものです。 ロールは許可を含むことが出来ますが、許可はロールを含むことが出来ません。

RBAC を構成する

権限付与データを定義してアクセス・チェックを実行する前に、 authManager アプリケーション・コンポーネントを構成する必要があります。 Yii は二種類の権限付与マネージャを提供しています。すなわち、yii\rbac\PhpManager と yii\rbac\DbManager です。 前者は権限付与データを保存するのに PHP スクリプト・ファイルを使いますが、後者は権限付与データをデータベースに保存します。 あなたのアプリケーションが非常に動的なロールと許可の管理を必要とするのでなければ、前者を使うことを考慮するのが良いでしょう。

PhpManager を使用する

次のコードは、アプリケーションの構成情報で yii\rbac\PhpManager クラスを使って authManager を構成する方法を示すものです。

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

これで authManager は \Yii::$app->authManager によってアクセスすることが出来るようになります。

デフォルトでは、yii\rbac\PhpManager は RBAC データを @app/rbac/ ディレクトリの下のファイルに保存します。 権限の階層をオンラインで変更する必要がある場合は、必ず、ウェブ・サーバのプロセスがこのディレクトリとその中の全てのファイルに対する書き込み権限を有するようにしてください。

DbManager を使用する　次のコードは、アプリケーションの構成情報で yii\rbac\DbManager クラスを使って authManager を構成する方法を示すものです。

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
            // RBAC アイテムの階層をキャッシュしたい場合はコメントを外す
            // 'cache' => 'cache',
        ],
        // ...
    ],
];
```

> 補足: yii2-basic-app テンプレートを使おうとする場合は、config/web.php に加えて、 config/console.php 構成ファイルにおいても authManager を宣言する必要があります。 yii2-advanced-app の場合は、authManager は common/config/main.php で一度だけ宣言されなければなりません。

DbManager は四つのデータベース・テーブルを使ってデータを保存します。

- itemTable: 権限アイテムを保存するためのテーブル。デフォルトは “auth_item”。
- itemChildTable: 権限アイテムの階層を保存するためのテーブル。デフォルトは “auth_item_child”。
- assignmentTable: 権限アイテムの割り当てを保存するためのテーブル。デフォルトは “auth_assignment”。

- `ruleTable`: 規則を保存するためのテーブル。デフォルトは“auth_rule”。

先に進む前にこれらのテーブルをデータベースに作成する必要があります。そのためには、`@yii/rbac/migrations` に保存されているマイグレーションを使うことが出来ます。

```
yii migrate --migrationPath=@yii/rbac/migrations
```

異なる名前空間のマイグレーションを扱う方法の詳細については 分離されたマイグレーション のセクションを参照して下さい。

これで `authManager` は `\Yii::$app->authManager` によってアクセスすることが出来るようになります。

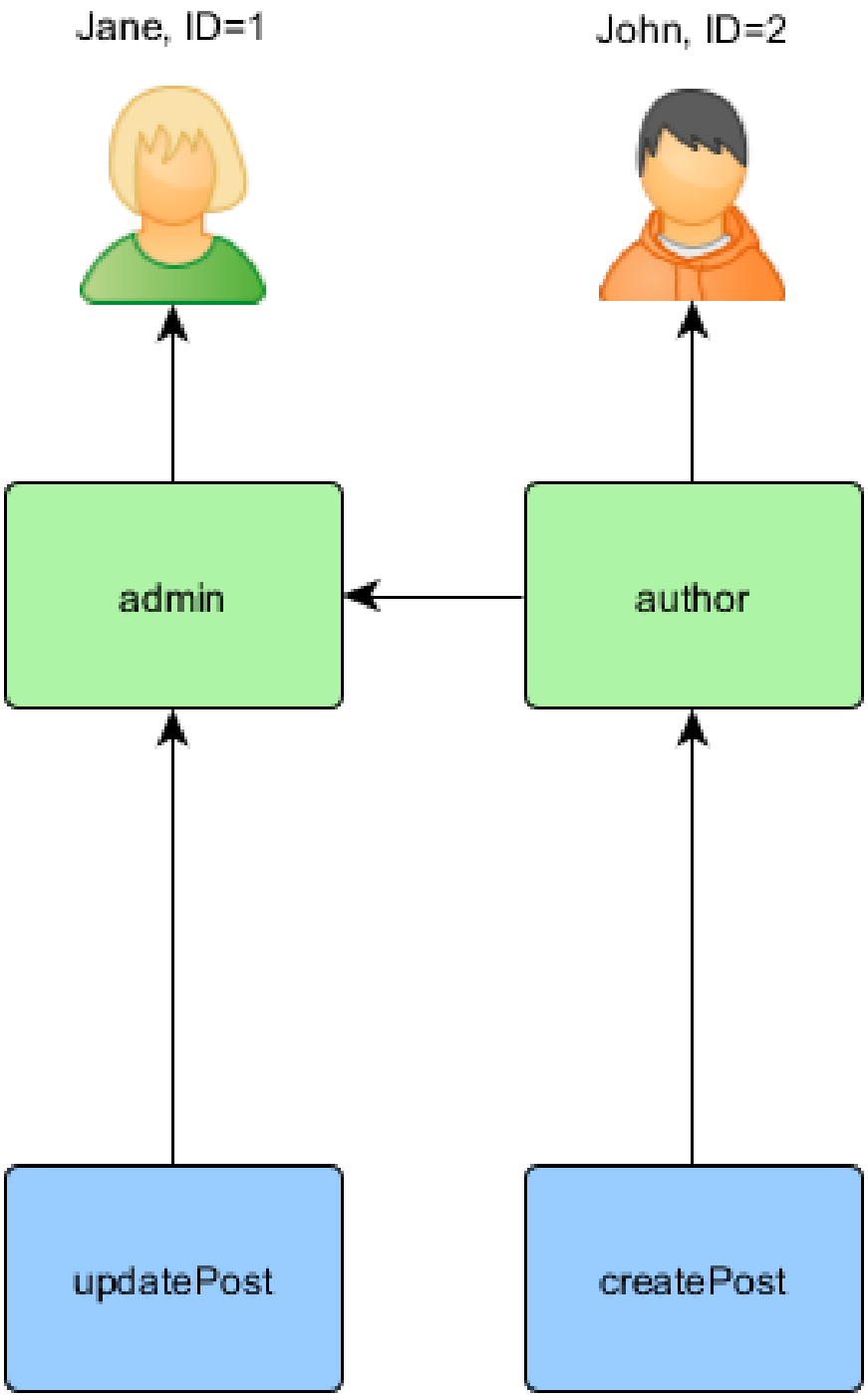
権限付与データを構築する

権限付与データを構築する作業は、つまるところ、以下のタスクに他なりません。

- ロールと許可を定義する
- ロールと許可の関係を定義する
- 規則を定義する
- 規則をロールと許可に結び付ける
- ロールをユーザに割り当てる

権限付与に要求される柔軟性の程度によって、上記のタスクのやりかたも異なってきます。 許可の階層構造が開発者によってのみ変更されることを意図する場合は、 マイグレーションまたはコンソールコマンドを使うことが出来ます。 マイグレーションを使う場合の利点は、他のマイグレーションと一緒に実行できることです。 コンソール・コマンドを使う場合の利点は、階層構造の全体が、複数のマイグレーションに分散することなく、コード中に見やすい形で保たれることです。

どちらの方法でも、結局は次のような RBAC 階層を得ることになります。



許可の階層構造が動的に形成される必要がある場合は、UI またはコンソール・コマンドが必要になります。 階層構造そのものを構築するた

めに使用される API には違いはありません。

マイグレーションを使う　マイグレーション を使って、 authManager が提供する API によって階層を初期化したり変更したりすることが出来ます。

./yii migrate/create init_rbac を使って新しいマイグレーションを作成し、階層の作成を実装します。

```
<?php
use yii\db\Migration;

class m170124_084304_init_rbac extends Migration
{
    public function up()
    {
        $auth = Yii::$app->authManager;

        // "createPost" という許可を追加する
        $createPost = $auth->createPermission('createPost');
        $createPost->description = '記事を投稿';
        $auth->add($createPost);

        // "updatePost" という許可を追加する
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = '記事を更新';
        $auth->add($updatePost);

        // "author" ロールを追加し、このロールに "createPost" の許可を付与する
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // "admin" ロールを追加し、このロールに "updatePost" の許可を付与する
        // 同時に、"author" ロールが持つ許可も付与する
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // ロールをユーザに割り当て
    る。1 と 2 は IdentityInterface::getId() によって返される ID
        // IdentityInterface::getId() は、通常は User モデルの中で実装される
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }

    public function down()
    {
        $auth = Yii::$app->authManager;

        $auth->removeAll();
    }
```

```
}
```

> どのユーザにどのロールを割り当てるかをハードコードしたくない場合は、マイグレーションに `->assign()` の呼び出しを書かないで下さい。 その代りに、ロールの割り当てを管理する UI またはコンソール・コマンドを作成して下さい。

マイグレーションは `yii migrate` を使って適用することが出来ます。

コンソール・コマンドを使う

許可の階層が全く変化せず、決った数のユーザしか存在しない場合は、`authManager` が提供する API によって権限付与データを一回だけ初期設定する コンソール・コマンド を作ることが出来ます。

```
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;
        $auth->removeAll();

        // "createPost" という許可を追加する
        $createPost = $auth->createPermission('createPost');
        $createPost->description = '記事を投稿';
        $auth->add($createPost);

        // "updatePost" という許可を追加
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = '記事を更新';
        $auth->add($updatePost);

        // "author" というロールを追加し、このロールに "createPost" の許可を付
    与する
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // "admin" というロールを追加し、このロールに "updatePost" 許可を付与
    する
        // 同時に、"author" ロールが持つ許可も付与する
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);
```

```
        // ロールをユーザに割り当て
    る。1 と 2 は IdentityInterface::getId() によって返される ID
        // IdentityInterface::getId() は、通常は User モデルの中で実装される
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }
}
```

> 補足: アドバンスト・テンプレートを使おうとするときは、RbacController を console/controllers ディレクトリの中に置いて、名前空間を console\controllers に変更する必要があります。

上記のコマンドは、コンソールから次のようにして実行することが出来ます。

```
yii rbac/init
```

> どのユーザにどのロールを割り当てるかをハードコードしたくない場合は、 コマンドに ->assign() の呼び出しを書かないで下さい。 その代りに、ロールの割り当てを管理する UI またはコンソール・コマンドを作成して下さい。

9.3.3 ロールをユーザに割り当てる

投稿者 (author) は記事を投稿することが出来、管理者 (admin) は記事を更新することに加えて投稿者が出来る全てのことが出来ます。

あなたのアプリケーションがユーザ自身によるユーザ登録を許している場合は、新しく登録されたユーザに一度はロールを割り当てる必要があります。 例えば、アドバンスト・プロジェクト・テンプレートにおいては、登録したユーザの全てを「投稿者」にするために、 frontend\models\SignupForm::signup() を次のように修正しなければなりません。

```
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save(false);

        // 次の三行が追加されたものです
        $auth = Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());
```

```
        return $user;
    }

    return null;
}
```

動的に更新される権限付与データを持つ複雑なアクセス制御を必要とするアプリケーションについては、 authManager が提供する API を使って、特別なユーザ・インタフェイス (つまり、管理パネル) を開発する必要があるでしょう。

規則を使う

既に述べたように、規則がロールと許可に制約を追加します。規則は yii\rbac\Rule を拡張したクラスであり、 execute() メソッドを実装しなければなりません。前に作った権限階層においては、投稿者は自分自身の記事を編集することが出来ませんでした。 これを修正しましょう。最初に、ユーザが記事の投稿者であることを確認する規則が必要です。

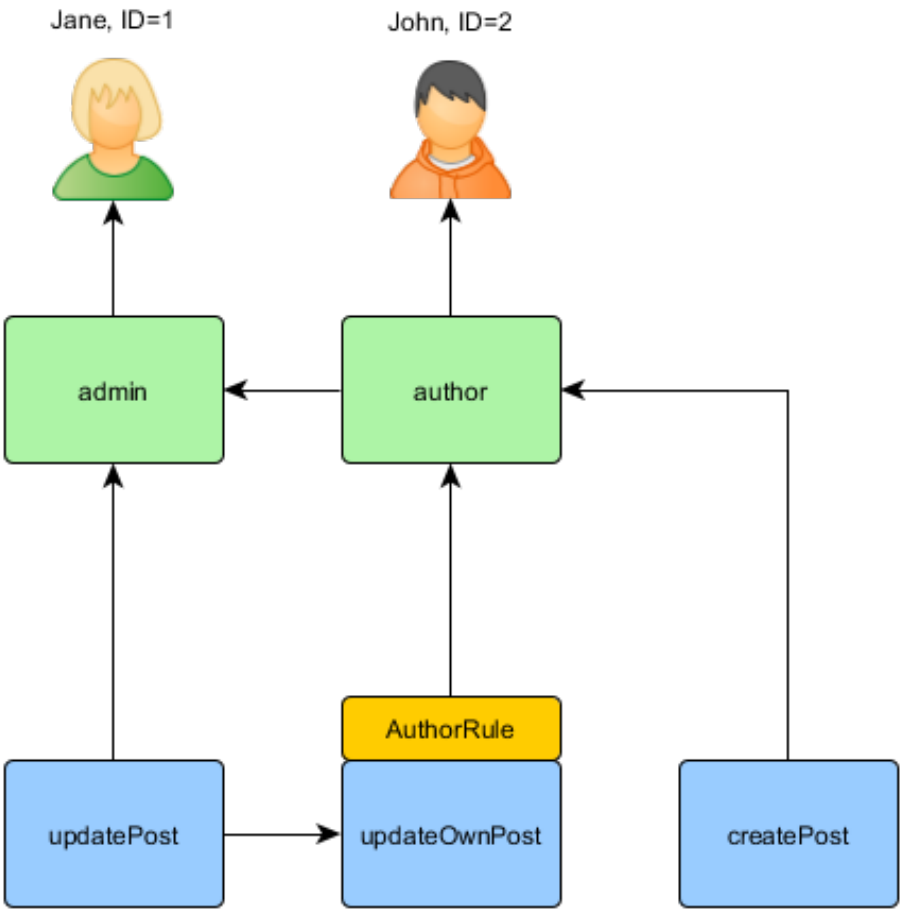
```
namespace app\rbac;

use yii\rbac\Rule;
use app\models\Post;

/**
 * authorID がパラメータで渡されたユーザと一致するかチェックする
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';

    /**
     * @param string|int $user ユーザ ID
     * @param Item $item この規則が関連付けられているロールまたは許可
     * @param array $params ManagerInterface::checkAccess() に渡されたパラ
    メータ
     * @return bool 関連付けられたロールまたは許可を認めるか否かを示す値
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user
    : false;
    }
}
```

上の規則は、post が $user によって作成されたかどうかをチェックします。 次に、前に使ったコマンドの中で、updateOwnPost という特別な許可を作成します。

```
$auth = Yii::$app->authManager;

// 規則を追加する
```

```
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// "updateOwnPost" という許可を作成し、それに規則を関連付ける
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = '自分の記事を更新';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" は "updatePost" から使われる
$auth->addChild($updateOwnPost, $updatePost);

// "author" に自分の記事を更新することを許可する
$auth->addChild($author, $updateOwnPost);
```

これで、次のような権限階層になります。



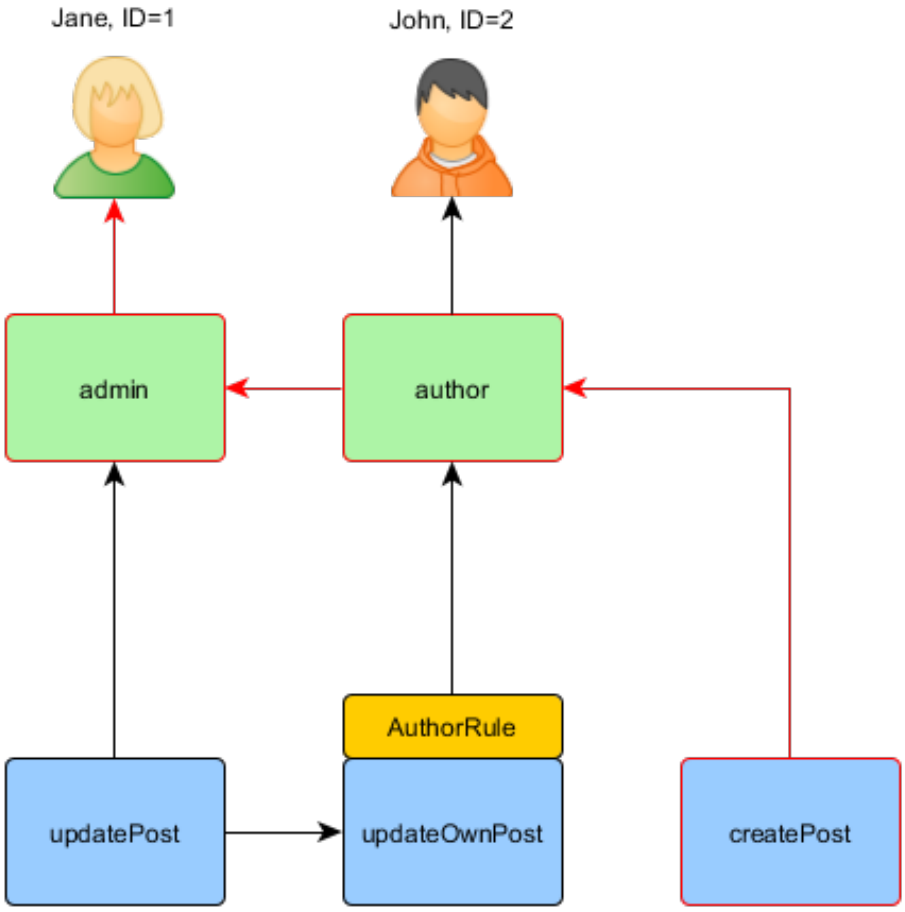
アクセス・チェック

権限付与データが準備できてしまえば、アクセス・チェックは `yii\rbac`

\ManagerInterface::checkAccess() メソッドを呼ぶだけの簡単な仕事です。 たいていのアクセス・チェックは現在のユーザに関するものですから、Yii は、便利なように、yii\web\User::can() というショートカット・メソッドを提供しています。 これは、次のようにして使うことが出来ます。

```
if (\Yii::$app->user->can('createPost')) {
    // 記事を作成する
}
```

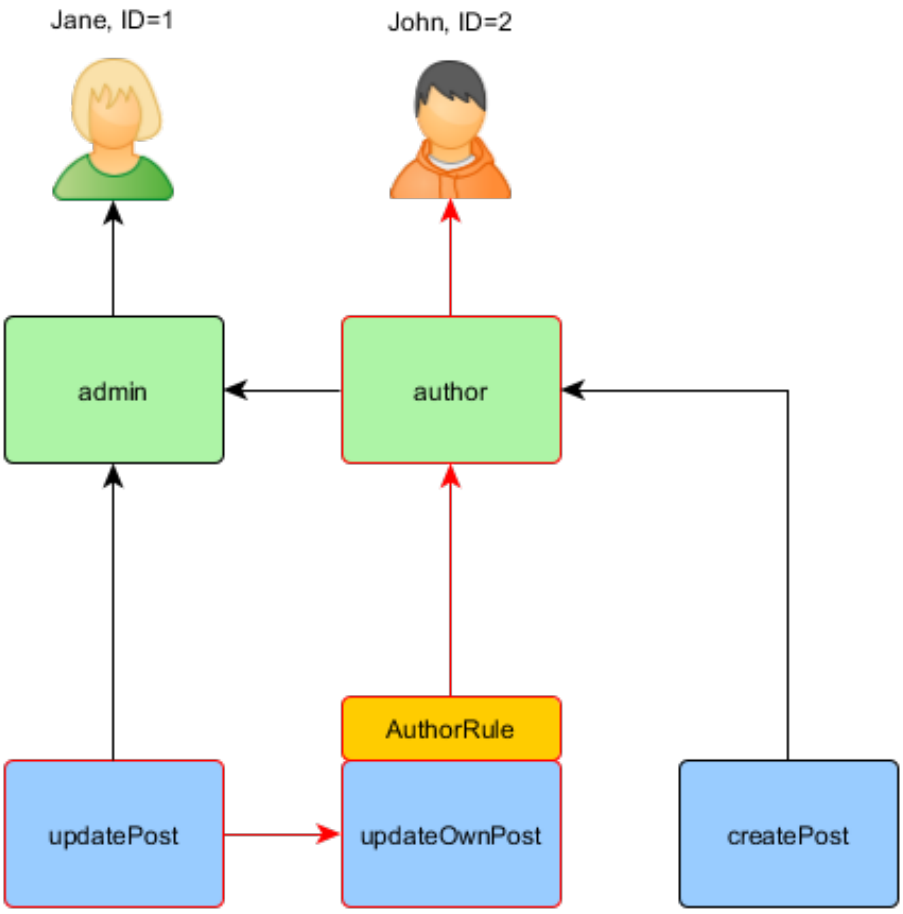
現在のユーザが ID=1 である Jane であるとすると、createPost からスタートして Jane まで到達しようと試みます。



ユーザが記事を更新することが出来るかどうかをチェックするためには、前に説明した AuthorRule によって要求される追加のパラメータを渡す必要があります。

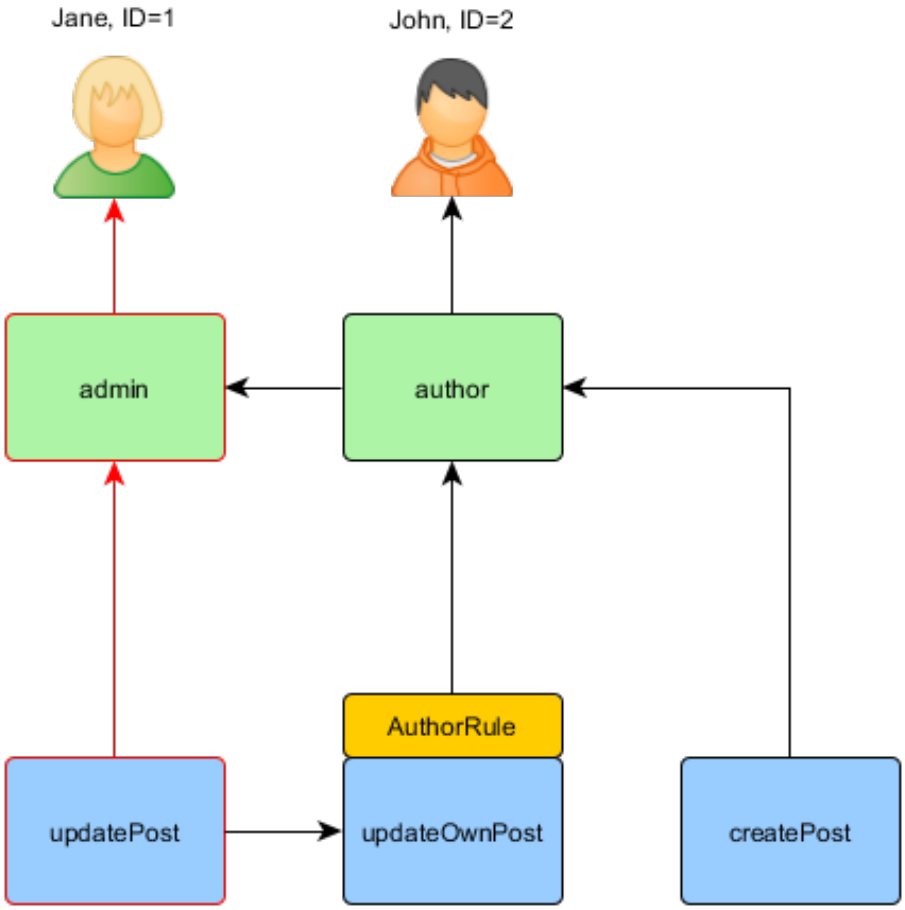
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // 記事を更新する
}
```

現在のユーザが John であるとすると、次の経路をたどります。



`updatePost` からスタートして、`updateOwnPost` を通過します。通過するためには、`AuthorRule` が `execute` メソッドで `true` を返さなければなりません。 `execute` メソッドは `can` メソッドの呼び出しから `$params` を受け取りますので、その値は `['post' => $post]` です。 すべて OK であれば、John に割り当てられている `author` に到達します。

Jane の場合は、彼女が管理者であるため、少し簡単になります。



コントローラ内で権限付与を実装するのには、いくつかの方法があります。 追加と削除に対するアクセス権を分離する細分化された許可が必要な場合は、それぞれのアクションに対してアクセス権をチェックする必要があります。 各アクション・メソッドの中で上記の条件を使用するか、または `yii\filters\AccessControl` を使います。

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['index'],
                    'roles' => ['managePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['view'],
```

```
                    'roles' => ['viewPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['create'],
                    'roles' => ['createPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['update'],
                    'roles' => ['updatePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['delete'],
                    'roles' => ['deletePost'],
                ],
            ],
        ],
    ];
}
```

全ての CRUD 操作がまとめて管理される場合は、managePost のような単一の許可を使い、 yii\web\Controller::beforeAction() の中でそれをチェックするのが良いアイデアです。

上記の例では、アクションにアクセスするために必要と指定されたロールについて、パラメータは渡されていません。 しかし、updatePost 許可の場合は、それが正しく動作するためには post パラメータを渡す必要があります。 アクセス規則の中で roleParams を指定することによって、 yii\web\User::can() にパラメータを渡すことが出来ます。

```
[
    'allow' => true,
    'actions' => ['update'],
    'roles' => ['updatePost'],
    'roleParams' => function() {
        return ['post' => Post::findOne(['id' => Yii::$app->request->get('id
    ')])];
    },
],
```

上記の例では、roleParams はアクセス規則がチェックされるときに評価されるクロージャになっています。 従って、モデルは必要になったときだけロードされます。 ロール・パラメータの作成が簡単な操作である場合は、次のように、単に配列を指定しても構いません。

```
[
    'allow' => true,
    'actions' => ['update'],
    'roles' => ['updatePost'],
    'roleParams' => ['postId' => Yii::$app->request->get('id')],
],
```

デフォルト・ロールを使う

デフォルト・ロールというのは、全て のユーザに 黙示的 に割り当てられるロールです。 `yii\rbac\ManagerInterface::assign()` を呼び出す必要はなく、権限付与データはその割り当て情報を含みません。

デフォルト・ロールは、通常、そのロールが当該ユーザに適用されるかどうかを決定する規則と関連付けられます。

デフォルト・ロールは、たいていは、何らかのロールの割り当てを既に持っているアプリケーションにおいて使われます。 例えば、アプリケーションによっては、ユーザのテーブルに “group” というカラムを持って、個々のユーザが属する特権グループを表している場合があります。 それぞれの特権グループを RBAC ロールに対応付けることが出来るのであれば、デフォルト・ロールの機能を使って、それぞれのユーザに RBAC ロールを自動的に割り当てることが出来ます。 どのようにすればこれが出来るのか、例を使って説明しましょう。

ユーザのテーブルに `group` というカラムがあって、1 は管理者グループ、2 は投稿者グループを示していると仮定しましょう。 これら二つのグループの権限を表すために、それぞれ、`admin` と `author` という RBAC ロールを作ることにします。 このとき、次のように RBAC データをセットアップすることが出来ます。

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * ユーザのグループが合致するかどうかをチェックする
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;
            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }
        return false;
    }
}
```

次に、前のセクション で説明したように、あなた独自のコマンド/マイグレーションを作成します。

```
$auth = Yii::$app->authManager;
```

```
$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... $author の子として許可を追加 ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... $admin の子として許可を追加 ...
```

上記において、"author" が "admin" の子として追加されているため、規則クラスの `execute()` メソッドを実装する時には、 この階層関係にも配慮しなければならないことに注意してください。 このために、ロール名が "author" である場合には、`execute()` メソッドは、ユーザのグループが 1 または 2 である (ユーザが "admin" グループまたは "author" グループに属している) ときに true を返しています。

次に、`authManager` の構成情報で、この二つのロールを `yii\rbac\BaseManager::$defaultRoles` としてリストします。

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

このようにすると、アクセス・チェックを実行すると、`admin` と `author` の両方のロールは、それらと関連付けられた規則を評価することによってチェックされるようになります。 規則が true を返せば、そのロールが現在のユーザに適用されることになります。 上述の規則の実装に基づいて言えば、ユーザの `group` の値が 1 であれば `admin` ロールがユーザに適用され、 `group` の値が 2 であれば `author` ロールが適用されるということを意味します。

9.4 パスワードを扱う

ほとんどの開発者はパスワードを平文テキストで保存してはいけないということを知っていますが、パスワードを `md5` や `sha1` でハッシュしてもまだ安全だと思っている開発者がたくさんいます。かつては、前述のハッシュ・アルゴリズムを使えば十分であった時もありましたが、 現

代のハードウェアをもってすれば、そのようなハッシュはブルート・フォース・アタックを使って非常に簡単に復元することが可能です。

最悪のシナリオ (アプリケーションに侵入された場合) であっても、ユーザのパスワードについて強化されたセキュリティを提供することが出来るように、 ブルート・フォース・アタックに対する耐性が強いハッシュ・アルゴリズムを使う必要があります。現在、最善の選択は `bcrypt` です。 PHP では、crypt 関数[4] を使って `bcrypt` ハッシュを生成することが出来ます。 Yii は `crypt` を使ってハッシュを安全に生成し検証することを容易にするために、二つのヘルパ関数を提供しています。

ユーザが初めてパスワードを提供するとき (例えば、ユーザ登録の時) には、パスワードをハッシュする必要があります。

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

そして、ハッシュを対応するモデル属性と関連付けて、後で使用するためにデータベースに保存します。

ユーザがログインを試みたときは、送信されたパスワードは、前にハッシュされて保存されたパスワードと照合して検証されなければなりません。

```
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {
    // よろしい、ユーザをログインさせる
} else {
    // パスワードが違う
}
```

9.5 暗号化

このセクションでは、セキュリティの以下の側面について見ていきます。

- 乱数データの生成
- 暗号化と復号化
- データの完全性の確認

9.5.1 擬似乱数データを生成する

擬似乱数データはさまざまな状況で役に立ちます。 例えば、メール経由でパスワードをリセットするときは、トークンを生成してデータベースに保存し、それをユーザにメールで送信します。 そして、ユーザはこのトークンを自分がアカウントの所有者であることの証拠として使用します。このトークンがユニークかつ推測困難なものであることは非常に重要なことです。 さもなくば、攻撃者がトークンの値を推測してユーザのパスワードをリセットする可能性があります。

Yii のセキュリティヘルパは擬似乱数データの生成を単純な作業にしてくれます。

[4] https://secure.php.net/manual/ja/function.crypt.php

```
$key = Yii::$app->getSecurity()->generateRandomString();
```

9.5.2 暗号化と復号化

Yii は秘密鍵を使ってデータを暗号化/復号化することを可能にする便利なヘルパ関数を提供しています。データを暗号化関数に渡して、秘密鍵を持つ者だけが復号化することが出来るようにすることが出来ます。 例えば、何らかの情報をデータベースに保存する必要があるけれども、(たとえアプリケーションのデータベースが第三者に漏洩した場合でも) 秘密鍵を持つユーザだけがそれを見ることが出来るようにする必要がある、という場合には次のようにします。

```
// $data と $secretKey はフォームから取得する
$encryptedData = Yii::$app->getSecurity()->encryptByPassword($data,
    $secretKey);
// $encryptedData をデータベースに保存する
```

そして、後でユーザがデータを読みたいときは、次のようにします。

```
// $secretKey はユーザ入力から取得、$encryptedData はデータベースから取得
$data = Yii::$app->getSecurity()->decryptByPassword($encryptedData,
    $secretKey);
```

yii\base\Security::encryptByKey() と yii\base\Security::decryptByKey() によって、 パスワードの代わりにキーを使うことも可能です。

9.5.3 データの完全性を確認する

データが第三者によって改竄されたり、更には何らかの形で毀損されたりしていないことを確認する必要がある、という場合があります。Yii は二つのヘルパ関数の形で、データの完全性を確認するための簡単な方法を提供しています。

秘密鍵とデータから生成されたハッシュをデータにプレフィクスします。

```
// $secretKey はアプリケーションまたはユーザの秘密、$genuineData は信頼できる
    ソースから取得
$data = Yii::$app->getSecurity()->hashData($genuineData, $secretKey);
```

データの完全性が毀損されていないかチェックします。

```
// $secretKey はアプリケーションまたはユーザの秘密、$data は信頼できないソース
    から取得
$data = Yii::$app->getSecurity()->validateData($data, $secretKey);
```

9.6 セキュリティのベスト・プラクティス

下記において、一般的なセキュリティの指針を復習し、Yii を使ってアプリケーションを開発するときに脅威を回避する方法を説明します。 これ

らの原則のほとんどのものは Yii に固有のものではなく、ウェブ・サイトまたはソフトウェアの開発一般に適用されるものです。 従って、これらの原則の背後にある一般的な考え方について、さらに参照すべき文書へのリンクが追加されています。

9.6.1 基本的な指針

どのようなアプリケーションが開発されているかに関わらず、セキュリティに関しては二つの大きな指針が存在します。

1. 入力をフィルタする。
2. 出力をエスケープする。

入力をフィルタする

入力をフィルタするとは、入力値は決して安全なものであると見なさず、取得した値が実際に許容されるものであるかどうかを、 常にチェックしなければならない、ということを意味します。例えば、並べ替えが三つのフィールド `title`、`created_at` および `status` によって実行され、フィールドの名前がユーザの入力によって提供されるものであることが判っている場合、取得した値を受信するその場でチェックする方が良い、ということです。 基本的な PHP の形式では、次のようなコードになります。

```
$sortBy = $_GET['sort'];
if (!in_array($sortBy, ['title', 'created_at', 'status'])) {
  throw new Exception('sort の値が不正です。');
}
```

Yii においては、たいていの場合、同様のチェックを行うために フォームの検証 を使うことになるでしょう。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/Data_Validation
- https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

出力をエスケープする

データを使用するコンテキストに応じて、出力をエスケープしなければなりません。 つまり、HTML のコンテキストでは、< や > などの特殊な文字をエスケープしなければなりません。 JavaScript や SQL のコンテキストでは、対象となる文字は別のセットになります。 全てを手動でエスケープするのは間違いを生じやすいことですから、Yii は異なるコンテキストに応じたエスケープを実行するためのさまざまなツールを提供しています。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/Command_Injection

- https://www.owasp.org/index.php/Code_Injection
- https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

9.6.2 SQL インジェクションを回避する

SQL インジェクションは、次のように、エスケープされていない文字列を連結してクエリ・テキストを構築する場合に発生します。

```
$username = $_GET['username'];
$sql = "SELECT * FROM user WHERE username = '$username'";
```

正しいユーザ名を提供する代りに、攻撃者は `'; DROP TABLE user; --` のような文字列をあなたのアプリケーションに与えることが出来ます。 結果として構築される SQL は次のようになります。

```
SELECT * FROM user WHERE username = ''; DROP TABLE user; --'
```

これは有効なクエリで、空のユーザ名を持つユーザを探してから、`user` テーブルを削除します。 おそらく、ウェブ・サイトは破壊されて、データは失われることになります (定期的なバックアップは設定済みですよね、ね?)。

Yii においては、ほとんどのデータベース・クエリは、PDO のプリペアド・ステートメントを適切に使用する アクティブ・レコード を経由して実行されます。 プリペアド・ステートメントの場合は、上で説明したようなクエリの改竄は不可能です。

それでも、生のクエリ や クエリ・ビルダ を必要とする場合はあります。 その場合には、データを渡すための安全な方法を使わなければなりません。データをカラムの値として使う場合は、プリペアド・ステートメントを使うことが望まれます。

```
// クエリ・ビルダ
$userIDs = (new Query())
    ->select('id')
    ->from('user')
    ->where('status=:status', [':status' => $status])
    ->all();

// DAO
$userIDs = $connection
    ->createCommand('SELECT id FROM user where status=:status')
    ->bindValues([':status' => $status])
    ->queryColumn();
```

データがカラム名やテーブル名を指定するために使われる場合は、事前定義された一連の値だけを許可するのが最善の方法です。

```
function actionList($orderBy = null)
{
    if (!in_array($orderBy, ['name', 'status'])) {
        throw new BadRequestHttpException('name と status だけを並べ替えに使うことが出来ます。')
```

```
    }

    // ...
}
```

それが不可能な場合は、テーブル名とカラム名をエスケープしなければなりません。 Yii はそういうエスケープのための特別な文法を持っており、それを使うと、サポートされている全てのデータベースに対して同じ方法でエスケープすることが出来ます。

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

この文法の詳細は、テーブルとカラムの名前を引用符で囲む で読むことが出来ます。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/SQL_Injection

9.6.3 XSS を回避する

XSS すなわちクロス・サイト・スクリプティングは、ブラウザに HTML を出力する際に、出力が適切にエスケープされていないと発生します。例えば、ユーザ名を入力できるフォームで `Alexander` の代りに `<script>alert('Hello!');</script>` と入力した場合、 ユーザ名をエスケープせずに出力している全てのページでは、JavaScript `alert('Hello!');` が実行されて、ブラウザにアラート・ボックスがポップアップ表示されます。 ウェブ・サイト次第では、そのようなスクリプトを使って、無害なアラートではなく、あなたの名前を使ってメッセージを送信したり、 さらには銀行取引を実行したりすることが可能です。

XSS の回避は、Yii においてはとても簡単です。一般に、二つのケースがあります。

1. データをプレーン・テキストとして出力したい。
2. データを HTML として出力したい。

プレーン・テキストしか必要でない場合は、エスケープは次のようにとても簡単です。

```
<?= \yii\helpers\Html::encode($username) ?>
```

HTML である場合は、HtmlPurifier から助けを得ることが出来ます。

```
<?= \yii\helpers\HtmlPurifier::process($description) ?>
```

HtmlPurifier の処理は非常に重いので、キャッシュを追加することを検討してください。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

9.6.4 CSRF を回避する

CSRF は、クロス・サイト・リクエスト・フォージェリ (cross-site request forgery) の略称です。 多くのアプリケーションは、ユーザのブラウザから来るリクエストはユーザ自身によって発せられたものだと仮定しているけれども、その仮定は間違っているかもしれない ... というのが CSRF の考え方です。

例えば、`an.example.com` というウェブ・サイトが `/logout` という URL を持っており、 この URL を単純な GET でアクセスするとユーザをログアウトさせるようになっているとします。 ユーザ自身によってこの URL がリクエストされる限りは何も問題はありませんが、ある日、悪い奴が、ユーザが頻繁に訪れるフォーラムに `<img src="http://an.example.com/logout">` というリンクを含むコンテントを何らかの方法で投稿します。ブラウザは画像のリクエストとページのリクエストの間に何ら区別を付けませんので、ユーザがそのような `img` タグを含むページを開くとブラウザはその URL に対して GET リクエストを送信します。 そして、ユーザが `an.example.com` からログアウトされてしまうことになる訳です。

これは CSRF 攻撃がどのように動作するかという基本的な考え方です。ユーザがログアウトされるぐらいは大したことではない、と言うことも出来るでしょう。 しかしこれは例に過ぎません。この考え方を使って、支払いを開始させたり、データを変更したりというような、もっとひどいことをすることも出来ます。 `http://an.example.com/purse/transfer?to=anotherUser&amount=2000` という URL を持つウェブ・サイトがあると考えて見てください。この URL に GET リクエストを使ってアクセスすると、権限を持つユーザ・アカウントから `anotherUser` に $2000 が送金されるのです。 私たちは、ブラウザは画像をロードするのに常に GET リクエストを使う、ということを知っていますから、 この URL が POST リクエストだけを受け入れるようにコードを修正することは出来ます。 しかし残念なことに、それで問題が解決する訳ではありません。攻撃者は `<img>` タグの代りに何らかの JavaScript コードを書いて、 その URL に対する POST リクエストの送信を可能にすることが出来ます。

これを理由として、Yii は CSRF 攻撃を防御するための追加のメカニズムを適用します。

CSRF を回避するためには、常に次のことを守らなければなりません。

1. HTTP の規格、すなわち、GET はアプリケーションの状態を変更すべきではない、という規則に従うこと。 詳細は RFC2616[5] を参照して下さい。

2. Yii の CSRF 保護を有効にしておくこと。

場合によっては、コントローラやアクションの単位で CSRF 検証を無効

[5] https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

化する必要があることがあるでしょう。これは、そのプロパティを設定することによって達成することが出来ます。

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $enableCsrfValidation = false;

    public function actionIndex()
    {
        // CSRF 検証はこのアクションおよびその他のアクションに対して適用されな
    い
    }

}
```

特定のアクションに対して CSRF 検証を無効化したいときは、次のようにすることが出来ます。

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function beforeAction($action)
    {
        // ... ここで何らかの条件に従って ‘$this->enableCsrfValidation‘ を設
    定する ...
        // 親のメソッドを呼ぶ。プロパティが true であれば、その中で CSRF が
    チェックされる。
        return parent::beforeAction($action);
    }
}
```

スタンドアロン・アクション における CSRF 検証の無効化は `init()` メソッドの中で行わなければなりません。 このコードは `beforeRun()` メソッドに置いてはいけません。そこでは効果がありません。

```
<?php

namespace app\components;

use yii\base\Action;

class ContactAction extends Action
{
    public function init()
    {
        parent::init();
        $this->controller->enableCsrfValidation = false;
```

```
    }

    public function run()
    {
          $model = new ContactForm();
          $request = Yii::$app->request;
          if ($request->referrer === 'yiipowered.com'
              && $model->load($request->post())
              && $model->validate()
          ) {
              $model->sendEmail();
          }
    }
}
```

> 警告: CSRF を無効化すると、あらゆるサイトから POST リクエストをあなたのサイトに送信することが出来るようになります。その場合には、IP アドレスや秘密のトークンをチェックするなど、追加の検証を実装することが重要です。
>
> 補足: バージョン 2.0.21 以降、Yii は `sameSite` クッキー設定 (PHP バージョン 7.3.0 以上が必要) をサポートしています。ただし、`sameSite` クッキー設定を行えば、上記の CSRF 対策が不要になるということではありません。何故なら、今はまだ全てのブラウザがこの設定をサポートしている訳ではないからです。 詳細については セッションとクッキー - sameSite オプション を参照して下さい。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/CSRF
- https://www.owasp.org/index.php/SameSite

9.6.5 ファイルの曝露を回避する

デフォルトでは、サーバのウェブ・ルートは、`index.php` がある `web` ディレクトリを指すように意図されています。 共有ホスティング環境の場合、それをすることが出来ずに、全てのコード、構成情報、ログをサーバのウェブ・ルートの下に置かなくてはならないことがあり得ます。

そういう場合には、`web` 以外の全てに対してアクセスを拒否することを忘れないでください。 それも出来ない場合は、アプリケーションを別の場所でホストすることを検討してください。

9.6.6 本番環境ではデバッグ情報とデバッグ・ツールを無効にする

デバッグ・モードでは、Yii は極めて多くのエラー情報を出力します。これは確かに開発には役立つものです。 しかし、実際の所、これらの饒

舌なエラー情報は、攻撃者にとっても、データベース構造、構成情報の値、コードの断片などを曝露してくれる重宝なものです。 本番でのアプリケーションにおいては、決して `index.php` の `YII_DEBUG` を `true` にして走らせてはいけません。

本番環境では Gii やデバッグ・ツール・バーを決して有効にしてはいけません。 これらを有効にすると、データベース構造とコードに関する情報を得ることが出来るだけでなく、コードを Gii によって生成したもので書き換えることすら出来てしまいます。

デバッグ・ツール・バーは本当に必要でない限り本番環境では使用を避けるべきです。これはアプリケーションと構成情報の全ての詳細を曝露することが出来ます。 どうしても必要な場合は、あなたの IP だけに適切にアクセス制限されていることを再度チェックしてください。

このトピックについて更に読むべき文書:

- https://www.owasp.org/index.php/Exception_Handling
- https://www.owasp.org/index.php/Top_10_2007-Information_Leakage

9.6.7 TLS によるセキュアな接続を使う

Yii が提供する機能には、クッキーや PHP セッションに依存するものがあります。これらのものは、接続が侵害された場合には、脆弱性となり得ます。 アプリケーションが TLS (しばしば SSL[6] と呼ばれます) によるセキュアな接続を使用している場合は、この危険性を減少させることが出来ます。

その設定の仕方については、あなたのウェブ・サーバのドキュメントの指示を参照してください。 H5BP プロジェクトが提供する構成例を参考にすることも出来ます。

- Nginx[7]
- Apache[8].
- IIS[9].
- Lighttpd[10].

> 補足: TLS が構成されているときは、(セッションの)クッキーを TLS のみで送信することが推奨されます。 これは、セッション および/または クッキーの の `secure` フラグを設定することで達成されます。 詳細は セッションとクッキー - secure フラグ を参照して下さい。

9.6.8 サーバの構成をセキュアにする

このセクションの目的は、Yii ベースのウェブ・サイトをホストするサー

[6] https://ja.wikipedia.org/wiki/Transport_Layer_Security
[7] https://github.com/h5bp/server-configs-nginx
[8] https://github.com/h5bp/server-configs-apache
[9] https://github.com/h5bp/server-configs-iis
[10] https://github.com/h5bp/server-configs-lighttpd

バの構成を作成するときに、 考慮に入れなければならないリスクに照明を当てることにあります。 ここで触れられる問題点以外にも、セキュリティに関連して考慮すべき構成オプションがあるかもしれません。 このセクションの説明が完全であるとは考えないで下さい。

Host ヘッダ攻撃を避ける

yii\web\UrlManager や yii\helpers\Url のクラスは、リンクを生成するために 現在リクエストされているホスト名 を使うことがあります。 ウェブ・サーバが Host ヘッダの値とは無関係に同じサイトとして応答するように構成されている場合は、 この情報は信頼できないものになっており、HTTP リクエストを送信するユーザによって偽装されている[11] 可能性があります。 そのような状況においては、ウェブ・サーバの構成を改修して、指定されたホスト名に対してのみ応答するようにするか、 または、request アプリケーション・コンポーネントの hostInfo プロパティを設定して、 ホスト名の値を明示的に設定ないしフィルタするか、どちらかの対策を取るべきです。

サーバの構成についての詳細な情報は、ウェブ・サーバのドキュメントを参照して下さい。

- Apache 2: http://httpd.apache.org/docs/trunk/vhosts/examples.html#defaultallports
- Nginx: https://www.nginx.com/resources/wiki/start/topics/examples/server_blocks/

サーバの構成にアクセスする権限がない場合は、このような攻撃に対して防御するために、yii\filters\HostControl フィルタを設定することが出来ます。

```
// ウェブ・アプリケーション構成ファイル
return [
    'as hostControl' => [
        'class' => 'yii\filters\HostControl',
        'allowedHosts' => [
            'example.com',
            '*.example.com',
        ],
        'fallbackHostInfo' => 'https://example.com',
    ],
    // ...
];
```

> 補足: 「ホスト・ヘッダ攻撃」に対する保護のためには、常に、フィルタの使用よりもウェブ・サーバの構成を優先すべきです。 yii\filters\HostControl は、サーバの構成が出来ない場合にだけ使うべきものです。

[11] https://www.acunetix.com/vulnerabilities/web/host-header-attack

Chapter 10

キャッシュ

10.1 キャッシュ

キャッシュはウェブ・アプリケーションのパフォーマンスを向上させるための安価で効果的な方法です。 比較的静的なデータをキャッシュに格納し、要求に応じてキャッシュからそれを取得します。 これによって、アプリケーションは、毎回一からデータを生成した場合に必要になるであろう時間を節約することができます。

キャッシュはアプリケーション内のさまざまなレベルと場所で使用することができます。 例えばサーバ・サイドでの低いレベルでは、データベースから取得した最新の記事情報リストのような 基本的なデータを格納するためにキャッシュを使用することが出来ます。 より高いレベルでは、ウェブ・ページの断片または全体、例えば、最新の記事のレンダリング結果を格納するためにキャッシュを使用することが出来ます。 クライアント・サイドでは、最近訪れたページの内容をブラウザのキャッシュに格納するために、HTTP キャッシュを使用することができます。

Yii はこれら全てのキャッシュ機構をサポートしています:

- データ・キャッシュ
- フラグメント・キャッシュ
- ページ・キャッシュ
- HTTP キャッシュ

10.2 データ・キャッシュ

データ・キャッシュは PHP の変数をキャッシュに格納し、あとでキャッシュからそれらを読み込みます。 これは、クエリ・キャッシュ や ページ・キャッシュ など、 より高度なキャッシュ機能の基礎でもあります。

以下のコードが、データ・キャッシュの典型的な利用パターンです。ここで、`$cache` は キャッシュ・コンポーネント を指しています。

```
// キャッシュから $data を取得しようと試みる
$data = $cache->get($key);
```

```
if ($data === false) {
    // キャッシュの中に $data が見つからない場合は一から作る
    $data = $this->calculateSomething();

    // $data をキャッシュに格納して、次回はそれを取得できるようにする
    $cache->set($key, $data);
}

// この時点で $data は利用可能になっている
```

バージョン 2.0.11 以降は、キャッシュ・コンポーネント が提供する getOrSet() メソッドを使って、 データを取得、計算、保存するためのコードを単純化することが出来ます。 次に示すコードは、上述の例と全く同じことをするものです。

```
$data = $cache->getOrSet($key, function () {
    return $this->calculateSomething();
});
```

キャッシュが $key と関連づけられたデータを保持している場合は、キャッシュされている値が返されます。 そうでない場合は、渡された無名関数が実行されて値が計算され、その値がキャッシュされるとともに返されます。

無名関数が外部のスコープの何らかのデータを必要とする場合は、それを use 文を使って渡すことが出来ます。 例えば、

```
$user_id = 42;
$data = $cache->getOrSet($key, function () use ($user_id) {
    return $this->calculateSomething($user_id);
});
```

> 補足: getOrSet() メソッドは、有効期限と依存もサポートしています。 詳しくは キャッシュの有効期限 と キャッシュの依存 を参照してください。

10.2.1 キャッシュ・コンポーネント

データ・キャッシュはメモリ、ファイル、データベースなどさまざまなキャッシュ・ストレージを表す、 いわゆる キャッシュ・コンポーネント に依存しています。

キャッシュ・コンポーネントは通常グローバルに設定しアクセスできるように アプリケーション・コンポーネント として登録されます。 以下のコードは、二台のキャッシュ・サーバを用いる Memcached[1] を使うように cache アプリケーション・コンポーネントを構成する方法を示すものです。

[1] http://memcached.org/

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

こうすると、上記のキャッシュ・コンポーネントに `Yii::$app->cache` という式でアクセスできるようになります。

すべてのキャッシュ・コンポーネントは同じ API をサポートしているので、アプリケーションの構成情報で設定しなおせば、 キャッシュを使っているコードに変更を加えることなく、異なるキャッシュ・コンポーネントに入れ替えることができます。 例えば上記の構成を APC キャッシュ を使うように変更する場合は以下のようにします:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> ヒント: キャッシュ・コンポーネントは複数登録することができます。`cache` という名前のコンポーネントが、 キャッシュに依存する多数のクラスによってデフォルトで使用されます (例えば `yii\web\UrlManager` など) 。

サポートされているキャッシュ・ストレージ

Yii はさまざまなキャッシュ・ストレージをサポートしています。以下がその概要です:

- `yii\caching\ApcCache`: PHP の APC[2] 拡張モジュールを使用します。 集中型の重厚なアプリケーションのキャッシュを扱うときには最速の一つとして考えることができます (例えば、サーバが一台で、専用のロード・バランサを持っていない、などの場合)。
- `yii\caching\DbCache`: キャッシュされたデータを格納するためにデータベースのテーブルを使用します。 このキャッシュを使用する

[2] https://secure.php.net/manual/ja/book.apc.php

には yii\caching\DbCache::$cacheTable で指定したテーブルを作成する必要があります。

- yii\caching\ArrayCache: 配列に値を保存することによって、現在のリクエストのためだけのキャッシュを提供します。 ArrayCache のパフォーマンスを高めるために、yii\caching\ArrayCache::$serializer を false に設定して、 保存するデータのシリアライズを無効にすることが出来ます。 .
- yii\caching\DummyCache: 実際にはキャッシュを行わない、キャッシュのプレースホルダとして働きます。 このコンポーネントの目的は、キャッシュの可用性をチェックする必要があるコードを簡略化することです。 たとえば、開発中やサーバに実際のキャッシュ・サポートがない場合でも、 このキャッシュを使用するようにキャッシュ・コンポーネントを構成することができます。 そして、実際のキャッシュ・サポートが有効になったときに、対応するキャッシュ・コンポーネントに切替えて使用します。 どちらの場合も、Yii::$app->cache が null かも知れないと心配せずに、 データを取得するために同じコード Yii::$app->cache->get($key) を使用できます。
- yii\caching\FileCache: キャッシュされたデータを保存するために通常のファイルを使用します。 これはページ・コンテントなど大きなかたまりのデータに特に適しています。
- yii\caching\MemCache: PHP の Memcache[3] と Memcached[4] 拡張モジュールを使用します。 分散型のアプリケーションでキャッシュを扱うときには最速の一つとして考えることができます (例えば、複数台のサーバで、ロード・バランサがある、などの場合) 。
- yii\redis\Cache: Redis[5] の key-value ストアに基づいてキャッシュ・コンポーネントを実装しています。 (Redis の バージョン 2.6.12 以降が必要とされます) 。
- yii\caching\WinCache: PHP の WinCache[6] エクステンションを使用します。 (参照リンク[7])
- yii\caching\XCache *(非推奨)*: PHP の XCache[8] 拡張モジュールを使用します。
- yii\caching\ZendDataCache *(非推奨)*: キャッシュ・メディアとして Zend Data Cache[9] を使用します。

ヒント: 同じアプリケーション内で異なるキャッシュを使用することもできます。 一般的なやり方として、小さくとも常に

[3] https://secure.php.net/manual/ja/book.memcache.php
[4] https://secure.php.net/manual/ja/book.memcached.php
[5] http://redis.io/
[6] http://iis.net/downloads/microsoft/wincache-extension
[7] https://secure.php.net/manual/ja/book.wincache.php
[8] http://xcache.lighttpd.net/
[9] http://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

使用されるデータ (例えば、統計データ) を格納する場合はメモリ・ベースのキャッシュ・ストレージを使用し、 大きくて使用頻度の低いデータ (例えば、ページ・コンテント) を格納する場合はファイル・ベース、またはデータベースのキャッシュ・ストレージを使用します 。

10.2.2 キャッシュ API

すべてのキャッシュ・コンポーネントが同じ基底クラス `yii\caching\Cache` を持っているので、以下の API をサポートしています。

- `get()`: 指定されたキーを用いてキャッシュからデータを取得します。 データが見つからないか、もしくは有効期限が切れたり無効になったりしている場合は false を返します。
- `set()`: キーによって識別されるデータをキャッシュに格納します。
- `add()`: キーがキャッシュ内で見つからない場合に、キーによって識別されるデータをキャッシュに格納します。
- `getOrSet()`: 指定されたキーを用いてキャッシュからデータを取得します。 取得できなかった場合は、渡されたコールバック関数を実行し、関数の返り値をそのキーでキャッシュに保存し、そしてその値を返します。
- `multiGet()`: 指定されたキーを用いてキャッシュから複数のデータを取得します。
- `multiSet()`: キャッシュに複数のデータを格納します。各データはキーによって識別されます。
- `multiAdd()`: キャッシュに複数のデータを格納します。 各データはキーによって識別されます。もしキャッシュ内にキーがすでに存在する場合はスキップされます。
- `exists()`: 指定されたキーがキャッシュ内で見つかったかどうかを示す値を返します。
- `delete()`: キャッシュからキーによって識別されるデータを削除します。
- `flush()`: キャッシュからすべてのデータを削除します。

> 補足: boolean 型の `false` を直接にキャッシュしてはいけません。 なぜなら、`get()` メソッドは、データがキャッシュ内に見つからないことを示すために戻り値として `false` を使用しているからです。 代りに、配列内に `false` を置いてキャッシュすることによって、この問題を回避して下さい。

キャッシュされたデータを取得する際に発生するオーバーヘッドを減らすために、MemCache, APC などのいくつかのキャッシュ・ストレージは、 バッチ・モードで複数のキャッシュされた値を取得することをサポートしています。 `multiGet()` や `multiAdd()` などの API はこの機能を十分に引き出すために提供されています。 基礎となるキャッシュ・ス

トレージがこの機能をサポートしていない場合には、シミュレートされます。

yii\caching\Cache は ArrayAccess インタフェイスを継承しているので、キャッシュ・コンポーネントは配列のように扱うことができます。以下はいくつかの例です:

```
$cache['var1'] = $value1;  // $cache->set('var1', $value1); と同等
$value2 = $cache['var2'];  // $value2 = $cache->get('var2'); と同等
```

キャッシュのキー

キャッシュに格納される各データは、一意のキーによって識別されます。 キャッシュ内にデータを格納するときはキーを指定する必要があります。 また、あとでキャッシュからデータを取得するときは、それに対応するキーを提供しなければなりません。

キャッシュのキーとしては、文字列または任意の値を使用することができます。 キーが文字列でない場合は、自動的に文字列にシリアライズされます。

キャッシュのキーを定義する一般的なやり方として、全ての決定要素を配列の形で含めるという方方があります。 例えば yii\db\Schema はデータベース・テーブルのスキーマ情報を以下のキーを使用してキャッシュしています。

```
[
    __CLASS__,              // スキーマ・クラス名
    $this->db->dsn,         // データベース接続のデータ・ソース名
    $this->db->username,    // データベース接続のログイン・ユーザ
    $name,                  // テーブル名
];
```

見ての通り、キーは一意にデータベースのテーブルを指定するために必要なすべての情報を含んでいます。

> 補足: multiSet() または multiAdd() によってキャッシュに保存される値が持つことが出来るのは、 文字列または整数のキーだけです。それらより複雑なキーを設定する必要がある場合は、 set() または add() によって、値を個別に保存してください。

同じキャッシュ・ストレージが異なるアプリケーションによって使用されているときは、 キャッシュのキーの競合を避けるために、各アプリケーションではユニークなキーの接頭辞を指定する必要があります。 これは yii\caching\Cache::$keyPrefix プロパティを設定することで出来ます。例えば、アプリケーション構成情報で以下のように書くことができます:

```
'components' => [
    'cache' => [
```

```
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',       // ユニークなキャッシュのキーの接頭辞
    ],
],
```

相互運用性を確保するために、英数字のみを使用する必要があります。

キャッシュの有効期限

キャッシュに格納されたデータは、何らかのキャッシュ・ポリシー (例えば、キャッシュ・スペースがいっぱいになったときは最も古いデータが削除される、など) の執行によって除去されない限り、永遠に残り続けます。 この動作を変えるために set() を呼んでデータ・アイテムを保存するときに、有効期限パラメータを指定することができます。 このパラメータは、データ・アイテムが何秒間有効なものとしてキャッシュ内に残ることが出来るかを示します。 get() でデータ・アイテムを取得する際に有効期限が切れていた場合は、 キャッシュ内にデータが見つからなかったことを示す false が返されます。例えば、

```
// 最大で 45 秒間キャッシュ内にデータを保持する
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // $data は有効期限が切れているか、またはキャッシュ内に見つからない
}
```

バージョン 2.0.11 以降は、デフォルトの無限の有効期限に替えて特定の有効期限を指定したい場合には、 キャッシュ・コンポーネントの構成で defaultDuration の値を指定することが出来ます。 これによって、特定の duration パラメータを毎回 set() に渡さなくてもよくなります。

キャッシュの依存

有効期限の設定に加えて、キャッシュされたデータは、いわゆる キャッシュの依存 (キャッシュが依存している事物) の変化によって無効にすることもできます。 例えば yii\caching\FileDependency は、キャッシュがファイルの更新時刻に依存していることを表しています。 この依存が変化したときは、対応するファイルが更新されたことを意味します。 その結果、キャッシュ内で見つかった古いファイルのコンテントは、無効とされるべきであり get() は false を返さなければなりません。

キャッシュの依存は yii\caching\Dependency の子孫クラスのオブジェクトとして表現されます。 set() でキャッシュにデータ・アイテムを格納する際に、 関連するキャッシュの依存のオブジェクトを一緒に渡すことができます。例えば、

```
// example.txt ファイルの更新日時への依存を作成します。
```

```
$dependency = new \yii\caching\FileDependency(['fileName' => 'example.txt'])
    ;

// データは 30 秒で期限切れになります。
// さらに、example.txt が変更された場合、有効期限内でも無効になります。
$cache->set($key, $data, 30, $dependency);

// キャッシュはデータの有効期限が切れているかをチェックします。
// 同時に、関連付けられた依存が変更されているかもチェックします。
// これらの条件のいずれかが満たされている場合は false を返します。
$data = $cache->get($key);
```

以下は利用可能なキャッシュの依存の概要です:

- yii\caching\ChainedDependency: チェーン上のいずれかの依存が変更された場合に、依存が変更されます。
- yii\caching\DbDependency: 指定された SQL 文のクエリ結果が変更された場合、依存が変更されます。
- yii\caching\ExpressionDependency: 指定された PHP の式の結果が変更された場合、依存が変更されます。
- yii\caching\FileDependency: ファイルの最終更新日時が変更された場合、依存が変更されます。
- yii\caching\TagDependency: キャッシュされるデータ・アイテムに一つまたは複数のタグを関連付けます。 yii\caching\TagDependency::invalidate() を呼び出すことによって、指定されたタグ (複数可) を持つキャッシュされたデータ・アイテムを無効にすることができます。

  補足: 依存を有するキャッシュについて exists() メソッドを使用することは避けてください。 このメソッドは、キャッシュされたデータに関連づけられた依存がある場合でも、依存が変化したかどうかをチェックしません。 つまり、exists() が true を返しているのに、 get() が false を返すという場合があり得ます。

10.2.3 クエリ・キャッシュ

クエリ・キャッシュは、データ・キャッシュ上に構築された特別なキャッシュ機能で、 データベースのクエリ結果をキャッシュするために提供されています。

クエリ・キャッシュは データベース接続 と有効な cache アプリケーション・コンポーネント を必要とします。 $db を yii\db\Connection のインスタンスと仮定した場合、クエリ・キャッシュの基本的な使い方は以下のようになります:

```
$result = $db->cache(function ($db) {

    // クエリ・キャッシュが有効で、かつクエリ結果がキャッシュ内にある場合、
```

```
    // SQL クエリ結果がキャッシュから提供されます
    return $db->createCommand('SELECT * FROM customer WHERE id=1')->queryOne
    ();

});
```

クエリ・キャッシュは DAO だけではなく アクティブ・レコード でも使用することができます。

```
$result = Customer::getDb()->cache(function ($db) {
    return Customer::find()->where(['id' => 1])->one();
});
```

> 情報: いくつかの DBMS (例えば MySQL[10]) もデータベース・サーバ・サイドのクエリ・キャッシュをサポートしています。 どちらのクエリ・キャッシュ・メカニズムを選んでも構いません。 前述した Yii のクエリ・キャッシュにはキャッシュの依存を柔軟に指定できるという利点があり、潜在的にはより効率的です。

2.0.14 以降は、下記のショートカットを使用することが出来ます。

```
(new Query())->cache(7200)->all();
// および
User::find()->cache(7200)->all();
```

構成

クエリ・キャッシュには yii\db\Connection を通して設定可能な三つのグローバルなオプションがあります:

- enableQueryCache: クエリ・キャッシュを可能にするかどうか。デフォルトは true です。 実効的にクエリ・キャッシュをオンにするには queryCache によって指定される有効なキャッシュを持っている必要があることに注意してください。
- queryCacheDuration: これはクエリ結果がキャッシュ内に有効な状態として 持続できる秒数を表します。 クエリ・キャッシュを永遠にキャッシュに残したい場合は 0 を指定することができます。 このプロパティは yii\db\Connection::cache() が持続時間を指定せず呼び出されたときに使用されるデフォルト値です。
- queryCache: これはキャッシュ・アプリケーション・コンポーネントの ID を表します。 デフォルトは 'cache' です。有効なキャッシュ・コンポーネントが存在する場合にのみ、クエリ・キャッシュが使用可能になります。

[10] http://dev.mysql.com/doc/refman/5.1/ja/query-cache.html

使い方

クエリ・キャッシュを使用する必要がある複数の SQL クエリを持っている場合は yii\db\Connection::cache() を使用することができます。使い方は以下のとおりです。

```
$duration = 60;     // クエリ結果を 60 秒間 キャッシュ
$dependency = ...;  // 依存のオプション

$result = $db->cache(function ($db) {

    // ... ここで SQL クエリを実行します ...

    return $result;

}, $duration, $dependency);
```

無名関数内の任意の SQL クエリは、指定した依存とともに指定された期間キャッシュされます。 もしキャッシュ内に有効なクエリ結果が見つかった場合は、クエリはスキップされ、代りに結果がキャッシュから提供されます。 $duration の指定がない場合 queryCacheDuration で指定されている値が代りに使用されます。

場合によっては cache() 内でいくつかの特定のクエリに対してクエリ・キャッシュを無効にしたいことが有るでしょう。 そのときは yii\db\Connection::noCache() を使用します。

```
$result = $db->cache(function ($db) {

    // クエリ・キャッシュを使用する SQL クエリ

    $db->noCache(function ($db) {

        // クエリ・キャッシュを使用しない SQL クエリ

    });

    // ...

    return $result;
});
```

単一のクエリのためだけにクエリ・キャッシュを使用したい場合は、コマンドを構築するときに yii\db\Command::cache() を呼び出すことができます。例えば、

```
// クエリ・キャッシュを使い、期間を 60 秒にセットする
$customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->cache
    (60)->queryOne();
```

また、一つのコマンドに対してクエリ・キャッシュを無効にするために yii\db\Command::noCache() を使用することもできます。例えば、

```
$result = $db->cache(function ($db) {
```

```
    // クエリ・キャッシュを使用する SQL クエリ

    // このコマンドにはクエリ・キャッシュを使用しない
    $customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->
    noCache()->queryOne();

    // ...

    return $result;
});
```

制約

リソース・ハンドラを返すようなクエリにはクエリ・キャッシュは働きません。 例えば、いくつかの DBMS において BLOB 型のカラムを用いる場合、 クエリ結果はカラム・データに対するリソース・ハンドラを返します。

いくつかのキャッシュ・ストレージはサイズに制約があります。 例えば Memcache では、各エントリのサイズは 1MB が上限値です。 そのためクエリ結果のサイズがこの制約を越える場合、キャッシュは失敗します。

10.2.4 キャッシュのフラッシュ <span id=""cache-flushing">

保存されている全てのキャッシュ・データを無効化する必要がある場合は、`yii\caching\Cache::flush()` を呼ぶことが出来ます。

コンソールから `yii cache/flush` を呼ぶことによっても、キャッシュをフラッシュすることが出来ます。

- `yii cache`: アプリケーションで利用可能なキャッシュのリストを表示します。
- `yii cache/flush cache1 cache2`: キャッシュ・コンポーネント `cache1` と `cache2` をフラッシュします (複数のコンポーネント名をスペースで区切って渡すことが出来ます)
- `yii cache/flush-all`: アプリケーションの全てのキャッシュ・コンポーネントをフラッシュします。
- `yii cache/flush-schema db`: 指定された DB 接続に対する DB スキーマ・キャッシュをクリアします。

> 情報: デフォルトでは、コンソール・アプリケーションは独立した構成情報ファイルを使用します。 正しい結果を得るためには、ウェブとコンソールのアプリケーション構成で同じキャッシュ・コンポーネントを使用していることを確認してください。

10.3 フラグメント・キャッシュ

フラグメント・キャッシュは、ウェブ・ページの断片をキャッシュすることを指します。 例えば、ページ内の表に年間販売の概要が表示されている場合、リクエスト毎にこの表を生成するのにかかる時間を削減するために、キャッシュにこの表を格納することができます。 フラグメント・キャッシュは データ・キャッシュ 上に構築されています。

フラグメント・キャッシュを使用するには ビュー で以下の構文を使用します:

```
if ($this->beginCache($id)) {

    // ... ここに生成するコンテントを書く ...

    $this->endCache();
}
```

つまり、コンテント生成ロジックを beginCache() と endCache() の呼び出しのペアで囲みます。 コンテントがキャッシュ内で見つかった場合、beginCache() はキャッシュされたコンテントをレンダリングして false を返し、結果として、コンテント生成ロジックがスキップされます。 それ以外の場合はコンテント生成ロジックが呼ばれ、そして endCache() が呼ばれたときに、 生成されたコンテントがキャプチャされ、キャッシュに格納されます。

データ・キャッシュ と同様に、キャッシュされるコンテントを識別するためにユニークな $id が必要になります。

10.3.1 キャッシュのオプション

beginCache() メソッドの 2 番目のパラメータとしてオプションの配列を渡すことによって、 フラグメント・キャッシュに関する追加のオプションを指定することができます。 舞台の裏側では、このオプションの配列が、実際のフラグメント・キャッシュ機能を実装する yii\widgets\FragmentCache ウィジェットを構成するために使用されます。

持続時間

おそらくフラグメント・キャッシュで通常よく使われるであろうオプションは duration でしょう。 このオプションはコンテントがどれだけの時間キャッシュ内において有効であるかを指定します。 以下のコードは最大で 1 時間コンテントの断片をキャッシュします:

```
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... ここに生成するコンテントを書く ...

    $this->endCache();
}
```

このオプションがセットされていない場合は、デフォルトである 60 が使われます。すなわち、キャッシュされたコンテントの有効期限は 60 秒後に切れることになります。

依存

データ・キャッシュ と同様に、キャッシュされたコンテントの断片は依存を持つことができます。 例えば、表示されている投稿の内容は、投稿が変更されたか否かに依存します。

依存を指定するには dependency オプションに yii\caching\Dependency オブジェクトを指定するか、または依存オブジェクトを作成するための構成情報配列を指定します。 以下のコードはコンテントの断片が updated_at カラムの値の変化に依存していることを指定しています:

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... ここに生成するコンテントを書く ...

    $this->endCache();
}
```

バリエーション

キャッシュされるコンテントには、何らかのパラメータによってバリエーションを持たせることが出来ます。 例えば、複数の言語をサポートしているウェブ・アプリケーションでは、ビュー・コードの同じ部分が言語によってさまざまに異なるコンテントを生成することが有り得ます。 従って、現在のアプリケーションの言語に応じて、キャッシュされるコンテントのバリエーションを持つ必要があります。

キャッシュのバリエーションを指定するには variations オプションを指定します。 このオプションは、それぞれが特定のバリエーションの要素を表すスカラ値の配列でなければなりません。 例えば、言語によるキャッシュ・コンテントのバリエーションを持つためには、以下のコードを使います。

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... ここに生成するコンテントを書く ...

    $this->endCache();
}
```

キャッシュをトグルする

時として、ある条件が満たされた場合にのみフラグメント・キャッシュを有効にしたい場合があるでしょう。 たとえば、フォームが表示されているページでは、フォームをキャッシュしたいのは最初の (GET リクエストによる) リクエストの場合だけです。 その後の (POST リクエストによる) フォームの表示では、フォームにユーザ入力が含まれている可能性があるため、キャッシュをすべきではありません。 これを行うには、以下のように enabled オプションをセットします:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... ここに生成するコンテントを書く ...

    $this->endCache();
}
```

10.3.2 キャッシュのネスト

フラグメント・キャッシュはネストすることができます。つまり、キャッシュされる断片を、それ自体もキャッシュされる別の断片に入れることができます。 例えば、内側のフラグメント・キャッシュにはコメントがキャッシュされており、外側のフラグメント・キャッシュには記事内容と一緒にコメントもキャッシュされている、という形です。 以下のコードは 2 つのフラグメント・キャッシュをネストする方法を示すものです。

```
if ($this->beginCache($id1)) {

    // コンテント生成ロジック......

    if ($this->beginCache($id2, $options2)) {

        // コンテント生成ロジック......

        $this->endCache();
    }

    // コンテント生成ロジック......

    $this->endCache();
}
```

ネストされたキャッシュには、異なるキャッシュ・オプションを設定することができます。 たとえば、上記の例における内側のキャッシュと外側のキャッシュに対して、異なる持続期間の値を設定する事が可能です。 これによって、外側のキャッシュでキャッシュされたデータが無効になった場合でも、内側のキャッシュが有効な内側の断片を提供することが可能になります。 しかし、その逆は真ではありません。 外側のキャッシュが有効であると判断された場合には、内側のキャッシュが無

効になった後でも、外側のキャッシュが古くなったコンテントのコピーを提供し続けます。 従って、ネストされたキャッシュの持続時間や依存の設定を間違うと、無効になった内側のキャッシュ・データが外側のキャッシュに残り続けることになるので、注意が必要です。

10.3.3 ダイナミック・コンテント

フラグメント・キャッシュを使用する際、出力全体が比較的静的で、一ヶ所ないし数ヶ所だけが例外的に動的であるというような状況に遭遇するでしょう。 例えば、ページのヘッダがメイン・メニュー・バーと現在のユーザ名を一緒に表示している場合です。 もう一つの問題は、キャッシュされるコンテントに、リクエスト毎に実行しなければいけない PHP のコード (例えば、アセット・バンドルを登録するためのコード) が含まれている場合です。 この両方の問題は、いわゆる ダイナミック・コンテント 機能によって解決することができます。

ダイナミック・コンテントは、それがフラグメント・キャッシュの中に含まれていても、キャッシュすべきではない出力の部分を意味します。 このコンテントを常に動的にするためには、外側のコンテントがキャッシュから提供されている場合でも、 すべてのリクエストに対して、何らかの PHP コードを実行することにより生成しなければいけません。

ダイナミック・コンテントを目的の場所に挿入するには、以下のように、キャッシュされる断片内で `yii\base\View::renderDynamic()` を呼び出します。

```
if ($this->beginCache($id1)) {

    // コンテント生成ロジック.....

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // コンテント生成ロジック.....

    $this->endCache();
}
```

`renderDynamic()` メソッドはパラメータとして PHP コードを取ります。この PHP コードの戻り値が、ダイナミック・コンテントとして扱われます。 囲んでいる断片がキャッシュから提供されるか否かにかかわらず、同じ PHP コードがすべてのリクエストに対して実行されます。

> 補足: バージョン 2.0.14 以降、`yii\base\DynamicContentAwareInterface` インタフェイスとその `yii\base\DynamicContentAwareTrait` トレイトによって、ダイナミック・コンテント API が公開されています。. その一例としては、 `yii\widgets\FragmentCache` クラスを参照して下さい。

10.4 ページ・キャッシュ

ページ・キャッシュはサーバ・サイドでページ全体のコンテントをキャッシュするものです。 後で再び同じページがリクエストされた場合に、その内容を一から生成するのではなく、キャッシュから提供します。

ページ・キャッシュは yii\filters\PageCache という アクション・フィルタ によってサポートされています。 これは、コントローラ・クラスで以下のように使用することができます：

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index'],
            'duration' => 60,
            'variations' => [
                \Yii::$app->language,
            ],
            'dependency' => [
                'class' => 'yii\caching\DbDependency',
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
        ],
    ];
}
```

上記のコードは、ページ・キャッシュが index アクションのみで使用されることを示しています。 ページのコンテントは最大 60 秒間キャッシュされ、現在のアプリケーションの言語によるバリエーションを持ち、 投稿の総数に変化があった場合キャッシュされたページが無効になります。

御覧のように、ページ・キャッシュは フラグメント・キャッシュ ととてもよく似ています。 それらは両方とも duration、dependencies、variations、そして enabled などのオプションをサポートしています。主な違いとしては、ページ・キャッシュは アクション・フィルタ として、フラグメント・キャッシュは ウィジェット として実装されているということです。

フラグメント・キャッシュ も、ダイナミック・コンテント も、 ページ・キャッシュと併用することができます。

10.5 HTTP キャッシュ

これまでのセクションで説明したサーバ・サイドのキャッシュに加えて、ウェブ・アプリケーションは、 同じページ・コンテントを生成し送信する時間を節約するために、クライアント・サイドでもキャッシュを利用することができます。

クライアント・サイドのキャッシュを使用するには、レンダリング結果をキャッシュできるように、 コントローラ・アクションのフィルタとして yii\filters\HttpCache を設定します。 yii\filters\HttpCache は GET と HEAD リクエストに対してのみ動作し、それらのリクエストに対する 3 種類のキャッシュ関連の HTTP ヘッダを扱うことができます:

- Last-Modified
- Etag
- Cache-Control

10.5.1 Last-Modified ヘッダ

Last-Modified ヘッダは、クライアントがキャッシュした時からページが変更されたかどうかを示すために、タイムスタンプを使用しています。

Last-Modified ヘッダの送信を有効にするには yii\filters\HttpCache::$lastModified プロパティを構成します。 このプロパティは、ページの更新時刻に関する UNIX タイムスタンプを返す PHP のコーラブルでなければなりません。 この PHP コーラブルのシグニチャは以下のとおりです。

```
/**
 * @param Action $action 現在扱っているアクション・オブジェクト
 * @param array $params "params" プロパティの値
 * @return int ページの更新時刻を表す UNIX タイムスタンプ
 */
function ($action, $params)
```

以下は Last-Modified ヘッダを使用する例です:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('post')->max('updated_at');
            },
        ],
    ];
}
```

上記のコードは index アクションでのみ HTTP キャッシュを有効にすべきことを記述しています。 Last-Modified は、投稿の最終更新時刻に基づいて生成される必要があります。 ブラウザが初めて index ページにアクセスしたときは、ページはサーバ上で生成されブラウザに送信されます。 もしブラウザが再度同じページにアクセスし、その期間中に投稿に変更がない場合は、サーバはページを再生成せず、 ブラウザはクライアント・サイドにキャッシュしたものを使用します。 その結果、サー

バ・サイドのレンダリング処理とページ・コンテントの送信は両方ともスキップされます。

10.5.2 ETag ヘッダ

“Entity Tag” (略して ETag) ヘッダはページ・コンテントを表すためのハッシュです。 ページが変更された場合ハッシュも同様に変更されます。 サーバ・サイドで生成されたハッシュとクライアント・サイドで保持しているハッシュを比較することによって、ページが変更されたかどうか、そして再送信するべきかどうかを決定します。

ETag ヘッダの送信を有効にするには yii\filters\HttpCache::$etagSeed プロパティを設定します。 プロパティは ETag のハッシュを生成するためのシードを返す PHP のコーラブルで、 以下のようなシグネチャを持たなければなりません。

```
/**
 * @param Action $action 現在扱っているアクション・オブジェクト
 * @param array $params "params" プロパティの値
 * @return string ETag のハッシュを生成するためのシードとして使用する文字列
 */
function ($action, $params)
```

以下は ETag ヘッダを使用している例です:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['view'],
            'etagSeed' => function ($action, $params) {
                $post = $this->findModel(\Yii::$app->request->get('id'));
                return serialize([$post->title, $post->content]);
            },
        ],
    ];
}
```

上記のコードは view アクションでのみ HTTP キャッシュを有効にすべきことを記述しています。 Etag HTTP ヘッダは、リクエストされた投稿のタイトルとコンテントに基づいて生成されなければなりません。 ブラウザが初めて view ページにアクセスしたときは、ページがサーバ上で生成されブラウザに送信されます。 ブラウザが再度同じページにアクセスし、投稿のタイトルやコンテントに変更がない場合には、 サーバはページを再生成せず、ブラウザはクライアント・サイドにキャッシュしたものを使用します。 その結果、サーバ・サイドのレンダリング処理とページ・コンテント送信は両方ともスキップされます。

ETag は Last-Modified ヘッダよりも複雑 かつ/または より正確なキャッシング方式を可能にします。 例えば、サイトが別のテーマに切り替わった場合には ETag を無効化する、といったことができます。

ETag はリクエスト毎に再評価する必要があるため、負荷の高い生成方法を使うと HttpCache の本来の目的を損なって不必要なオーバーヘッドが生じる場合があります。 ページのコンテントが変更されたときにキャッシュを無効化するための式は単純なものを指定するようにして下さい。

> 補足: RFC 7232[11] に準拠して Etag と Last-Modified ヘッダの両方を設定した場合、 HttpCache はその両方とも送信します。 また、もし If-None-Match ヘッダと If-Modified-Since ヘッダの両方を送信した場合は前者のみが尊重されます。

10.5.3 Cache-Control ヘッダ

Cache-Control ヘッダはページのための一般的なキャッシュ・ポリシーを指定します。 yii\filters\HttpCache::$cacheControlHeader プロパティにヘッダの値を設定することで、それを送ることができます。 デフォルトでは、以下のヘッダが送信されます:

```
Cache-Control: public, max-age=3600
```

10.5.4 セッション・キャッシュ・リミッタ

ページでセッションを使用している場合、PHP はいくつかのキャッシュ関連の HTTP ヘッダ (PHP の INI 設定ファイル内で指定されている session.cache_limiter など) を自動的に送信します。 これらのヘッダが HttpCache が実現しようとしているキャッシュ機能を妨害したり無効にしたりすることがあります。 この問題を防止するために、HttpCache はこれらのヘッダの送信をデフォルトで自動的に無効化します。 この動作を変更したい場合は yii\filters\HttpCache::$sessionCacheLimiter プロパティを設定します。 このプロパティには public、private、private_no_expire、そして nocache などの文字列の値を使用することができます。 これらの値についての説明は session_cache_limiter()[12] を参照してください。

10.5.5 SEO への影響

検索エンジンのボットはキャッシュ・ヘッダを尊重する傾向があります。 クローラの中には、一定期間内に処理するドメインごとのページ数に制限を持っているものもあるため、 キャッシュ・ヘッダを導入して、処理の必要があるページ数を減らしてやると、サイトのインデックスの作成を促進できるかも知れません。

---

[11] http://tools.ietf.org/html/rfc7232#section-2.4
[12] https://secure.php.net/manual/ja/function.session-cache-limiter.php

Chapter 11

RESTful ウェブ・サービス

11.1 クイック・スタート

Yii は、RESTful ウェブサービス API を実装する仕事を簡単にするために、一揃いのツールを提供しています。 具体的に言えば、RESTful API に関する次の機能をサポートしています。

- アクティブ・レコード のための共通 API をサポートした迅速なプロトタイプ作成
- レスポンス形式のネゴシエーション (デフォルトで JSON と XML をサポート)
- 出力フィールドの選択をサポートした、カスタマイズ可能なオブジェクトのシリアライゼーション
- コレクション・データと検証エラーの適切な書式設定
- コレクションのページネーション、フィルタリングおよびソーティング
- HATEOAS[1] のサポート
- HTTP 動詞を適切にチェックする効率的なルーティング
- `OPTIONS` および `HEAD` 動詞のサポートを内蔵
- 認証と権限付与
- データ・キャッシュと HTTP キャッシュ
- レート制限

以下においては、例を使って、どのようにして最小限のコーディング労力で一組の RESTful API を構築することが出来るかを説明します。

ユーザのデータを RESTful API によって公開したいと仮定しましょう。 ユーザのデータは `user` という DB テーブルに保存されており、それにアクセスするための アクティブ・レコード クラス `app\models\User` が既に作成済みであるとします。

[1] http://en.wikipedia.org/wiki/HATEOAS

11.1.1 コントローラを作成する

最初に、コントローラ・クラス app\controllers\UserController を次のようにして作成します。

```
namespace app\controllers;

use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

このコントローラ・クラスは、よく使用される一揃いの RESTful アクションを実装した yii\rest\ActiveController を拡張するものです。modelClass として app\models\User が指定されているため、コントローラがどのモデルを使用してデータの取得と操作が出来るかがわかります。

11.1.2 URL 規則を構成する

次に、アプリケーションの構成情報において、urlManager コンポーネントの構成情報を修正します。

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

上記の構成情報は、主として、user コントローラの URL 規則を追加して、ユーザのデータが綺麗な URL と意味のある HTTP 動詞によってアクセスおよび操作できるようにするものです。

> 情報: Yii はコントローラの名前を自動的に複数形にしてエンド・ポイントとして使用します (下の「試してみる」(#trying-it-out) を参照してください)。この振る舞いは yii\rest\UrlRule::$pluralize プロパティを使って構成することが可能です。

11.1.3 JSON の入力を可能にする

API が JSON 形式で入力データを受け取ることが出来るように、request アプリケーション・コンポーネント の parsers プロパティを構成して、JSON 入力のために yii\web\JsonParser を使うようにします。

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

情報: 上記の構成はオプションです。 上記のように構成しない場合は、API は `application/x-www-form-urlencoded` と `multipart/form-data` だけを入力形式として認識します。

11.1.4 試してみる

上記で示した最小限の労力によって、ユーザのデータにアクセスする RESTful API を作成する仕事は既に完成しています。 作成した API は次のものを含みます。

- `GET /users`: 全てのユーザをページごとに一覧する
- `HEAD /users`: ユーザ一覧の概要を示す
- `POST /users`: 新しいユーザを作成する
- `GET /users/123`: ユーザ 123 の詳細を返す
- `HEAD /users/123`: ユーザ 123 の概要を示す
- `PATCH /users/123` と `PUT /users/123`: ユーザ 123 を更新する
- `DELETE /users/123`: ユーザ 123 を削除する
- `OPTIONS /users`: エンド・ボイント `/users` に関してサポートされている動詞を示す
- `OPTIONS /users/123`: エンド・ボイント `/users/123` に関してサポートされている動詞を示す

作成した API は、次のように、`curl` コマンドでアクセスすることが出来ます。

```
$ curl -i -H "Accept:application/json" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

受入れ可能なコンテント・タイプを `application/xml` に変更してみてください。 すると、結果が XML 形式で返されます。

```
$ curl -i -H "Accept:application/xml" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<response>
    <item>
        <id>1</id>
        ...
    </item>
    <item>
        <id>2</id>
        ...
    </item>
    ...
</response>
```

次のコマンドは、JSON 形式でユーザのデータを持つ POST リクエストを送信して、新しいユーザを作成します。

```
$ curl -i -H "Accept:application/json" -H "Content-Type:application/json" \
    -XPOST "http://localhost/users" \
    -d '{"username": "example", "email": "user@example.com"}'

HTTP/1.1 201 Created
...
Location: http://localhost/users/1
Content-Length: 99
Content-Type: application/json; charset=UTF-8

{"id":1,"username":"example","email":"user@example.com","created_at
    ":1414674789,"updated_at":1414674789}
```

> ヒント: URL `http://localhost/users` を入力すれば、ウェブ・ブラウザ経由で API にアクセスすることも出来ます。 ただし、特殊なリクエスト・ヘッダを送信するためには、何らかのブラウザプラグインが必要になるでしょう。

ご覧のように、レスポンス・ヘッダの中には、総ユーザ数やページ数などの情報が書かれています。 また、データの他のページへナビゲートす

ることを可能にするリンクもあります。 例えば、http://localhost/users?page=2 にアクセスすれば、ユーザのデータの次のページを取得することが出来ます。

fields と expand パラメータを使えば、どのフィールドが結果に含まれるべきかを指定することも出来ます。 例えば、URL http://localhost/users?fields=id,email は、id と email のフィールドだけを返します。

> 情報: 気がついたかも知れませんが、http://localhost/users の結果は、いくつかの公開すべきでないフィールド、例えば password_hash や auth_key を含んでいます。 当然ながら、これらが API の結果に出現することは避けたいでしょう。 リソース のセクションで説明されているように、これらのフィールドを除外することは出来ますし、また、除外しなければなりません。

さらに、http://localhost/users?sort=email や http://localhost/users?sort=-email のように、コレクションをソートすることも出来ます。 http://localhost/users?filter[id]=10 や http://localhost/users?filter[email][like]=gmail.com のように、 コレクションをフィルタリングすることも、データ・フィルターを使って実装することが出来ます。 詳細は、リソース のセクションを参照して下さい。

11.1.5 まとめ

Yii の RESTful API フレームワークを使う場合は、API エンド・ボイントをコントローラ・アクションの形式で実装します。 そして、コントローラを使って、単一タイプのリソースに対するエンド・ボイントを実装するアクションを編成します。

リソースは yii\base\Model クラスを拡張するデータ・モデルとして表現されます。 データベース (リレーショナルまたは NoSQL) を扱っている場合は、ActiveRecord を使ってリソースを表現することが推奨されます。

yii\rest\UrlRule を使って API エンド・ボイントへのルーティングを簡単にすることが出来ます。

これは要求されてはいませんが、RESTful API は、保守を容易にするために、 ウェブのフロントエンドやバックエンドとは別の独立したアプリケーションとして開発することが推奨されます。

11.2 リソース

RESTful API は、つまるところ、リソース にアクセスし、それを操作するものです。 MVC の枠組の中では、リソースは モデル として見ることが出来ます。

リソースをどのように表現すべきかについて制約がある訳ではありませんが、 Yii においては、通常は、次のような理由によって、リソースを yii\base\Model またはその子クラス (例えば yii\db\ActiveRecord) のオブジェクトとして表現することになります。

- yii\base\Model は yii\base\Arrayable インタフェイスを実装しています。 これによって、リソースのデータを RESTful API を通じて公開する仕方をカスタマイズすることが出来ます。
- yii\base\Model は 入力値の検証 をサポートしています。 これは、RESTful API がデータ入力をサポートする必要がある場合に役に立ちます。
- yii\db\ActiveRecord は DB データのアクセスと操作に対する強力なサポートを提供しています。 リソース・データがデータベースに保存されているときは、アクティブ・レコードが最適の選択です。

このセクションでは、主として、yii\base\Model クラス (またはその子クラス) から拡張したリソース・クラスにおいて、 RESTful API を通じて返すことが出来るデータを指定する方法を説明します。 リソース・クラスが yii\base\Model から拡張したものでない場合は、全てのパブリックなメンバ変数が返されます。

11.2.1 フィールド

RESTful API のレスポンスにリソースを含めるとき、リソースは文字列にシリアライズされる必要があります。 Yii はこのプロセスを二つのステップに分けます。 最初に、リソースは yii\rest\Serializer によって配列に変換されます。 次に、その配列が レスポンス・フォーマッタ によって、リクエストされた形式 (例えば JSON や XML) の文字列にシリアライズされます。 リソース・クラスを開発するときに主として力を注ぐべきなのは、最初のステップです。

fields() および/または extraFields() をオーバーライドすることによって、 リソースのどういうデータ (フィールド と呼ばれます) を配列表現に入れることが出来るかを指定することが出来ます。 この二つのメソッドの違いは、前者が配列表現に含まれるべきフィールドのデフォルトのセットを指定するのに対して、 後者はエンド・ユーザが expand クエリ・パラメータで要求したときに配列に含めることが出来る追加のフィールドを指定する、 という点にあります。例えば、

```
// fields() で宣言されている全てのフィールドを返す。
http://localhost/users

// "id" と "email" のフィールドだけを返す ただし、(fields() で宣言されていれ
    ば) 。
http://localhost/users?fields=id,email

// fields() の全てのフィールドと "profile" のフィールドを返す ただ
    し、("profile" が extraFields() で宣言されていれば)。
http://localhost/users?expand=profile
```

```
// fields() の全てのフィールドと post の "author" フィールドを返す
// ただし、("author" が post モデルの extraFields() にあれば）。
http://localhost/comments?expand=post.author

// "id" と"email" ただし、(fileds() で宣言されていれば) と
// "profile" ただし、(extraFields() で宣言されていれば) を返す。
http://localhost/users?fields=id,email&expand=profile
```

fields()‘ をオーバーライドする

デフォルトでは、yii\base\Model::fields() は、モデルの全ての属性をフィールドとして返し、 yii\db\ActiveRecord::fields() は、DB から投入された属性だけを返します。

fields() をオーバーライドして、フィールドを追加、削除、名前変更、または再定義することが出来ます。 fields() の返り値は配列でなければなりません。 配列のキーはフィールド名であり、配列の値は対応するフィールドの定義です。 フィールドの定義は、プロパティ/属性の名前か、あるいは、対応するフィールドの値を返す無名関数とすることが出来ます。 フィールド名がそれを定義する属性名と同一であるという特殊な場合においては、配列のキーを省略することが出来ます。例えば、

```
// 明示的に全てのフィールドをリストする方法。(API の後方互換性を保つため
    に) DB テーブルやモデル属性の
// 変更がフィールドの変更を引き起こさないことを保証したい場合に適している。
public function fields()
{
    return [
        // フィールド名が属性名と同じ
        'id',
        // フィールド名は "email、対応する属性名は" "email_address"
        'email' => 'email_address',
        // フィールド名は "name、その値は" PHP コールバックで定義
        'name' => function ($model) {
            return $model->first_name . ' ' . $model->last_name;
        },
    ];
}

// いくつかのフィールドを除去する方法。親の実装を継承しつつ、
// 公開すべきでないフィールドを除外したいときに適している。
public function fields()
{
    $fields = parent::fields();

    // 公開すべきでない情報を含むフィールドを削除する
    unset($fields['auth_key'], $fields['password_hash'], $fields['
    password_reset_token']);

    return $fields;
```

```
}
```

> 警告: デフォルトではモデルの全ての属性がエクスポートされる配列に含まれるため、データを精査して、 公開すべきでない情報が含まれていないことを確認すべきです。そういう情報がある場合は、 `fields()` をオーバーライドして、除去すべきです。上記の例では、`auth_key`、`password_hash` および `password_reset_token` を選んで除去しています。

`extraFields()` をオーバーライドする

デフォルトでは、`yii\base\Model::extraFields()` は空の配列を返し、`yii\db\ActiveRecord::extraFields()` は DB から取得されたリレーションの名前を返します。

`extraFields()` によって返されるデータの形式は `fields()` のそれと同じです。 通常、`extraFields()` は、主として、値がオブジェクトであるフィールドを指定するのに使用されます。 例えば、次のようなフィールドの宣言があるとしましょう。

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

`http://localhost/users?fields=id,email&expand=profile` というリクエストは、次のような JSON データを返すことが出来ます。

```
[
    {
        "id": 100,
        "email": "100@example.com",
        "profile": {
            "id": 100,
            "age": 30,
        }
    },
    ...
]
```

11.2.2 リンク

HATEOAS[2] は、Hypermedia as the Engine of Application State (アプリケーション状態のエンジンとしてのハイパーメディア) の略称です。

[2] http://en.wikipedia.org/wiki/HATEOAS

HATEOAS は、RESTful API は自分が返すリソースについて、どのようなアクションがサポートされているかをクライアントが発見できるような情報を返すべきである、という概念です。 HATEOAS のキーポイントは、リソース・データが API によって提供されるときには、 関連する情報を一群のハイパーリンクによって返すべきである、ということです。

あなたのリソース・クラスは、`yii\web\Linkable` インタフェイスを実装することによって、HATEOAS をサポートすることが出来ます。 このインタフェイスは、リンク のリストを返すべき `getLinks()` メソッド一つだけを含みます。 典型的には、少なくとも、リソース・オブジェクトそのものへの URL を表現する `self` リンクを返さなければなりません。例えば、

```
use yii\base\Model;
use yii\web\Link; // JSON ハイパーメディア API 言語に定義されているリンク・オブジェクトを表す
use yii\web\Linkable;
use yii\helpers\Url;

class UserResource extends Model implements Linkable
{
    public $id;
    public $email;

    //...

    public function fields()
    {
        return ['id', 'email'];
    }

    public function extraFields()
    {
        return ['profile'];
    }

    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id], true),
            'edit' => Url::to(['user/view', 'id' => $this->id], true),
            'profile' => Url::to(['user/profile/view', 'id' => $this->id], true),
            'index' => Url::to(['users'], true),
        ];
    }
}
```

`UserResource` オブジェクトがレスポンスで返されるとき、レスポンスはそのユーザに関連するリンクを表現する `_links` 要素を含むことになります。 例えば、

```
{
    "id": 100,
    "email": "user@example.com",
    // ...
    "_links" => {
        "self": {
            "href": "https://example.com/users/100"
        },
        "edit": {
            "href": "https://example.com/users/100"
        },
        "profile": {
            "href": "https://example.com/users/profile/100"
        },
        "index": {
            "href": "https://example.com/users"
        }
    }
}
```

11.2.3 コレクション

リソース・オブジェクトは コレクション としてグループ化することが出来ます。 各コレクションは、同じ型のリソースのリストを含みます。

コレクションは配列として表現することも可能ですが、通常は、データ・プロバイダ として表現する方がより望ましい方法です。 これは、データ・プロバイダがリソースの並べ替えとページネーションをサポートしているからです。 並べ替えとページネーションは、コレクションを返す RESTful API にとっては、普通に必要とされる機能です。 例えば、次のアクションは投稿のリソースについてデータ・プロバイダを返すものです。

```
namespace app\controllers;

use yii\rest\Controller;
use yii\data\ActiveDataProvider;
use app\models\Post;

class PostController extends Controller
{
    public function actionIndex()
    {
        return new ActiveDataProvider([
            'query' => Post::find(),
        ]);
    }
}
```

データ・プロバイダが RESTful API のレスポンスで送信される場合は、`yii\rest\Serializer` が現在のページのリソースを取り出して、リソース・オブジェクトの配列としてシリアライズします。 それだけでな

く、yii\rest\Serializer は次の HTTP ヘッダを使ってページネーション情報もレスポンスに含めます。

- X-Pagination-Total-Count: リソースの総数
- X-Pagination-Page-Count: ページ数
- X-Pagination-Current-Page: 現在のページ (1 から始まる)
- X-Pagination-Per-Page: 各ページのリソース数
- Link: クライアントがリソースをページごとにたどることが出来るようにするための一群のナビゲーションリンク

REST API におけるコレクションはデータ・プロバイダであるため、データ・プロバイダの全ての機能、すなわち、ページネーションやソーティングを共有しています。

その一例を クイック・スタート のセクションで見ることが出来ます。

コレクションをフィルタリングする

バージョン 2.0.13 以降、Yii はコレクションをフィルタリングする便利な機能を提供しています。 その一例を クイック・スタート のガイドに見ることが出来ます。 エンド・ポイントをあなた自身が実装しようとしている場合、フィルタリングは データ・プロバイダのガイドの [データ・フィルタを使ってデータ・プロバイダをフィルタリングする](output-data-providers.md#filtering-data-providers-using-data-filters のセクションで述べられている方法で行うことが出来ます。

11.3 コントローラ

リソース・クラスを作成して、リソース・データをどのようにフォーマットすべきかを指定したら、 次は、RESTful API を通じてエンド・ユーザにリソースを公開するコントローラ・アクションを作成します。

Yii は、RESTful アクションを作成する仕事を簡単にするための二つの基底コントローラ・クラスを提供しています。 すなわち、yii\rest\Controller と yii\rest\ActiveController です。 二つのコントローラの違いは、後者は アクティブ・レコード として表現されるリソースの扱いに特化した一連のアクションをデフォルトで提供する、という点にあります。 従って、あなたが アクティブ・レコード を使っていて、提供される組み込みのアクションに満足できるのであれば、 コントローラ・クラスを yii\rest\ActiveController から拡張することを検討すると良いでしょう。 そうすれば、最小限のコードで強力な RESTful API を作成することが出来ます。

yii\rest\Controller と yii\rest\ActiveController は、ともに、下記の機能を提供します。 これらのいくつかについては、後続のセクションで詳細に説明します。

- HTTP メソッドのバリデーション
- コンテント・ネゴシエーションとデータの書式設定

- 認証
- レート制限

yii\rest\ActiveController は次の機能を追加で提供します。

- 普通は必要とされる一連のアクション: index、view、create、update、delete、options
- リクエストされたアクションとリソースに対するユーザへの権限付与

11.3.1 コントローラ・クラスを作成する

新しいコントローラ・クラスを作成する場合、コントローラ・クラスの命名規約は、 リソースの型の名前を単数形で使う、というものです。 例えば、ユーザの情報を提供するコントローラは UserController と名付けることが出来ます。

新しいアクションを作成する仕方はウェブ・アプリケーションの場合とほぼ同じです。 唯一の違いは、render() メソッドを呼んでビューを使って結果を表示する代りに、RESTful アクションの場合はデータを直接に返す、という点です。 シリアライザ と レスポンス・オブジェクトが、 元のデータからリクエストされた形式への変換を処理します。 例えば、

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2 フィルタ

yii\rest\Controller によって提供される RESTful API 機能のほとんどは フィルタ の形で実装されています。 具体的に言うと、次のフィルタがリストされた順に従って実行されます。

- contentNegotiator: コンテント・ネゴシエーションをサポート。 レスポンス形式の設定 のセクションで説明します。
- verbFilter: HTTP メソッドのバリデーションをサポート。
- authenticator: ユーザ認証をサポート。 認証 のセクションで説明します。
- rateLimiter: レート制限をサポート。 レート制限 のセクションで説明します。

これらの名前付きのフィルタは、behaviors() メソッドで宣言されます。 このメソッドをオーバーライドして、個々のフィルタを構成したり、どれかを無効にしたり、あなた自身のフィルタを追加したりすることが出来ます。 例えば、HTTP 基本認証だけを使いたい場合は、次のようなコードを書くことが出来ます。

```
use yii\filters\auth\HttpBasicAuth;
```

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```

CORS

コントローラに CORS (クロス・オリジン・リソース共有) フィルタを追加するのは、上記の他のフィルタを追加するのより、若干複雑になります。 と言うのは、CORS フィルタは認証メソッドより前に適用されなければならないため、他のフィルタとは少し異なるアプローチが必要だからです。 また、ブラウザが認証クレデンシャルを送信する必要なく、リクエストが出来るかどうかを前もって安全に判断できるように、 CORS プリフライト・リクエスト[3] の認証を無効にする必要もあります。 下記のコードは、`yii\rest\ActiveController` を拡張した既存のコントローラに `yii\filters\Cors` フィルタを追加するのに必要なコードを示しています。

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();

    // 認証フィルタを削除する
    $auth = $behaviors['authenticator'];
    unset($behaviors['authenticator']);

    // CORS フィルタを追加する
    $behaviors['corsFilter'] = [
        'class' => \yii\filters\Cors::className(),
    ];

    // 認証フィルタを再度追加する
    $behaviors['authenticator'] = $auth;
    // CORS プリフライト・リクエスト (HTTP OPTIONS メソッド) の認証を回避する
    $behaviors['authenticator']['except'] = ['options'];

    return $behaviors;
}
```

[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS#Preflighted_requests

11.3.3 ActiveController を拡張する

コントローラを yii\rest\ActiveController から拡張する場合は、このコントローラを通じて提供しようとしているリソース・クラスの名前を modelClass プロパティにセットしなければなりません。 リソース・クラスは yii\db\ActiveRecord から拡張しなければなりません。

アクションをカスタマイズする

デフォルトでは、yii\rest\ActiveController は次のアクションを提供します。

- index: リソースをページごとにリストする。
- view: 指定されたリソースの詳細を返す。
- create: 新しいリソースを作成する。
- update: 既存のリソースを更新する。
- delete: 指定されたりソースを削除する。
- options: サポートされている HTTP メソッドを返す。

これらのアクションは全て actions() メソッドによって宣言されます。actions() メソッドをオーバーライドすることによって、これらのアクションを構成したり、そのいくつかを無効化したりすることが出来ます。例えば、

```
public function actions()
{
    $actions = parent::actions();

    // "delete" と "create" のアクションを無効にする
    unset($actions['delete'], $actions['create']);

    // データ・プロバイダの準備を "prepareDataProvider()" メソッドでカスタマイズする
    $actions['index']['prepareDataProvider'] = [$this, 'prepareDataProvider'];

    return $actions;
}

public function prepareDataProvider()
{
    // "index" アクションのためにデータ・プロバイダを準備して返す
}
```

どういう構成オプションが利用できるかを学ぶためには、個々のアクション・クラスのリファレンスを参照してください。

アクセス・チェックを実行する

RESTful API によってリソースを公開するときには、たいてい、 現在のユーザがリクエストしているリソースにアクセスしたり操作したりする

許可を持っているか否かをチェックする必要があります。 これは、yii\rest\ActiveController を使う場合は、checkAccess() メソッドを次のようにオーバーライドすることによって出来ます。

```
/**
 * 現在のユーザの特権をチェックする。
 *
 * 現在のユーザが指定されたデータ・モデルに対して指定されたアクションを実行する特権を
 * 有するか否かをチェックするためには、このメソッドをオーバーライドしなければなりません。
 * ユーザが権限をもたない場合は、[[ForbiddenHttpException]] が投げられなければなりません。
 *
 * @param string $action 実行されるアクションの 。ID
 * @param \yii\base\Model $model アクセスされるモデル。null の場合は、アクセスされる特定のモデルが無いことを意味する。
 * @param array $params 追加のパラメータ
 * @throws ForbiddenHttpException ユーザが権限をもたない場合
 */
public function checkAccess($action, $model = null, $params = [])
{
    // ユーザが $action と $model に対する権限を持つかどうかをチェック
    // アクセスを拒否すべきときは ForbiddenHttpException を投げる
    if ($action === 'update' || $action === 'delete') {
        if ($model->author_id !== \Yii::$app->user->id)
            throw new \yii\web\ForbiddenHttpException(sprintf('You can only %s articles that you\'ve created.', $action));
    }
}
```

checkAccess() メソッドは yii\rest\ActiveController のデフォルトのアクションから呼ばれます。 新しいアクションを作成して、それに対してもアクセス・チェックをしたい場合は、新しいアクションの中からこのメソッドを明示的に呼び出さなければなりません。

> ヒント: ロール・ベース・アクセス制御 (RBAC) コンポーネント を使って checkAccess() を実装することも可能です。

11.4 ルーティング

リソースとコントローラのクラスが準備できたら、通常のウェブ・アプリケーションと同じように、 http://localhost/index.php?r=user/create というような URL を使ってリソースにアクセスすることが出来ます。

実際には、綺麗な URL を有効にして HTTP 動詞を利用したいというのが普通でしょう。 例えば、POST /users というリクエストが user/create アクションへのアクセスを意味するようにする訳です。 これは、アプリケーションの構成情報で urlManager アプリケーション・コンポーネント を次のように構成することによって容易に達成することが出来ます。

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

ウェブ・アプリケーションの URL 管理と比べたときに、上記で目に付く新しいことは、 RESTful API リクエストのルーティングに `yii\rest\UrlRule` を使用していることです。 この特殊な URL 規則クラスが、一揃いの子 URL 規則を作成して、指定されたコントローラのルーティングと URL 生成をサポートします。 例えば、上記のコードは、おおむね下記の規則と等価です。

```
[
    'PUT,PATCH users/<id>' => 'user/update',
    'DELETE users/<id>' => 'user/delete',
    'GET,HEAD users/<id>' => 'user/view',
    'POST users' => 'user/create',
    'GET,HEAD users' => 'user/index',
    'users/<id>' => 'user/options',
    'users' => 'user/options',
]
```

そして、次の API エンド・ボイントがこの規則によってサポートされます。

- `GET /users`: 全てのユーザをページごとにリストする。
- `HEAD /users`: ユーザ一覧の概要を示す。
- `POST /users`: 新しいユーザを作成する。
- `GET /users/123`: ユーザ 123 の詳細を返す。
- `HEAD /users/123`: ユーザ 123 の概要情報を示す。
- `PATCH /users/123` と `PUT /users/123`: ユーザ 123 を更新する。
- `DELETE /users/123`: ユーザ 123 を削除する。
- `OPTIONS /users`: エンド・ボイント `/users` に関してサポートされる動詞を示す。
- `OPTIONS /users/123`: エンド・ボイント `/users/123` に関してサポートされる動詞を示す。

`only` および `except` オプションを構成すると、それぞれ、どのアクションをサポートするか、 また、どのアクションを無効にするかを明示的に指定することが出来ます。例えば、

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

また、`patterns` あるいは `extraPatterns` を構成して、既存のパターンを再定義したり、この規則によってサポートされる新しいパターンを追加し

たりすることも出来ます。 例えば、エンド・ボイント GET /users/search によって新しいアクション search をサポートするためには、extraPatterns オプションを次のように構成します。

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
]
```

エンド・ボイントの URL ではコントローラ ID user が users という複数形で出現していることに気が付いたかもしれません。 これは、yii\rest\UrlRule が子 URL 規則を作るときに、コントローラの ID を自動的に複数形にするためです。 この振舞いは yii\rest\UrlRule::$pluralize を false に設定することで無効にすることが出来ます。

> 情報: コントローラ ID の複数形化は yii\helpers\Inflector::pluralize() によって行われます。 このメソッドは特殊な複数形の規則を考慮します。例えば、box という単語の複数形は boxs ではなく boxes になります。

自動的な複数形化があなたの要求を満たさない場合は、yii\rest\UrlRule::$controller プロパティを構成して、 エンド・ボイント URL で使用される名前とコントローラ ID の対応を明示的に指定することも可能です。 例えば、次のコードはエンド・ボイント名 u をコントローラ ID user に割り当てます。

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

11.4.1 内蔵の規則に対する追加の構成

yii\rest\UrlRule の中に含まれるそれぞれの規則に対して適用される追加の構成を指定すると役に立つことがあります。 expand パラメータに対するデフォルトの指定が良い例です。

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['user'],
    'ruleConfig' => [
        'class' => 'yii\web\UrlRule',
        'defaults' => [
            'expand' => 'profile',
        ]
    ],
],
```

11.5 レスポンス形式の設定

RESTful API のリクエストを処理するとき、アプリケーションは、通常、レスポンス形式の設定に関して次のステップを踏みます。

1. レスポンス形式に影響するさまざまな要因、例えば、メディア・タイプ、言語、バージョンなどを決定します。 このプロセスは コンテント・ネゴシエーション[4] としても知られるものです。
2. リソース・オブジェクトを配列に変換します。 リソース のセクションで説明したように、この作業は `yii\rest\Serializer` によって実行されます。
3. 配列をコンテント・ネゴシエーションのステップで決定された形式の文字列に変換します。 この作業は、`response` アプリケーション・コンポーネント の `formatters` プロパティに登録された レスポンス・フォーマッタ によって実行されます。

11.5.1 コンテント・ネゴシエーション

Yii は `yii\filters\ContentNegotiator` フィルタによってコンテント・ネゴシエーションをサポートします。 RESTful API の基底コントローラ・クラス `yii\rest\Controller` は `contentNegotiator` という名前でこのフィルタを持っています。 このフィルタは、レスポンス形式のネゴシエーションと同時に言語のネゴシエーションも提供します。 例えば、RESTful API リクエストが下記のヘッダを含んでいるとします。

```
Accept: application/json; q=1.0, */*; q=0.1
```

この場合、リクエストは JSON 形式のレスポンスを受け取ることになります。例えば、次のような具合です。

```
$ curl -i -H "Accept: application/json; q=1.0, */*; q=0.1" "http://localhost
    /users"

HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8
```

[4] http://en.wikipedia.org/wiki/Content_negotiation

```
[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

舞台裏では、RESTful API コントローラ・アクションが実行される前に、yii\filters\ContentNegotiator フィルタがリクエストの Accept HTTP ヘッダをチェックして、レスポンス形式 を 'json' に設定します。 アクションが実行されて、その結果のリソースのオブジェクトまたはコレクションが返されると、yii\rest\Serializer が結果を配列に変換します。 そして最後に、yii\web\JsonResponseFormatter が配列を JSON 文字列に変換して、それをレスポンス・ボディに入れます。

デフォルトでは、RESTful API は JSON と XML の両方の形式をサポートします。 新しい形式をサポートするためには、下記のように、API コントローラ・クラスの中で contentNegotiator フィルタの formats プロパティを構成しなければなりません。

```
use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['contentNegotiator']['formats']['text/html'] = Response::
    FORMAT_HTML;
    return $behaviors;
}
```

formats プロパティのキーはサポートされる MIME タイプであり、値は対応するレスポンス形式名です。 このレスポンス形式名は、yii\web\Response::$formatters の中でサポートされているものでなければなりません。

11.5.2 データのシリアライズ

上記で説明したように、yii\rest\Serializer が、リソースのオブジェクトやコレクションを配列に変換する際に、中心的な役割を果たします。 Serializer は、yii\base\Arrayable および yii\data\DataProviderInterface のインタフェイスを実装したオブジェクトを認識します。 前者は主としてリソース・オブジェクトによって実装され、後者はリソース・コレクションによって実装されています。

yii\rest\Controller::$serializer プロパティに構成情報配列をセットしてシリアライザを構成することが出来ます。 例えば、場合に

よっては、クライアントの開発作業を単純化するために、ページネーション情報をレスポンス・ボディに直接に含ませたいことがあるでしょう。 そうするためには、yii\rest\Serializer::$collectionEnvelope プロパティを次のように構成します。

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

このようにすると、http://localhost/users というリクエストに対して、次のレスポンスを得ることが出来ます。

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "items": [
        {
            "id": 1,
            ...
        },
        {
            "id": 2,
            ...
        },
        ...
    ],
    "_links": {
        "self": {
            "href": "http://localhost/users?page=1"
        },
        "next": {
            "href": "http://localhost/users?page=2"
        },
        "last": {
            "href": "http://localhost/users?page=50"
```

```
        }
    },
    "_meta": {
        "totalCount": 1000,
        "pageCount": 50,
        "currentPage": 1,
        "perPage": 20
    }
}
```

JSON 出力を制御する

JSON 形式のレスポンスを生成する `JsonResponseFormatter` クラスは `JSON` ヘルパ を内部的に使用します。 このフォーマッタはさまざまなオプションによって構成することが可能です。 例えば、`$prettyPrint` オプションは、より読みやすいレスポンスのためのもので、開発時に有用なオプションです。 また、`$encodeOptions` によって JSON エンコーディングの出力を制御することが出来ます。

フォーマッタは、以下のように、アプリケーションの 構成情報 の中で、`response` アプリケーション・コンポーネントの `formatters` プロパティの中で構成することが出来ます。

```
'response' => [
    // ...
    'formatters' => [
        \yii\web\Response::FORMAT_JSON => [
            'class' => 'yii\web\JsonResponseFormatter',
            'prettyPrint' => YII_DEBUG, // デバッグ・モードでは きれい"" な出
    力を使用
            'encodeOptions' => JSON_UNESCAPED_SLASHES |
    JSON_UNESCAPED_UNICODE,
            // ...
        ],
    ],
],
```

DAO データベース・レイヤを使ってデータベースからデータを返す場合は、全てのデータが文字列として表されます。 しかし、特に数値は JSON では数として表現されなければなりませんので、これは必ずしも期待通りの結果であるとは言えません。 一方、ActiveRecord レイヤを使ってデータベースからデータを取得する場合は、数値カラムの値は、`yii\db\ActiveRecord::populateRecord()` においてデータベースからデータが取得される際に、整数に変換されます。

11.6 認証

ウェブ・アプリケーションとは異なり、RESTful API は通常はステート・レスです。 これは、セッションやクッキーは使用すべきでないこ

とを意味します。 従って、ユーザの認証ステータスをセッションやクッキーで保持することが出来ないため、全てのリクエストに何らかの認証情報を付加する必要があります。 通常使われるのは、ユーザを認証するための秘密のアクセス・トークンを全てのリクエストとともに送信する方法です。 アクセス・トークンはユーザを一意に特定して認証することが出来るものですので、 **API** リクエストは、中間者攻撃 **(man-in-the-middle attack)** を防止するために、常に **HTTPS** 経由で送信されなければなりません。

アクセス・トークンを送信するには、いくつかの異なる方法があります。

- HTTP Basic 認証[5]: アクセス・トークンはユーザ名として送信されます。 この方法は、アクセス・トークンを API コンシューマ側で安全に保存することが出来る場合、 例えば API コンシューマがサーバ上で走るプログラムである場合などにのみ使用されるべきです。
- クエリ・パラメータ: アクセス・トークンは API の URL、例えば、`https://example.com/users?access-token=xxxxxxxx` でクエリ・パラメータとして送信されます。 ほとんどのウェブ・サーバはクエリ・パラメータをサーバのログに記録するため、この手法は、 アクセス・トークンを HTTP ヘッダを使って送信することができない `JSONP` リクエストに応答するために主として使用されるべきです。
- OAuth 2[6]: OAuth2 プロトコルに従って、 アクセス・トークンはコンシューマによって権限付与サーバから取得され、HTTP Bearer Tokens[7] 経由で API サーバに送信されます。

Yii は上記の全ての認証方法をサポートしています。新しい認証方法を作成することも簡単に出来ます。

あなたの API に対して認証を有効にするためには、次のステップを実行します。

1. `user` アプリケーション・コンポーネント を構成します。
   - `enableSession` プロパティを `false` に設定します。
   - `loginUrl` プロパティを `null` に設定し、ログインページにリダイレクトする代りに HTTP 403 エラーを表示します。
2. REST コントローラ・クラスにおいて、`authenticator` ビヘイビアを構成することによって、 どの認証方法を使用するかを指定します。
3. ユーザ・アイデンティティ・クラス において `yii\web\IdentityInterface::findIdentityByAccessToken()` を実装します。

ステップ 1 は必須ではありませんが、ステート・レスであるべき RESTful API のために推奨されます。 `enableSession` が `false` である場合、

[5] http://ja.wikipedia.org/wiki/Basic%E8%AA%8D%E8%A8%BC
[6] http://oauth.net/2/
[7] http://tools.ietf.org/html/rfc6750

ユーザの認証ステータスがセッションを使ってリクエストをまたいで存続することはありません。 その代りに、すべてのリクエストに対して認証が実行されます。このことは、ステップ 2 と 3 によって達成されます。

> ヒント: RESTful API をアプリケーションの形式で開発する場合は、アプリケーションの構成情報で `user` アプリケーション・コンポーネント(structure-application-components.md) `enableSession` プロパティを構成することが出来ます。 RESTful API をモジュールとして開発する場合は、次のように、モジュールの `init()` メソッドに一行を追加することが出来ます。
>
> ```
> public function init()
> {
>     parent::init();
>     \Yii::$app->user->enableSession = false;
> }
> ```

例えば、HTTP Basic 認証を使う場合は、`authenticator` ビヘイビアを次のように構成することが出来ます。

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```

上で説明した三つの認証方法を全てサポートしたい場合は、次のように `CompositeAuth` を使うことが出来ます。

```
use yii\filters\auth\CompositeAuth;
use yii\filters\auth\HttpBasicAuth;
use yii\filters\auth\HttpBearerAuth;
use yii\filters\auth\QueryParamAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => CompositeAuth::className(),
        'authMethods' => [
            HttpBasicAuth::className(),
            HttpBearerAuth::className(),
            QueryParamAuth::className(),
        ],
    ];
    return $behaviors;
}
```

authMethods の各要素は、認証方法クラスの名前であるか、構成情報配列でなければなりません。

findIdentityByAccessToken() の実装はアプリケーション固有のものです。 例えば、各ユーザが一つだけアクセス・トークンを持ち得るような単純なシナリオでは、アクセス・トークンをユーザのテーブルの access_token カラムに保存することが出来ます。 そうすれば、次のように、findIdentityByAccessToken() メソッドを User クラスにおいて簡単に実装することが出来ます。

```
use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }
}
```

上記のように認証が有効化された後は、全ての API リクエストに対して、リクエストされたコントローラ beforeAction() の段階でユーザを認証することを試みます。

認証が成功すると、コントローラはその他のチェック (レート制限、権限付与など) をしてから、アクションを実行します。 認証されたユーザのアイデンティティは Yii::$app->user->identity によって取得することが出来ます。

認証が失敗したときは、HTTP ステータス 401 およびその他の適切なヘッダ (HTTP Basic 認証に対する WWW-Authenticate ヘッダなど) を持つレスポンスが送り返されます。

11.6.1 権限付与

ユーザが認証された後、おそらくは、リクエストされたリソースに対してリクエストされたアクションを実行する許可を 彼または彼女が持っているかどうかをチェックしたいでしょう。 権限付与 と呼ばれるこのプロセスについては、権限付与 のセクションで詳細に説明されています。

あなたのコントローラが yii\rest\ActiveController から拡張したものである場合は、 checkAccess() メソッドをオーバーライドして権限付与のチェックを実行することが出来ます。 このメソッドが yii\rest\ActiveController によって提供されている内蔵のアクションから呼び出されます。

11.7 レート制限

悪用を防止するために、あなたの API に レート制限 を加えることを検討すべきです。 例えば、各ユーザの API 使用を 10 分間で最大 100 回ま

での API 呼び出しに制限したいとしましょう。 ユーザから上記の期間内に多すぎるリクエストを受け取った場合は、ステータス・コード 429 (「リクエストが多すぎる」の意味) を持つレスポンスを返さなければなりません。

レート制限を可能にするためには、ユーザ・アイデンティティ・クラス で `yii\filters\RateLimitInterface` を実装しなければなりません。このインタフェイスは次の三つのメソッドを実装することを要求します。

- `getRateLimit()`: 許可されているリクエストの最大数と期間を返します (例えば、`[100, 600]` は 600 秒間に最大 100 回の API 呼び出しが出来ることを意味します)。
- `loadAllowance()`: 許可されているリクエストの残り数と、レート制限が最後にチェックされたときの対応する UNIX タイムスタンプを返します。
- `saveAllowance()`: 許可されているリクエストの残り数と現在の UNIX タイムスタンプの両方を保存します。

ユーザ・テーブルに二つのカラムを追加して、許容されているリクエスト数とタイムスタンプの情報を記録するのが良いでしょう。 それらを定義すれば、`loadAllowance()` と `saveAllowance()` は、 認証された現在のユーザに対応する二つのカラムの値を読み書きするものとして実装することが出来ます。 パフォーマンスを向上させるために、これらの情報をキャッシュや NoSQL ストレージに保存することを検討しても構いません。

`User` モデルにおける実装は次のようなものになります。

```
public function getRateLimit($request, $action)
{
    return [$this->rateLimit, 1]; // 秒間に1 $rateLimit 回のリクエスト
}

public function loadAllowance($request, $action)
{
    return [$this->allowance, $this->allowance_updated_at];
}

public function saveAllowance($request, $action, $allowance, $timestamp)
{
    $this->allowance = $allowance;
    $this->allowance_updated_at = $timestamp;
    $this->save();
}
```

アイデンティティのクラスに必要なインタフェイスを実装すると、Yii は `yii\rest\Controller` のアクション・フィルタとして構成された `yii\filters\RateLimiter` を使って、自動的にレート制限のチェックを行うようになります。 レート制限を超えると、レート・リミッタが `yii\web\TooManyRequestsHttpException` を投げます。

レート・リミッタは、REST コントローラ・クラスの中で、 次のよう

にして構成することが出来ます。

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;
    return $behaviors;
}
```

レート制限が有効にされると、デフォルトでは、送信される全てのレスポンスに、 現在のレート制限の情報を含む次の HTTP ヘッダが付加されます。

- `X-Rate-Limit-Limit` - 一定期間内に許可されるリクエストの最大数
- `X-Rate-Limit-Remaining` - 現在の期間において残っている許可されているリクエスト数
- `X-Rate-Limit-Reset` - 許可されているリクエストの最大数にリセットされるまで待たなければならない秒数

これらのヘッダは、上記のコード例で示されているように、`yii\filters\RateLimiter::$enableRateLimitHeaders` を `false` に設定することで無効にすることが出来ます。

11.8 バージョン管理

良い API はバージョン管理 されています。すなわち、一つのバージョンを絶え間なく変更するのではなく、変更と新機能は API の新しいバージョンにおいて実装されます。 クライアント・サイドとサーバ・サイドの両方のコードを完全に制御できるウェブ・アプリケーションとは違って、API はあなたの制御が及ばないクライアントによって使用されることを想定したものです。このため、API の後方互換性 (BC) は、可能な限り保たれなければなりません。 BC を損なうかも知れない変更が必要な場合は、それを API の新しいバージョンにおいて導入し、バージョン番号を上げるべきです。そうすれば、既存のクライアントは、API の古いけれども動作するバージョンを使い続けることが出来ますし、新しいまたはアップグレードされたクライアントは、新しい API バージョンで新しい機能を使うことが出来ます。

> ヒント: API のバージョン番号の設計に関する詳細情報は Semantic Versioning[8] を参照してください。

API のバージョン管理を実装する方法としてよく使われるのは、バージョン番号を API の URL に埋め込む方法です。 例えば、`http://example.com/v1/users` が API バージョン 1 の `/users` エンド・ポイントを指す、というものです。

[8] http://semver.org/

API のバージョン管理のもう一つの方法は、最近流行しているものですが、バージョン番号を HTTP リクエスト・ヘッダに付ける方法です。これは、典型的には、`Accept` ヘッダによって行われます。

```
// パラメータによって
Accept: application/json; version=v1
// ベンダーのコンテント・タイプによって
Accept: application/vnd.company.myapp-v1+json
```

どちらの方法にも長所と短所があり、それぞれのアプローチに対して多くの議論があります。 下記では、この二つの方法をミックスした API バージョン管理の実際的な戦略を紹介します。

- API 実装の各メジャー・バージョンを独立したモジュールに置き、モジュールの ID はメジャー・バージョン番号 (例えば `v1` や `v2`) とします。 当然ながら、API の URL はメジャー・バージョン番号を含むことになります。
- 各メジャー・バージョンの中では (従って対応するモジュールの中では) `Accept` HTTP リクエスト・ヘッダを使ってマイナー・バージョン番号を決定し、 マイナー・バージョンに応じたレスポンスのための条件分岐コードを書きます。

メジャー・バージョンを提供する各モジュールは、それぞれ、指定されたバージョンのためのリソースとコントローラのクラスを含んでいなければなりません。 コードの責任範囲をより良く分離するために、共通の基底のリソースとコントローラのクラスを保持して、 それをバージョンごとの個別のモジュールでサブ・クラス化することが出来ます。 サブ・クラスの中で、`Model::fields()` のような具体的なコードを実装します。

あなたのコードを次のように編成することが出来ます。

```
api/
    common/
        controllers/
            UserController.php
            PostController.php
        models/
            User.php
            Post.php
    modules/
        v1/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
            Module.php
        v2/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
```

```
                Post.php
            Module.php
```

アプリケーションの構成情報は次のようなものになります。

```
return [
    'modules' => [
        'v1' => [
            'class' => 'app\modules\v1\Module',
        ],
        'v2' => [
            'class' => 'app\modules\v2\Module',
        ],
    ],
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'enableStrictParsing' => true,
            'showScriptName' => false,
            'rules' => [
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v1/user',
    'v1/post']],
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v2/user',
    'v2/post']],
            ],
        ],
    ],
];
```

上記のコードの結果として、http://example.com/v1/users はバージョン 1 のユーザ一覧を返し、 http://example.com/v2/users はバージョン 2 のユーザ一覧を返すことになります。

モジュール化のおかげで、異なるメジャー・バージョンのためのコードを綺麗に分離することが出来ます。 しかし、モジュール化しても、共通の基底クラスやその他の共有リソースを通じて、モジュール間でコードを再利用することは引き続いて可能です。

マイナー・バージョン番号を扱うためには、contentNegotiator ビヘイビアによって提供されるコンテント・ネゴシエーションの機能を利用することが出来ます。 contentNegotiator ビヘイビアは、どのコンテント・タイプをサポートするかを決定するときに、 yii\web\Response::$acceptParams プロパティをセットします。

例えば、リクエストが HTTP ヘッダ Accept: application/json; version=v1 を伴って送信された場合、 コンテント・ネゴシエーションの後では、yii\web\Response::$acceptParams に ['version' => 'v1'] という値が含まれています。

acceptParams のバージョン情報に基づいて、アクション、リソース・クラス、シリアライザなどの個所で条件付きのコードを書いて、 適切な機能を提供することが出来ます。

マイナー・バージョンは、定義上、後方互換性を保つことを要求するものですので、コードの中でバージョンチェックをする個所はそれほど

多くないものと期待されます。 そうでない場合は、たいていは、新しいメジャー・バージョンを作成する必要がある、ということです。

11.9 エラー処理

RESTful API リクエストを処理していて、ユーザのリクエストにエラーがあったり、何か予期しないことがサーバ上で起ったりしたときには、単に例外を投げて、ユーザに何かがうまく行かなかったことを知らせることも出来ます。 しかし、エラーの原因 (例えば、リクエストされたリソースが存在しない、など) を特定することが出来るなら、 適切な HTTP ステータス・コード (例えば、404 ステータス・コードを表わす `yii\web\NotFoundHttpException`) と一緒に例外を投げることを検討すべきです。 そうすれば、Yii は対応する HTTP ステータスのコードとテキストをレスポンスとともに送信します。 Yii はまた、レスポンス・ボディにも、シリアライズされた表現形式の例外を含めます。例えば、

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

次のリストは、Yii の REST フレームワークで使われる HTTP ステータス・コードの要約です。

- 200: OK。すべて期待されたとおりに動作しました。
- 201: `POST` リクエストに対するレスポンスとしてリソースが成功裡に作成されました。 `Location` ヘッダが、新しく作成されたリソースを指し示す URL を含んでいます。
- 204: リクエストは成功裡に処理されましたが、レスポンスはボディ・コンテントを含んでいません (`DELTE` リクエストなどの場合)。
- 304: リソースは修正されていません。キャッシュしたバージョンを使うことが可能です。
- 400: 無効なリクエストです。これはユーザのさまざまな行為によって引き起こされます。 例えば、リクエストのボディに無効な JSON データを入れたり、無効なアクションパラメータを指定したり、など。
- 401: 認証が失敗しました。
- 403: 認証されたユーザは指定された API エンド・ボイントにアクセスすることを許可されていません。

- 404: リクエストされたリソースは存在しません。
- 405: メソッドが許可されていません。どの HTTP メソッドが許可されているか、`Allow` ヘッダをチェックしてください。
- 415: サポートされていないメディア・タイプです。リクエストされたコンテント・タイプまたはバージョン番号が無効です。
- 422: データの検証が失敗しました (例えば `POST` リクエストに対するレスポンスで)。レスポンス・ボディで詳細なエラー・メッセージをチェックしてください。
- 429: リクエストの数が多すぎます。レート制限のためにリクエストが拒絶されました。
- 500: 内部的サーバエラー。これは内部的なプログラムエラーによって生じ得ます。

11.9.1 エラー・レスポンスをカスタマイズする

場合によっては、デフォルトのエラー・レスポンス形式をカスタマイズしたいことがあるでしょう。 例えば、さまざまな HTTP ステータスを使ってさまざまなエラーを示すという方法によるのではなく、次に示すように、 HTTP ステータスとしては常に 200 を使い、実際の HTTP ステータス・コードはレスポンスの JSON 構造の一部として包み込む、という方式です。

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
    }
}
```

アプリケーションの構成情報で `response` コンポーネントの `beforeSend` イベントに応答することで、この目的を達することが出来ます。

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null && Yii::$app->request->get('
    suppress_response_code')) {
                    $response->data = [
```

```
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

上記のコードは、`suppress_response_code` が `GET` のパラメータとして渡された場合に、 レスポンスを (成功したものも、失敗したものも) 上記で説明したように再フォーマットします。

Chapter 12

開発ツール

Chapter 13

テスト

13.1 テスト

テストはソフトウェア開発の重要な部分です。 気付いているか否かにかかわらず、私たちは継続的にテストをしています。 例えば、PHP でクラスを書くとき、私たちはステップごとにデバッグしたり、または単純に `echo` 文や `die` 文を使ったりして、実装が最初の計画通りに動作することを検証します。 ウェブ・アプリケーションの場合は、何らかのテスト・データをフォームに入力して、ページが期待通りにユーザと相互作用をすることを確認します。

テストを実行するプロセスを自動化して、何かを検証する必要があるときは、いつでも、 それを代行してくれるコードを呼び出すだけでよいようにすることが出来ます。 結果が計画したものと合致することを検証するコードが テスト と呼ばれ、それを作成して更に実行するプロセスが テスト自動化 として知られています。 このテストの章の主題は、このテストの自動化です。

13.1.1 テストとともに開発する

テスト駆動開発 (TDD) とビヘイビア駆動開発 (BDD) のソフトウェア開発手法においては、実際のコードを書く前に、 コードの断片または全体の機能の振る舞いを一連のシナリオまたはテストとして記述します。 そして、その後で初めて、テストに合格するように実装を作成して、 意図された振る舞いが達成されていることを検証します。

一つの機能を開発するプロセスは以下のようになります。

- 実装されるべき機能を記述するテストを作成する。
- 新しいテストを走らせて、失敗することを確認する。まだ実装がないので、これは予期された結果です。
- 新しいテストに合格するための単純なコードを書く。
- 全てのテストを走らせて、全てが合格することを確認する。
- コードを改良して、それでも全てのテストが OK であることを確認する。

完了すれば、別の機能または改良のために、このプロセスを再び繰り返します。 既存の機能が変更される場合は、テストも変更されなければなりません。

> ヒント: 多数の小さくて単純なイテレーションを繰り返すために時間を取られていると感じる場合は、テスト・シナリオのカバー範囲を広くして、テストを再度実行するまでの作業量を増やしてみてください。 デバッグばかりやっている場合は、逆に範囲を狭めてみてください。

全ての実装作業の前にテストを作成する理由は、そうすれば、その後で、 達成したい事柄に集中して「どのようにするか」に没頭することが出来るからです。 通常、そのようにすることは、良い抽象化、機能修正時の容易なテスト保守、また、結合度の低いコンポーネントにつながります。

ですから、このような手法の長所を要約すると次のようになります。

- 一時に一つの事柄に集中できるため、計画と実装がより良いものになる。
- より多くの機能をより詳細にテストでカバーできる。つまり、テストが OK なら何も問題がないと期待できる。

通常は、長い期間で見れば、かなり時間を節約する効果があります。

13.1.2 いつ、どうやって、テストするか

上記で説明したテスト・ファーストの手法は長期間にわたる比較的複雑なプロジェクトには合理的なものですが、簡単なプロジェクトでは、やりすぎとなるおそれもあります。 この手法が適切であることを示す指標がいくつかあります。

- プロジェクトは既に大きくて複雑である。
- プロジェクトの要求仕様が複雑になってきている。プロジェクトが継続的に大きくなっている。
- プロジェクトが長期にわたる予定である。
- 失敗のコストが高すぎる。

既存の実装の振る舞いをカバーするテストを作成することは、何も悪いことではありません。

- プロジェクトはレガシーなものであるが、段階的に刷新される予定である。
- 従事すべきプロジェクトを得たが、それにはテストがなかった。

どんな形式の自動化テストもやりすぎになる、という場合もあるでしょう。

- プロジェクトは単純で、この先も、複雑になる心配はない。
- これ以上かかわることはない一度限りのプロジェクトである。

ただ、このような場合であっても、時間に余裕があれば、テストを自動化することは良いことです。

13.1.3 参考

- Test Driven Development: By Example / Kent Beck. ISBN: 0321146530.

13.2 テスト環境の構築

Yii 2 は `Codeception`[1] テスト・フレームワークとの統合を公式にサポートしており、 次のタイプのテストを作成することを可能にしています。

- 単体テスト - 一かたまりのコードが期待通りに動くことを検証する。
- 機能テスト - ブラウザのエミュレーションによって、ユーザの視点からシナリオを検証する。
- 受入テスト - ブラウザの中で、ユーザの視点からシナリオを検証する。

これら三つのタイプのテスト全てについて、Yii は、`yii2-basic`[2] と `yii2-advanced`[3] の両方のプロジェクト・テンプレートで、 そのまま使えるテストセットを提供しています。

ベーシック・テンプレート、アドバンスト・テンプレートの両方とも、Codeception がプリ・インストールされて付いて来ます。 これらのテンプレートの一つを使っていない場合は、下記のコンソールコマンドを発行することで Codeception をインストールすることが出来ます。

```
composer require codeception/codeception
composer require codeception/specify
composer require codeception/verify
```

13.3 単体テスト

単体テストは、一かたまりのコードが期待通りに動作することを検証するものです。 すなわち、さまざまな入力パラメータを与えて、クラスのメソッドが期待通りの結果を返すかどうかを検証します。 単体テストは、通常は、テストされるクラスを書く人によって開発されます。

Yii における単体テストは、PHPUnit と Codeception (こちらはオプションです) の上に構築されます。従って、それらのドキュメントを通読することが推奨されます。

- Codeception for Yii framework[4]
- Codeception Unit Tests[5]
- PHPUnit のドキュメントの第2章以降[6].

[1] https://github.com/Codeception/Codeception
[2] https://github.com/yiisoft/yii2-app-basic
[3] https://github.com/yiisoft/yii2-app-advanced
[4] http://codeception.com/for/yii
[5] http://codeception.com/docs/05-UnitTests
[6] http://phpunit.de/manual/current/en/writing-tests-for-phpunit.html

13.3.1 ベーシック・テンプレート、アドバンスト・テンプレートのテストを実行する

アドバンスト・テンプレートでプロジェクトを開始した場合、テストの実行については、“テスト” のガイド[7] を参照して下さい。

ベーシック・テンプレートでプロジェクトを開始した場合は、README の “testing” のセクション[8] を参照して下さい。

13.3.2 フレームワークの単体テスト

Yii フレームワーク自体に対する単体テストを走らせたい場合は、”Yii 2 の開発を始めよう[9]“ の説明に従ってください。

13.4 機能テスト

機能テストはユーザの視点からシナリオを検証するものです。 受入テスト と似ていますが、HTTP によって通信する代りに、 POST や GET のパラメータなどの環境変数を設定しておいてから、アプリケーションのインスタンスをコードから直接に実行します。

機能テストは一般的に受入テストより高速であり、失敗した場合に詳細なスタックトレースを提供してくれます。 経験則から言うと、特別なウェブ・サーバ設定や JavaScript による複雑な UI を持たない場合は、機能テストの方を選ぶべきです。

機能テストは Codeception フレームワークの助けを借りて実装されています。これにつては、優れたドキュメントがあります。

- Codeception for Yii framework[10]
- Codeception Functional Tests[11]

13.4.1 ベーシック・テンプレート、アドバンスト・テンプレートのテストを実行する

アドバンスト・テンプレートで開発をしている場合は、テスト実行の詳細について、“テスト” のガイド[12] を参照して下さい。

ベーシック・テンプレートで開発をしている場合は、README の “testing” のセクション[13] を参照して下さい。

---

[7] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/start-testing.md

[8] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

[9] https://github.com/yiisoft/yii2/blob/master/docs/internals-ja/getting-started.md

[10] http://codeception.com/for/yii

[11] http://codeception.com/docs/04-FunctionalTests

[12] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/start-testing.md

[13] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

13.5 受入テスト

受入テストはユーザの視点からシナリオを検証するものです。テストされるアプリケーションは PhpBrowser または実際のブラウザによってアクセスされます。 どちらの場合でも、ブラウザは HTTP によって通信しますので、アプリケーションはウェブ・サーバによってホストされる必要があります。

受入テストは Codeception フレームワークの助けを借りて実装されています。Codeception フレームワークには優れたドキュメントがありますので、参照して下さい。

- Codeception for Yii framework[14]
- Codeception Acceptance Tests[15]

13.5.1 ベーシック・テンプレート、アドバンスト・テンプレートのテストを実行する

アドバンスト・テンプレートで開発をしている場合は、テスト実行の詳細について、“テスト” のガイド[16] を参照して下さい。

ベーシック・テンプレートで開発をしている場合は、README の “testing” のセクション[17] を参照して下さい。

13.6 フィクスチャ

フィクスチャはテストの重要な部分です。 フィクスチャの主な目的は、テストを期待されている方法で繰り返して実行できるように、環境を固定された既知の状態に設定することです。 Yii は、Codeception でテストを実行する場合でも、単独でテストを実行する場合でも、 フィクスチャを正確に定義して容易に使うことが出来るように、フィクスチャ・フレームワークを提供しています。

Yii のフィクスチャ・フレームワークにおける鍵となる概念は、いわゆる フィクスチャ・オブジェクト です。 フィクスチャ・オブジェクトはテスト環境のある特定の側面を表現するもので、`yii\test\Fixture` またはその子クラスのインスタンスです。 例えば、ユーザの DB テーブルが固定されたデータセットを含むことを保証するために `UserFixture` を使う、という具合です。 テストを実行する前に一つまたは複数のフィクスチャ・オブジェクトをロードし、テストの完了時にアンロードします。

フィクスチャは他のフィクスチャに依存する場合があります。依存は `yii\test\Fixture::$depends` プロパティによって定義されます。 フィクスチャがロードされるとき、依存するフィクスチャはそのフィクス

[14] http://codeception.com/for/yii
[15] http://codeception.com/docs/03-AcceptanceTests
[16] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/start-testing.md
[17] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

チャの前に自動的にロードされます。 そしてフィクスチャがアンロードされるときには、依存するフィクスチャはそのフィクスチャの後にアンロードされます。

13.6.1 フィクスチャを定義する

フィクスチャを定義するためには、yii\test\Fixture または yii\test\ActiveFixture を拡張して新しいクラスを作ります。 前者は汎用目的のフィクスチャに最も適しています。 一方、後者はデータベースとアクティブ・レコードを扱うために専用に設計された拡張機能を持っています。

次のコードは、User アクティブ・レコードとそれに対応するテーブルに関して、フィクスチャを定義するものです。

```
<?php
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture
{
    public $modelClass = 'app\models\User';
}
```

> ヒント: すべての ActiveFixture は、テストの目的のために DB テーブルを準備するものです。 yii\test\ActiveFixture::$tableName プロパティまたは yii\test\ActiveFixture::$modelClass プロパティを設定することによって、テーブルを指定することが出来ます。 後者を使う場合は、modelClass によって指定される ActiveRecord クラスからテーブル名が取得されます。
>
> 補足: yii\test\ActiveFixture は SQL データベースにのみ適しています。 NoSQL データベースのためには、Yii は以下の ActiveFixture クラスを提供しています。
>
> - Mongo DB: yii\mongodb\ActiveFixture
> - Elasticsearch: yii\elasticsearch\ActiveFixture (バージョン 2.0.2 以降)

ActiveFixture フィクスチャのフィクスチャ・データは通常は FixturePath/data/TableName.php として配置されるファイルで提供されます。 ここで FixturePath はフィクスチャ・クラス・ファイルを含むディレクトリを意味し、TableName はフィクスチャと関連付けられているテーブルの名前です。 上記の例では、ファイルは @app/tests/fixtures/data/user.php となります。 データ・ファイルは、ユーザのテーブルに挿入されるデータ行の配列を返さなければなりません。 例えば、

```
<?php
return [
    'user1' => [
        'username' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/
    iK0r3jRuwQEs2ldRu.a2',
    ],
    'user2' => [
        'username' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6
    viYG5xJExU6',
    ],
];
```

データ行にはエイリアスを付けることが出来て、後でテストのときにエイリアスを使って行を参照することが出来ます。 上の例では、二つの行はそれぞれ `user1` および `user2` というエイリアスを付けられています。

また、オート・インクリメントのカラムに対してはデータを指定する必要はありません。 フィクスチャがロードされるときに Yii が自動的に実際の値を行に入れます。

> ヒント: `yii\test\ActiveFixture::$dataFile` プロパティを設定して、データ・ファイルの所在をカスタマイズすることが出来ます。 `yii\test\ActiveFixture::getData()` をオーバーライドしてデータを提供することも可能です。

前に説明したように、フィクスチャは別のフィクスチャに依存する場合があります。 例えば、ユーザ・プロファイルのテーブルはユーザのテーブルを指す外部キーを含んでいるため、`UserProfileFixture` は `UserFixture` に依存します。 依存関係は、次のように、`yii\test\Fixture::$depends` プロパティによって指定されます。

```
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserProfileFixture extends ActiveFixture
{
    public $modelClass = 'app\models\UserProfile';
    public $depends = ['app\tests\fixtures\UserFixture'];
}
```

依存関係は、また、複数のフィクスチャが正しく定義された順序でロードされ、アンロードされることを保証します。 上記の例では、全ての外部キー参照が存在することを保証するために `UserFixture` は常に `UserProfileFixture` の前にロードされます。 また、同じ理由によっ

て、UserFixture は常に UserProfileFixture がアンロードされた後でアンロードされます。

上記では、DB テーブルに関してフィクスチャを定義する方法を示しました。 DB と関係しないフィクスチャ (例えば、何らかのファイルやディレクトリに関するフィクスチャ) を定義するためには、より汎用的な基底クラス yii\test\Fixture から拡張して、 load() と unload() のメソッドをオーバーライドすることが出来ます。

13.6.2 フィクスチャを使用する

Codeception[18] を使ってコードをテストしている場合は、フィクスチャのローディングとアクセスについては、 内蔵されているサポートを使用することが出来ます。

その他のテスト・フレームワークを使っている場合は、テスト・ケースで yii\test\FixtureTrait を使って同じ目的を達することが出来ます。

次に、Codeception を使って UserProfile 単体テストクラスを書く方法を説明します。

\Codeception\Test\Unit を拡張するあなたの単体テスト・クラスにおいて、 _fixtures() メソッドの中で使いたいフィクスチャを宣言するか、 または、アクターの haveFixtures() メソッドを直接に使用します。例えば、

```
namespace app\tests\unit\models;


use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends \Codeception\Test\Unit
{
    public function _fixtures()
    {
        return [
            'profiles' => [
                'class' => UserProfileFixture::className(),
                // フィクスチャ・データは tests/_data/user.php に配置されてい
    る
                'dataFile' => codecept_data_dir() . 'user.php'
            ],
        ];
    }

    // ... テストのメソッド ...
}
```

_fixtures() メソッドにリストされたフィクスチャは、テストが実行される前に自動的にロードされます。 前に説明したように、フィクスチャがロードされるときには、それが依存するフィクスチャのすべてが自動的

[18] http://codeception.com/

に先にロードされます。 上の例では、UserProfileFixture は UserFixture に依存しているので、テスト・クラスのどのテスト・メソッドを走らせるときでも、二つのフィクスチャが連続してロードされます。 すなわち、最初に UserFixture がロードされ、次に UserProfileFixture がロードされます。

_fixtures() でフィクスチャを指定するときも、haveFixtures() でフィクスチャを指定するときも、 クラス名あるいはフィクスチャを指す構成情報配列を使うことが出来ます。 構成情報配列を使うと、フィクスチャがロードされるときのフィクスチャのプロパティをカスタマイズすることが出来ます。

また、フィクスチャにエイリアスを割り当てることも出来ます。上記の例では、UserProfileFixture に profiles というエイリアスが与えられています。 そうすると、テスト・メソッドの中でエイリアスを使ってフィクスチャ・オブジェクトにアクセスすることが出来るようになります。例えば、

```
$profile = $I->grabFixture('profiles');
```

は UserProfileFixture オブジェクトを返します。

さらには、UserProfileFixture は ActiveFixture を拡張するものですので、フィクスチャによって提供されたデータに対して、 次の構文を使ってアクセスすることも出来ます。

```
// 'user1' というエイリアスのデータ行に対応する UserProfileModel を返す
$profile = $I->grabFixture('profiles', 'user1');
// フィクスチャにある全てのデータ行をたどる
foreach ($I->grabFixture('profiles') as $profile) ...
```

13.6.3 フィクスチャ・クラスとデータ・ファイルを編成する

デフォルトでは、フィクスチャ・クラスは対応するデータ・ファイルを探すときに、フィクスチャのクラス・ファイルを含むフォルダのサブ・フォルダである data フォルダの中を見ます。 簡単なプロジェクトではこの規約に従うことができます。 大きなプロジェクトでは、おそらくは、同じフィクスチャ・クラスを異なるテストに使うために、データ・ファイルを切り替える必要がある場合が頻繁に生じるでしょう。 従って、クラスの名前空間と同じように、データ・ファイルを階層的な方法で編成することを推奨します。 例えば、

```
# tests\unit\fixtures フォルダの下に

data\
    components\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
    models\
        fixture_data_file1.php
```

```
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
# 等々
```

このようにして、テスト間でフィクスチャのデータ・ファイルが衝突するのを回避し、必要に応じてデータ・ファイルを使い分けます。

> 補足: 上の例では、フィクスチャ・ファイルには例示目的だけの名前が付けられています。 実際の仕事では、フィクスチャ・クラスがどのフィクスチャ・クラスを拡張したものであるかに従って名前を付けるべきです。 例えば、DB フィクスチャを `yii\test\ActiveFixture` から拡張している場合は、DB テーブルの名前をフィクスチャのデータ・ファイル名として使うべきです。 MongoDB フィクスチャを `yii\mongodb\ActiveFixture` から拡張している場合は、コレクション名をファイル名として使うべきです。

同様な階層は、フィクスチャ・クラス・ファイルを編成するのにも使うことが出来ます。 `data` をルート・ディレクトリとして使うのでなく、データ・ファイルとの衝突を避けるために `fixtures` をルート・ディレクトリとして使うのが良いでしょう。

13.6.4 `yii fixture` でフィクスチャを管理する

Yii は `yii fixture` コマンドライン・ツールでフィクスチャをサポートしています。以下の機能をサポートしています。

- 異なるストレージ (RDBMS、NoSQL など) へのフィクスチャのロード
- 様々な方法でのフィクスチャのアンロード (通常はストレージをクリア)
- フィクスチャの自動生成およびランダム・データの投入

フィクスチャのデータ形式

次のようなフィクスチャ・データをロードするとしましょう。

```
# users.php ファイル - フィクスチャ・データ・パス デフォルトで
    は( @tests\unit\fixtures\data) に保存

return [
    [
        'name' => 'Chase',
        'login' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/
    iK0r3jRuwQEs2ldRu.a2',
    ],
```

```
    [
        'name' => 'Celestine',
        'login' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6
    viYG5xJExU6',
    ],
];
```

データベースにデータをロードするフィクチャを使う場合は、これらの行が `users` テーブルに対して適用されます。NoSQL フィクスチャ、例えば `mongodb` フィクチャを使う場合は、このデータは `users` コレクションに対して適用されます。 さまざまなロード戦略を実装する方法などについて公式 ドキュメント[19]を参照して下さい。 上記のフィクスチャのサンプルは `yii2-faker` エクステンションによって生成されました。これについての詳細は、自動生成のセクション を参照して下さい。 フィクスチャ・クラスの名前は複数形であってはいけません。

フィクスチャをロードする

フィクスチャ・クラスは `Fixture` という接尾辞を持たなければいけません。デフォルトでは、フィクスチャは `tests\unit\fixtures` 名前空間の下で探されます。 この挙動は構成またはコマンド・オプションによって変更することが出来ます。`-User` のように名前の前に `-` を指定することで、ロードまたはアンロードから除外するフィクスチャを指定することが出来ます。

フィクスチャをロードするためには、次のコマンドを実行します。

> 補足: データをロードする前に、アンロードのシーケンスが実行されます。これによって、通常は、前に実行されたフィクスチャによって挿入された 既存のデータが全てクリーンアップされることになります。

```
yii fixture/load <fixture_name>
```

要求される `fixture_name` パラメータが、データがロードされるフィクスチャの名前を指定するものです。 いくつかのフィクスチャを一度にロードすることが出来ます。下記はこのコマンドの正しい形式です。

```
// ‘User‘ フィクスチャをロードする
yii fixture/load User

// 上記と同じ、"fixture" コマンドのデフォルトのアクションは "load" であるため
yii fixture User

// いくつかのフィクスチャをロードする
```

[19] https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixtures.md

```
yii fixture "User, UserProfile"

// 全てのフィクスチャをロードする
yii fixture/load "*"

// 同上
yii fixture "*"

// 一つを除いて全てのフィクスチャをロードする
yii fixture "*, -DoNotLoadThisOne"

// 異なる名前空間からフィクスチャをロードする デフォルトの名前空間
    は( tests\unit\fixtures)
yii fixture User --namespace='alias\my\custom\namespace'

// 他のフィクスチャをロードする前に、グローバルフィクス
    チャ ‘some\name\space\CustomFixture‘ をロードする。
// デフォルトでは、このオプションが ‘InitDbFixture‘ について適用され、整合性
    チェックが無効化有効化されます。/
// カンマで区切って複数のグローバル・フィクスチャを指定することが出来ます。
yii fixture User --globalFixtures='some\name\space\Custom'
```

フィクスチャをアンロードする

フィクスチャをアンロードするためには、次のコマンドを実行します。

```
// Users フィクスチャをアンロードする。デフォルトではフィクスチャのストレージを
    クリアします。例えば、("users" テーブル、または、mongodb フィクスチャな
    ら "users" コレクションがクリアされます。)
yii fixture/unload User

// いくつかのフィクスチャをアンロードする
yii fixture/unload "User, UserProfile"

// すべてのフィクスチャをアンロードする
yii fixture/unload "*"

// 一つを除いて全てのフィクスチャをアンロードする
yii fixture/unload "*, -DoNotUnloadThisOne"
```

このコマンドでも、namespace や globalFixtures という同じオプションを適用することが出来ます。

コマンドをグローバルに構成する

コマンドライン・オプションはフィクスチャ・コマンドをその場で構成することを可能にするものですが、 コマンドを一度だけ構成して済ませたい場合もあります。 例えば、次のように、異なるフィクスチャのパスを構成することが出来ます。

```
'controllerMap' => [
```

```
    'fixture' => [
        'class' => 'yii\console\controllers\FixtureController',
        'namespace' => 'myalias\some\custom\namespace',
        'globalFixtures' => [
            'some\name\space\Foo',
            'other\name\space\Bar'
        ],
    ],
]
```

フィクスチャを自動生成する

Yii は、あなたの代りに、何らかのテンプレートに従ってフィクスチャを自動生成することが出来ます。さまざまなデータで、また、いろいろな言語と形式で、フィクスチャを生成することが出来ます。 この機能は、Faker[20] ライブラリと `yii2-faker` エクステンションによって実現されています。 詳細については、エクステンションの ガイド[21] を参照して下さい。.

13.6.5 まとめ

以上、フィクスチャを定義して使用する方法を説明しました。 下記に、DB に関連した単体テストを走らせる場合の典型的なワークフローをまとめておきます。

1. `yii migrate` ツールを使って、テストのデータベースを最新版にアップグレードする
2. テスト・ケースを走らせる
   - フィクスチャをロードする - 関係する DB テーブルをクリーンアップし、フィクスチャ・データを投入する
   - 実際のテストを実行する
   - フィクスチャをアンロードする
3. 全てのテストが完了するまで、ステップ 2 を繰り返す

[20] https://github.com/fzaninotto/Faker
[21] https://github.com/yiisoft/yii2-faker/tree/master/docs/guide-ja

Chapter 14

スペシャル・トピック

14.1 あなた自身のアプリケーション構造を作成する

補足: このセクションはまだ執筆中です。

ベーシック[1] と アドバンスト[2] のプロジェクト・テンプレートは、あなたの要求をほとんどカバーする優れたものですが、 あなたのプロジェクトを開始するためのあなた自身のテンプレートを作成したいこともあるでしょう。

Yii におけるプロジェクト・テンプレートは、`composer.json` ファイルを含み、Composer パッケージとして登録されたレポジトリであるに過ぎません。 どのようなレポジトリでも、Composer パッケージとして特定し、`create-project` Composer コマンドによってインストール可能なものにすることが出来ます。

テンプレート全体を最初から構築するのは少々大変ですので、内蔵のテンプレートの一つを基礎として使うのが良いでしょう。 ここでは、ベーシック・テンプレートを使いましょう。

14.1.1 ベーシック・テンプレートをクローンする

最初のステップは、ベーシック Yii テンプレートの Git レポジトリをクローンすることです。

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

そして、レポジトリがあなたのコンピュータにダウンロードされるのを待ちます。 テンプレートに加えられた変更がレポジトリにプッシュ・バックされることはありませんので、ダウンロードしたものから `.git` ディレクトリとその中身を全て削除して構いません。

[1] https://github.com/yiisoft/yii2-app-basic
[2] https://github.com/yiisoft/yii2-app-advanced

14.1.2 ファイルを修正する

次に、あなたのテンプレートに合うように `composer.json` を修正します。`name`、`description`、`keywords`、`homepage`、`license` および `support` の値を、あなたの新しいテンプレートを説明するものに変更します。 また、`require`、`require-dev`、`suggest` や、その他のオプションも、あなたのテンプレートの要求に合うように調整します。

> 補足: `composer.json` ファイルで、`extra` の下の `writable` パラメータを使って、 アプリケーションがテンプレートを使って作成された後に設定されるべきファイル単位のアクセス権限を指定してください。

次に、あなたが好むデフォルトの状態に合うように、アプリケーションの構造と内容を実際に修正します。 最後に、あなたのテンプレートに適用できるように、README ファイルを更新します。

14.1.3 パッケージを作る

テンプレートが定義できたら、それを基に Git レポジトリを作成して、ファイルをそこにプッシュします。 あなたのテンプレートをオープンソース化するつもりなら、レポジトリをホストするのには Github[3] が最適の場所です。 テンプレートを共同作業に使わないつもりであれば、どんな Git レポジトリサイトでも構いません。

次に、Composer のためにパッケージを登録する必要があります。パブリックなテンプレートであれば、パッケージは Packagist[4] に登録すべきです。 プライベートなテンプレートは、パッケージの登録が少々トリッキーです。 その説明については Composer ドキュメント[5] を参照してください。

14.1.4 テンプレートを使う

Yii の新しいプロジェクト・テンプレートを作成するのに必要なことは以上です。 これで、あなたのテンプレートを使ってプロジェクトを作成することが出来ます。

```
composer create-project --prefer-dist --stability=dev mysoft/yii2-app-
    coolone new-project
```

14.2 コンソール・アプリケーション

ウェブ・アプリケーションを構築するための豊富な機能に加えて、Yii はコンソール・アプリケーションのためのフル装備のサポートを持ってい

[3] http://github.com
[4] https://packagist.org/
[5] https://getcomposer.org/doc/05-repositories.md#hosting-your-own

ます。 コンソール・アプリケーションは、主として、ウェブ・サイトのために実行する必要のあるバックグラウンドのタスクやメンテナンスのタスクを作成するために使われるものです。

コンソール・アプリケーションの構造は Yii のウェブ・アプリケーションのそれと非常に良く似ています。 コンソール・アプリケーションは一つまたは複数の `yii\console\Controller` クラスから構成されます。コントローラはコンソール環境ではしばしば「コマンド」と呼ばれます。 また、各コントローラは、ウェブのコントローラと全く同じように、一つまたは複数のアクションを持つことが出来ます。

プロジェクト・テンプレートは、両方とも、既にコンソール・アプリケーションを持っています。 レポジトリのベース・ディレクトリにある `yii` スクリプトを呼び出すことによって、コンソール・アプリケーションを実行することが出来ます。 このスクリプトは、何もパラメータを追加せずに実行すると、利用できるコマンドの一覧を表示します。

```
{lamb} php yii

This is Yii version 2.0.8-dev.

The following commands are available:

- asset                       Allows you to combine and compress your JavaScript and CSS files.
    asset/compress (default)  Combines and compresses the asset files according to the given configuration.
    asset/template            Creates template of configuration file for [[actionCompress]].

- cache                       Allows you to flush cache.
    cache/flush               Flushes given cache components.
    cache/flush-all           Flushes all caches registered in the system.
    cache/flush-schema        Clears DB schema cache for a given connection component.
    cache/index (default)     Lists the caches that can be flushed.

- fixture                     Manages fixture data loading and unloading.
    fixture/load (default)    Loads the specified fixture data.
    fixture/unload            Unloads the specified fixtures.

- help                        Provides help information about console commands.
    help/index (default)      Displays available commands or the detailed information

- message                     Extracts messages to be translated from source files.
    message/config            Creates a configuration file for the "extract" command using command line options specified
    message/config-template   Creates a configuration file template for the "extract" command.
    message/extract (default) Extracts messages to be translated from source code.

- migrate                     Manages application migrations.
    migrate/create            Creates a new migration.
    migrate/down              Downgrades the application by reverting old migrations.
    migrate/history           Displays the migration history.
    migrate/mark              Modifies the migration history to the specified version.
    migrate/new               Displays the un-applied new migrations.
    migrate/redo              Redoes the last few migrations.
    migrate/to                Upgrades or downgrades till the specified version.
    migrate/up (default)      Upgrades the application by applying new migrations.

- serve                       Runs PHP built-in web server
    serve/index (default)     Runs PHP built-in web server


To see the help of each command, enter:

  yii help <command-name>
```

スクリーン・ショットに表示されているように、デフォルトで利用できる一連のコマンドが Yii によって既に定義されています。

- `AssetController` - JavaScript と CSS ファイルを結合して圧縮することが出来ます。 このコマンドについては、アセットのセクションでさらに学習することが出来ます。
- `CacheController` - アプリケーションのキャッシュをフラッシュすることが出来ます。
- `FixtureController` - テストのために、フィクスチャ・データのロードとアンロードを管理します。 このコマンドについては テストのフィクスチャのセクション で詳細に説明されています。

- HelpController - コンソール・コマンドについてのヘルプ情報を提供します。 これがデフォルトのコマンドであり、上のスクリーン・ショットで見た出力を表示するものです。
- MessageController - ソース・ファイルから翻訳すべきメッセージを抽出します。 このコマンドについてさらに学習するためには、国際化のセクション を参照してください。
- MigrateController - アプリケーションのマイグレーションを管理します。 データベースのマイグレーションについては、データベースのマイグレーションのセクション で詳しく説明されています。
- ServeController - PHP の内蔵ウェブ・サーバを走らせることが出来ます。

14.2.1 使用方法

コンソールのコントローラ・アクションは次の構文を使って実行します。

```
yii <route> [--option1=value1 ... argument1 argument2 ... --option2=value2]
```

オプションはどの位置で指定しても構いません。

上記において、<route> はコントローラ・アクションへのルートを示すものです。 オプション (options) はクラスのプロパティに代入され、引数 (arguments) はアクション・メソッドのパラメータとなります。

例えば、MigrateController::$migrationTable として migrations を指定し、 マイグレーションの上限を 5 と指定して MigrateController::actionUp() を呼び出すためには、次のようにします。

```
yii migrate/up 5 --migrationTable=migrations
```

> 補足: コンソールで * を使う場合は、"*" として引用符号で囲むことを忘れないでください。 これは、* をカレント・ディレクトリの全てのファイル名に置き換えられるシェルのグロブとして実行してしまうことを避けるためです。

14.2.2 エントリ・スクリプト

コンソール・アプリケーションのエントリ・スクリプトは、ウェブ・アプリケーションで使用されるブートストラップ・ファイル index.php に相当するものです。 コンソールのエントリ・スクリプトは通常は yii と呼ばれるもので、アプリケーションのルート・ディレクトリに配置されています。 それは次のようなコードを含んでいます。

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */
```

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

require __DIR__ . '/vendor/autoload.php';
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';

$config = require __DIR__ . '/config/console.php';

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

このスクリプトはアプリケーションの一部として生成されるものです。あなたの必要を満たすように、自由に編集して構いません。 エラー発生時にスタック・トレースを見たくない、または、全体のパフォーマンスを上げたい、という場合は、YII_DEBUG 定数を false に設定することが出来ます。 ベーシック・プロジェクト・テンプレートでも、アドバンスト・プロジェクト・テンプレートでも、コンソール・アプリケーションのエントリ・スクリプトは、開発者に優しい環境を提供するために、デフォルトでデバッグを有効にしています。

14.2.3 構成情報

上記のコードで見るように、コンソール・アプリケーションは、console.php という名前のそれ自身の構成情報ファイルを使用します。 このファイルの中で、さまざまな アプリケーション・コンポーネント、取り分け、コンソール・アプリケーションのためのプロパティを構成しなければなりません。

ウェブ・アプリケーションとコンソール・アプリケーションが構成情報のパラメータと値を数多く共有する場合は、共通の部分を独立したファイルに移動して、 そのファイルを両方のアプリケーション (ウェブとコンソール) の構成情報にインクルードすることを検討しても良いでしょう。 その例をアドバンスト・プロジェクト・テンプレートの中で見ることが出来ます。

> ヒント: 場合によっては、エントリ・スクリプトで指定されているのとは異なるアプリケーション構成情報を使って コンソール・コマンドを実行したいことがあります。 例えば、yii migrate コマンドを使ってテストのデータベースをアップグレードするとき、 データベースが個々のテストスイートの中で構成されているような場合です。 構成情報を動的に変更するためには、コマンドを実行するときに appconfig オプションを使ってカスタムの構成情報ファイルを指定するだけで大丈夫です。
>
> ```
> yii <route> --appconfig=path/to/config.php ...
> ```

14.2.4 コンソール・コマンドの補完

シェルで作業をしている場合、コマンド引数の自動補完は便利なものです。 2.0.11 以降、./yii コマンドは、内蔵で Bash および ZSH のために補完をサポートしています。

Bash の補完

bash completion がインストールされていることを確認して下さい。ほとんどの bash のインストレーションでは、デフォルトで利用可能になっています。

補完スクリプトを /etc/bash_completion.d/ に置いて下さい。

```
curl -L https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/
    completion/bash/yii -o /etc/bash_completion.d/yii
```

一時的な利用の場合は、ファイルをカレント・ディレクトリに置いて、source yii でカレント・セッションに読み込みます。 グローバルにインストールした場合は、ターミナルを再起動するか、source ~/.bashrc を実行して、有効化する必要があります。

あなたの環境で補完スクリプトを読み込む他の方法については、Bash マニュアル[6] を参照して下さい。

ZSH の補完

補完のためのディレクトリ、例えば ~/.zsh/completion/ に補完スクリプトを置いて下さい。

```
mkdir -p ~/.zsh/completion
curl -L https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/
    completion/zsh/_yii -o ~/.zsh/completion/_yii
```

そのディレクトリを $fpath に追加します。例えば ~/.zshrc に次の記述を追加します。

```
fpath=(~/.zsh/completion $fpath)
```

compinit がロードされていることを確認して下さい。そうでなければ、~/.zshrc の中でロードします。

```
autoload -Uz compinit && compinit -i
```

そしてシェルをリロードします。

```
exec $SHELL -l
```

[6] https://www.gnu.org/software/bash/manual/html_node/Programmable-Completion.html

14.2.5 あなた自身のコンソール・コマンドを作成する

コンソールのコントローラとアクション

コンソール・コマンドは、yii\console\Controller を拡張するコントローラ・クラスとして定義することが出来ます。 コントローラ・クラスの中で、コントローラのサブ・コマンドに対応する一つまたは複数のアクションを定義します。各アクションの中で、その特定のサブ・コマンドのための適切なタスクを実装するコードを書きます。

コマンドを実行するときは、コントローラのアクションに対するルートを指定する必要があります。 例えば、ルート migrate/create は、MigrateController::actionCreate() アクション・メソッドに対応するサブコマンドを呼び出します。 実行時に提供されたルートにアクション ID が含まれない場合は、(ウェブのコントローラの場合と同じように) デフォルトのアクションが実行されます。

オプション

yii\console\Controller::options() メソッドをオーバーライドすることによって、コンソール・コマンド (controller/actionID) で利用できるオプションを指定することが出来ます。 このメソッドはコントローラ・クラスのパブリックなプロパティのリストを返さなければなりません。 コマンドを実行するときは、--OptionName=OptionValue という構文を使ってオプションの値を指定することが出来ます。 これはコントローラ・クラスの OptionName プロパティに OptionValue を割り当てるものです。

オプションのデフォルト値が配列型である場合、実行時にこのオプションをセットすると、 オプションの値は、入力文字列をカンマで分離することによって、配列に変換されます。

オプションのエイリアス

バージョン 2.0.8 以降、コンソールコマンドは、オプションにエイリアスを追加するための yii\console\Controller::optionAliases() メソッドを提供しています。

エイリアスを定義するためには、コントローラで yii\console\Controller::optionAliases() をオーバーライドします。例えば、

```
namespace app\commands;

use yii\console\Controller;

class HelloController extends Controller
{
    public $message;

    public function options($actionID)
    {
        return ['message'];
```

```
    }

    public function optionAliases()
    {
        return ['m' => 'message'];
    }

    public function actionIndex()
    {
        echo $this->message . "\n";
    }
}
```

これで、次の構文を使ってコマンドを走らせることが出来るようになります。

```
./yii hello -m=hello
```

引数

オプションに加えてに、コマンドは引数を取ることも出来ます。引数は、リクエストされたサブ・コマンドに対応するアクション・メソッドへのパラメータとして渡されます。 最初の引数は最初のパラメータに対応し、二番目の引数は二番目のパラメータに対応し、以下同様です。コマンドが呼び出されたときに十分な数の引数が提供されなかったときは、対応するパラメータは、定義されていれば、宣言されているデフォルト値をとります。 デフォルト値が設定されておらず、実行時に値が提供されなかった場合は、コマンドはエラーで終了します。

`array` タイプ・ヒントを使って、引数が配列として扱われるべきことを示すことが出来ます。 配列は入力文字列をカンマで分割することによって生成されます。

次に引数を宣言する方法を示す例を挙げます。

```
class ExampleController extends \yii\console\Controller
{
    // コマンド "yii example/create test" は "actionCreate('test')" を呼び出す
    public function actionCreate($name) { ... }

    // コマンド "yii example/index city" は "actionIndex('city', 'name')" を呼び出す
    // コマンド "yii example/index city id" は call "actionIndex('city', 'id')" を呼び出す
    public function actionIndex($category, $order = 'name') { ... }

    // コマンド "yii example/add test" は "actionAdd(['test'])" を呼び出す
    // コマンド "yii example/add test1,test2" は "actionAdd(['test1', 'test2'])" を呼び出す
```

```
    public function actionAdd(array $name) { ... }
}
```

終了コード

終了コードを使うことはコンソール・アプリケーション開発のベスト・プラクティスです。 コマンドは何も問題が無かったことを示すために 0 を返すのが慣例です。コマンドが 1 以上の値を返したときは、何かエラーを示唆するものとみなされます。 返される数値がエラーコードであり、それによってエラーに関する詳細を見出すことが出来る場合もあります。 例えば、1 は一般的な未知のエラーを示すものとし、2 以上の全てのコードは特定のエラー、例えば、入力エラー、ファイルが見つからない、等々を示すものとすることが出来ます。

コンソール・コマンドに終了コードを返させるためには、 単にコントローラのアクション・メソッドで整数を返すようにします。

```
public function actionIndex()
{
    if (/* 何らかの問題が発生 */) {
        echo "A problem occurred!\n";
        return 1;
    }
    // 何かをする
    return 0;
}
```

いくつか使用できる事前定義された定数があります。それらは `yii\console\ExitCode` クラスで定義されています。

```
public function actionIndex()
{
    if (/* 何らかの問題が発生 */) {
        echo "A problem occurred!\n";
        return ExitCode::UNSPECIFIED_ERROR;
    }
    // 何かをする
    return ExitCode::OK;
}
```

もっと詳細なエラー・コードを必要とする場合は、コントローラで詳細な定数を定義するのが良いプラクティスです。

書式設定と色

Yii のコンソール・コマンドは出力の書式設定をサポートしています。 これは、コマンドを走らせている端末がサポートしていない場合は、自動的に書式設定の無い出力にグレードダウンされます。

書式設定された文字列を出力することは簡単です。ボールドのテキストを出力するには、次のようにします。

```
$this->stdout("Hello?\n", Console::BOLD);
```

複数のスタイルを動的に結合して文字列を構成する必要がある場合は、ansiFormat() を使うほうが良いでしょう。

```
$name = $this->ansiFormat('Alex', Console::FG_YELLOW);
echo "Hello, my name is $name.";
```

表形式

バージョン 2.0.13 以降、表形式のデータをコンソールに表示するウィジェットが追加されています。次のようにして使うことが出来ます。

```
echo Table::widget([
    'headers' => ['Project', 'Status', 'Participant'],
    'rows' => [
        ['Yii', 'OK', '@samdark'],
        ['Yii', 'OK', '@cebe'],
    ],
]);
```

詳細については API リファレンス を参照して下さい。

14.3 コア・バリデータ

Yii は、一般的に使われる一連のコア・バリデータを提供しています。コア・バリデータは、主として、yii\validators 名前空間の下にあります。長ったらしいバリデータ・クラス名を使う代りに、エイリアス を使って使用するコア・バリデータを指定することが出来ます。 例えば、yii\validators\RequiredValidator クラスを参照するのに required というエイリアスを使うことが出来ます。

```
public function rules()
{
    return [
        [['email', 'password'], 'required'],
    ];
}
```

yii\validators\Validator::$builtInValidators プロパティがサポートされている全てのコア・バリデータのエイリアスを宣言しています。

以下では、全てのコア・バリデータについて、主な使用方法とプロパティを説明します。

14.3.1 boolean

```
[
    // データ型にかかわらず、"selected" が 0 または 1 であるかどうかチェック
    ['selected', 'boolean'],
```

```
    // "deleted" が boolean 型であり、true または false であるかどうかチェック
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false, '
    strict' => true],
]
```

このバリデータは、入力値が真偽値であるかどうかをチェックします。

- `trueValue`: `true` を表す値。デフォルト値は `'1'`。
- `falseValue`: `false` を表す値。デフォルト値は `'0'`。
- `strict`: 入力値の型が `trueValue` と `falseValue` の型と一致しなければならないかどうか。デフォルト値は `false`。

  補足: HTML フォームで送信されたデータ入力値は全て文字列であるため、通常は、 `strict` プロパティは `false` のままにすべきです。

14.3.2 captcha

```
[
    ['verificationCode', 'captcha'],
]
```

このバリデータは、通常、`yii\captcha\CaptchaAction` および `yii\captcha\Captcha` と一緒に使われ、 入力値が `CAPTCHA` ウィジェットによって表示された検証コードと同じであることを確認します。

- `caseSensitive`: 検証コードの比較で大文字と小文字を区別するかどうか。デフォルト値は `false`。
- `captchaAction`: CAPTCHA 画像を表示する `CAPTCHA` アクション に対応する ルート。デフォルト値は `'site/captcha'`。
- `skipOnEmpty`: 入力値が空のときに検証をスキップできるかどうか。デフォルト値は `false` で、 入力が必須であることを意味します。

14.3.3 compare

```
[
    // "password" 属性の値が "password_repeat" 属性の値と同じであるかどうか検
    証する
    ['password', 'compare'],

    // 上記と同じだが、比較する属性を明示的に指定
    ['password', 'compare', 'compareAttribute' => 'password_repeat'],

    // "age" が 30 以上であるかどうか検証する
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>=', 'type' => '
    number'],
]
```

このバリデータは指定された入力値を他の値と比較し、両者の関係が `operator` プロパティで指定されたものであることを確認します。

- compareAttribute: その値が比較対象となる属性の名前。 このバリデータが属性を検証するのに使用されるとき、このプロパティのデフォルト値はその属性の名前に接尾辞 _repeat を付けた名前になります。 例えば、検証される属性が password であれば、このプロパティのデフォルト値は password_repeat となります。
- compareValue: 入力値が比較される定数値。 このプロパティと compareAttribute の両方が指定された場合は、このプロパティが優先されます。
- operator: 比較演算子。デフォルト値は == で、入力値が compareAttribute の値または compareValue と等しいことを検証することを意味します。 次の演算子がサポートされています。
  - ==: 二つの値が等しいことを検証。厳密でない比較を行う。
  - ===: 二つの値が等しいことを検証。厳密な比較を行う。
  - !=: 二つの値が等しくないことを検証。厳密でない比較を行う。
  - !==: 二つの値が等しくないことを検証。厳密な比較を行う。
  - >: 検証される値が比較される値よりも大きいことを検証する。
  - >=: 検証される値が比較される値よりも大きいか等しいことを検証する。
  - <: 検証される値が比較される値よりも小さいことを検証する。
  - <=: 検証される値が比較される値よりも小さいか等しいことを検証する。
- type: デフォルトの比較タイプは ‘string‘ (文字列) であり、その場合、値は 1 バイトごとに比較されます。 数値を比較する場合は、必ず $type を ‘number‘ に設定して、 数値としての比較を有効にして下さい。

日付の値を比較する

compare バリデータは、文字列や数値を比較するためにしか使えません。 日付のような値を比較する必要がある場合は、二つの選択肢があります。 日付をある固定値と比較するときは、単に date バリデータを使って、 その $min や $max のプロパティを指定すれば良いでしょう。 フォームに入力された二つの日付、例えば、fromDate と toDate のフィールドを比較する必要がある場合は、 次のように、compare バリデータと date バリデータを組み合わせて使うことが出来ます。

```
['fromDate', 'date', 'timestampAttribute' => 'fromDate'],
['toDate', 'date', 'timestampAttribute' => 'toDate'],
['fromDate', 'compare', 'compareAttribute' => 'toDate', 'operator' => '<', '
    enableClientValidation' => false],
```

バリデータは指定された順序に従って実行されますので、まず最初に、fromDate と toDate に入力された値が有効な日付であることが確認されます。 そして、有効な日付であった場合は、機械が読める形式に変換されます。 その後に、これらの二つの値が compare バリデータによって比較されます。 現在、date バリデータはクライアント・

サイドのバリデーションを提供していませんので、これはサーバ・サイドでのみ動作します。 そのため、compare バリデータについても、$enableClientValidation は false に設定されています。

14.3.4 date

date バリデータには三つの異なる ショートカットがあります。

```
[
    [['from_date', 'to_date'], 'date'],
    [['from_datetime', 'to_datetime'], 'datetime'],
    [['some_time'], 'time'],
]
```

このバリデータは、入力値が正しい書式の date、time、または datetime であるかどうかをチェックします。 オプションとして、入力値を UNIX タイムスタンプ (または、その他、機械による読み取りが可能な形式) に変換して、 timestampAttribute によって指定された属性に保存することも出来ます。

- format: 検証される値が従っているべき日付/時刻の書式。 これには ICU manual[7] で記述されている日付/時刻のパターンを使うことが出来ます。 あるいは、PHP の Datetime クラスによって認識される書式に接頭辞 php: を付けた文字列でも構いません。 サポートされている書式については、https://secure.php.net/manual/ja/datetime.createfromformat.php を参照してください。 このプロパティが設定されていないときは、Yii::$app->formatter->dateFormat の値を取ります。
- timestampAttribute: このバリデータが、入力された日付/時刻から変換した UNIX タイムスタンプを代入することが出来る属性の名前。 これは、検証される属性と同じ属性であってもかまいません。 その場合は、元の値は検証実行後にタイムスタンプで上書きされます。 DatePicker で日付の入力を扱う[8] に使用例がありますので、参照してください。
  バージョン 2.0.4 以降では、$timestampAttributeFormat と $timestampAttributeTimeZone を使って、 この属性に対するフォーマットとタイム・ゾーンを指定することが出来ます。

timestampAttribute を使う場合、入力値が UNIX タイムスタンプに変換されること、そして、UNIX タイムスタンプは定義により UTC であることに注意して下さい。 すなわち、入力のタイム・ゾーン から UTC への変換が実行されます。

- バージョン 2.0.4 以降では、タイムスタンプの 最小値 または 最大値 を指定することも出来ます。

[7] http://userguide.icu-project.org/formatparse/datetime#TOC-Date-Time-Format-Syntax

[8] https://github.com/yiisoft/yii2-jui/blob/master/docs/guide-ja/topics-date-picker.md

入力が必須でない場合には、date バリデータに加えて、default バリデータ (デフォルト値フィルタ) を追加すれば、 空の入力値が `null` として保存されることを保証することが出来ます。そうしないと、データベースに `0000-00-00` という日付が保存されたり、 デート・ピッカーの入力フィールドが `1970-01-01` になったりしてしまいます。

```
[
    [['from_date', 'to_date'], 'default', 'value' => null],
    [['from_date', 'to_date'], 'date'],
]
```

14.3.5 default

```
[
    // 空のときは "age" を null にする
    ['age', 'default', 'value' => null],

    // 空のときは "country" を "USA" にする
    ['country', 'default', 'value' => 'USA'],

    // 空のときは "from" と "to" に今日から三日後・六日後の日付を入れる
    [['from', 'to'], 'default', 'value' => function ($model, $attribute) {
        return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6
     days'));
    }],
]
```

このバリデータはデータを検証しません。 その代りに、検証される属性が空のときに、その属性にデフォルト値を割り当てます。

- `value`: デフォルト値、または、デフォルト値を返す PHP コーラブル。 検証される属性が空のときにこのデフォルト値が割り当てられます。PHP コーラブルのシグニチャは、次のものでなければなりません。

```
function foo($model, $attribute) {
    // ... $value を計算 ...
    return $value;
}
```

> 情報: 値が空であるか否かを決定する方法については、独立したトピックとして、空の入力値を扱う のセクションでカバーされています。 データベース・スキーマによるデフォルト値は、モデルの loadDefaultValues() によってロードすることが出来ます。

14.3.6 double

```
[
    // "salary" が実数であるかどうかチェック
    ['salary', 'double'],
]
```

このバリデータは、入力値が実数値であるかどうかをチェックします。number バリデータと等価です。

- max: 上限値 (その値を含む)。設定されていない場合は、バリデータが上限値をチェックしないことを意味します。
- min: 下限値 (その値を含む)。設定されていない場合は、バリデータが下限値をチェックしないことを意味します。

14.3.7 each

情報: このバリデータは、バージョン 2.0.4 以降で利用できます。

```
[
    // 全てのカテゴリ ID が整数であるかどうかチェックする
    ['categoryIDs', 'each', 'rule' => ['integer']],
]
```

このバリデータは配列の属性に対してのみ働きます。 これは、配列の 全ての 要素が指定された検証規則による検証に成功するかどうかを調べるものです。 上の例では、categoryIDs 属性は配列を値として取らなければならず、配列の各要素は integer の検証規則によって検証されることになります。

- rule: 検証規則を指定する配列。配列の最初の要素がバリデータのクラス名かエイリアスを指定します。 配列の残りの「名前・値」のペアが、バリデータ・オブジェクトを構成するのに使われます。
- allowMessageFromRule: 埋め込まれた検証規則によって返されるエラー・メッセージを使うかどうか。 デフォルト値は true です。これが false の場合は、message をエラー・メッセージとして使います。

  補足: 属性が配列でない場合は、検証が失敗したと見なされ、message がエラー・メッセージとして返されます。

14.3.8 email

```
[
    // "email" が有効なメール・アドレスであるかどうかチェック
    ['email', 'email'],
]
```

このバリデータは、入力値が有効なメール・アドレスであるかどうかをチェックします。

- allowName: メール・アドレスに表示名 (例えば、John Smith <john.smith@example.com>) を許容するか否か。デフォルト値は false。
- checkDNS: メールのドメインが存在して A または MX レコードを持っているかどうかをチェックするか否か。 このチェックは、メール・アドレスが実際には有効なものでも、一時的な DNS の問題によって失敗する場合があることに注意してください。 デフォルト値は false。
- enableIDN: 検証のプロセスが IDN (国際化ドメイン名) を考慮に入れるか否か。 デフォルト値は false。 IDN の検証を使用するためには、intl PHP 拡張をインストールして有効化する必要があることに注意してください。そうしないと、例外が投げられます。

14.3.9 exist

```
[
    // a1 の入力値が a1 のカラムに存在する必要がある
    ['a1', 'exist'],

    // a1 の入力値が a2 のカラムに存在する必要がある
    ['a1', 'exist', 'targetAttribute' => 'a2'],

    // a1 と a2 の両方が存在する必要がある。両者はともにエラー・メッセージを受け取る
    [['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 と a2 の両方が存在する必要がある。a1 のみがエラー・メッセージを受け取る
    ['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a2 の値が a2 のカラム、a1 の値が a3 のカラムに存在する必要がある。a1 がエラー・メッセージを受け取る
    ['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

    // a1 が存在する必要がある。a1 が配列である場合は、その全ての要素が存在する必要がある
    ['a1', 'exist', 'allowArray' => true],

    // type_id が ProductType クラスで定義されているテーブルの id カラムに存在する必要がある
    ['type_id', 'exist', 'targetClass' => ProductType::class, 'targetAttribute' => ['type_id' => 'id']],

    // 同上。定義済みの "type" リレーションを使用。
    ['type_id', 'exist', 'targetRelation' => 'type'],
]
```

このバリデータは、入力値が アクティブ・レコード の属性によって表されるテーブルのカラムに存在するかどうかをチェックします。 targetAttribute を使って アクティブ・レコード の属性を指定し、 targetClass

によって対応するクラスを指定することが出来ます。 これらを指定しない場合は、検証されるモデルの属性とクラスが使用されます。

このバリデータは、一つまたは複数のカラムに対する検証に使用することが出来ます (複数のカラムに対する検証の場合は、それらの属性の組み合せが存在しなければならないことを意味します)。

- `targetClass`: 検証される入力値を探すために使用される アクティブ・レコード クラスの名前。 設定されていない場合は、現在検証されているモデルのクラスが使用されます。
- `targetAttribute`: `targetClass` において、入力値の存在を検証するために使用される属性の名前。 設定されていない場合は、現在検証されている属性の名前が使用されます。 複数のカラムの存在を同時に検証するために配列を使うことが出来ます。 配列の値は存在を検証するのに使用される属性であり、配列のキーは入力値が検証される属性です。 キーと値が同じ場合は、値だけを指定することが出来ます。
- `filter`: 入力値の存在をチェックするのに使用される DB クエリに適用される追加のフィルタ。 これには、文字列、または、追加のクエリ条件を表現する配列を使うことが出来ます (クエリ条件の書式については、`yii\db\Query::where()` を参照してください)。 または、`function ($query)` というシグニチャを持つ無名関数でも構いません。 `$query` は関数の中で修正できる `Query` オブジェクトです。
- `allowArray`: 入力値が配列であることを許容するか否か。デフォルト値は `false`。 このプロパティが `true` で入力値が配列であった場合は、配列の全ての要素がターゲットのカラムに存在しなければなりません。 `targetAttribute` を配列で指定して複数のカラムに対して検証しようとしている場合は、このプロパティを `true` に設定することが出来ないことに注意してください。

14.3.10 `file`

```
[
    // "primaryImage" が 、、またはPNGJPG GIF 形式のアップロードされた
    // 画像ファイルであり、ファイルサイズが 1MB 以下であるかどうかチェック
    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'], 'maxSize
    ' => 1024*1024],
]
```

このバリデータは、入力値がアップロードされた有効なファイルであるかどうかをチェックします。

- `extensions`: アップロードを許可されるファイル名拡張子のリスト。 リストは、配列、または、空白かカンマで区切られたファイル名拡張子からなる文字列 (例えば、"gif, jpg") で指定することが出来ます。 拡張子名は大文字と小文字を区別しません。 デフォルト値は `null` であり、すべてのファイル名拡張子が許可されることを意味します。

- mimeTypes: アップロードを許可されるファイルの MIME タイプのリスト。 リストは、配列、または、空白かカンマで区切られたファイルの MIME タイプからなる文字列 (例えば、"image/jpeg, image/png") で指定することが出来ます。 特殊文字 * によるワイルドカードのマスクを使って、一群の MIME タイプに一致させることも出来ます。 例えば image/* は、image/ で始まる全ての MIME タイプ (image/jpeg, image/png など) を通します。 MIME タイプ名は大文字と小文字を区別しません。デフォルト値は null であり、すべての MIME タイプが許可されることを意味します。 MIME タイプの詳細については、一般的なメディア・タイプ[9] を参照してください。
- minSize: アップロードされるファイルに要求される最小限のバイト数。デフォルト値は null であり、下限値が無いことを意味します。
- maxSize: アップロードされるファイルに許可される最大限のバイト数。デフォルト値は null であり、上限値が無いことを意味します。
- maxFiles: 指定された属性が保持しうる最大限のファイル数。 デフォルト値は 1 であり、入力値がアップロードされた一つだけのファイルでなければならないことを意味します。 この値が 2 以上である場合は、入力値は最大で maxFiles 数のアップロードされたファイルからなる配列でなければなりません。
- checkExtensionByMimeType: ファイルの MIME タイプでファイル拡張子をチェックするか否か。 MIME タイプのチェックから導かれる拡張子がアップロードされたファイルの拡張子と違う場合に、そのファイルは無効であると見なされます。 デフォルト値は true であり、そのようなチェックが行われることを意味します。

FileValidator は yii\web\UploadedFile と一緒に使用されます。 ファイルのアップロードおよびアップロードされたファイルの検証の実行に関する完全な説明は、ファイルをアップロードする のセクションを参照してください。

14.3.11 filter

```
[
    // "username" と "email" の入力値をトリムする
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' =>
    true],

    // "phone" の入力値を正規化する
    ['phone', 'filter', 'filter' => function ($value) {
        // 電話番号の入力値をここで正規化する
        return $value;
    }],
```

[9] http://en.wikipedia.org/wiki/Internet_media_type#List_of_common_media_types

```
    // 関数 "normalizePhone" を使って "phone" の入力値を正規化する
    ['phone', 'filter', 'filter' => [$this, 'normalizePhone']],

    public function normalizePhone($value) {
        return $value;
    }
]
```

このバリデータはデータを検証しません。 代りに、入力値にフィルタを適用して、それを検証される属性に書き戻します。

- `filter`: フィルタを定義する PHP コールバック。これには、グローバル関数の名前、無名関数などを指定することが出来ます。 関数のシグニチャは `function ($value) { return $newValue; }` でなければなりません。このプロパティは必須項目です。
- `skipOnArray`: 入力値が配列である場合にフィルタをスキップするか否か。デフォルト値は `false`。 フィルタが配列の入力を処理できない場合は、このプロパティを `true` に設定しなければなりません。そうしないと、何らかの PHP エラーが生じ得ます。

  ヒント: 入力値をトリムしたい場合は、trim バリデータを直接使うことが出来ます。

  ヒント: `filter` のコールバックに期待されるシグニチャを持つ PHP 関数が多数存在します。 例えば、(intval[10] や boolval[11] などを使って) 型キャストを適用し、属性が特定の型になるように保証したい場合は、 それらの関数をクロージャで包む必要はなく、単にフィルタの関数名を指定するだけで十分です。

  ```
  ['property', 'filter', 'filter' => 'boolval'],
  ['property', 'filter', 'filter' => 'intval'],
  ```

14.3.12 image

```
[
    // "primaryImage" が適切なサイズの有効な画像であることを検証
    ['primaryImage', 'image', 'extensions' => 'png, jpg',
        'minWidth' => 100, 'maxWidth' => 1000,
        'minHeight' => 100, 'maxHeight' => 1000,
    ],
]
```

このバリデータは、入力値が有効な画像ファイルであるかどうかをチェックします。 これは file バリデータを拡張するものであり、従って、そのプロパティの全てを継承しています。 それに加えて、画像の検証の目的に特化した次のプロパティをサポートしています。

[10] https://secure.php.net/manual/ja/function.intval.php
[11] https://secure.php.net/manual/ja/function.boolval.php

- minWidth: 画像の幅の最小値。デフォルト値は null であり、下限値がないことを意味します。
- maxWidth: 画像の幅の最大値。デフォルト値は null であり、上限値がないことを意味します。
- minHeight: 画像の高さの最小値。デフォルト値は null であり、下限値がないことを意味します。
- maxHeight: 画像の高さの最大値。デフォルト値は null であり、上限値がないことを意味します。

14.3.13 ip

```
[
    // "ip_address" が有効な IPv4 または IPv6 アドレスであることを検証
    ['ip_address', 'ip'],

    // "ip_address" が有効な IPv6 アドレスまたはサブネットであることを検証
    // 値は完全な IPv6 記法に展開される
    ['ip_address', 'ip', 'ipv4' => false, 'subnet' => null, 'expandIPv6' =>
    true],

    // "ip_address" が有効な IPv4 または IPv6 アドレスであることを検証
    // 先頭に否定文字 '!' を置くことを許可
    ['ip_address', 'ip', 'negation' => true],
]
```

このバリデータは属性の値が有効な IPv4/IPv6 アドレスまたはサブネットであることを検証します。 正規化または IPv6 展開が有効にされた場合は、属性の値を変更することも出来ます。

バリデータは以下の構成オプションを持っています。

- ipv4: 検証の対象となる値が IPv4 アドレスであってよいか否か。デフォルト値は true。
- ipv6: 検証の対象となる値が IPv6 アドレスであってよいか否か。デフォルト値は true。
- subnet: アドレスが 192.168.10.0/24 のような CIDR サブネットを持つ IP であってよいか否か。
  - true - サブネットが必要。CIDR の無いアドレスは却下されます
  - false - アドレスは CIDR を伴ってはいけません
  - null - CIDR は有っても無くても構いません

  デフォルト値は false。
- normalize: CIDR を持たないアドレスに、最も短い (IPv4 では 32、IPv6 では 128) CIDR プレフィクスを追加するか否か。 subnet が false 以外の場合にのみ動作します。例えば、
  - 10.0.1.5 は 10.0.1.5/32 に正規化され、
  - 2008:db0::1 は 2008:db0::1/128 に正規化されます

  デフォルト値は false。

- negation: 検証の対象となるアドレスが先頭に否定文字 ! を持つことが出来るか否か。デフォルト値は false。
- expandIPv6: IPv6 アドレスを完全な記法に展開するか否か。 例えば、2008:db0::1 は 2008:0db0:0000:0000:0000:0000:0000:0001 に展開されます。デフォルト値は false。
- ranges: 許容または禁止される IPv4 または IPv6 の範囲の配列。
  配列が空の場合、またはこのオプションが設定されていない場合は、全ての IP アドレスが許容されます。 そうでない場合は、最初に合致するものが見つかるまで、規則が順番にチェックされます。どの規則にも合致しなかった場合、その IP アドレスは禁止されます。
  例えば、 ‘php [

```
'client_ip', 'ip', 'ranges' => [
    '192.168.10.128'
    '!192.168.10.0/24',
    'any' // 他の IP アドレスは全て許容
]
```

  ] ‘ この例では、192.168.10.0/24 のサブネットを除いて、全ての IPv4 および IPv6 アドレスが許容されます。 IPv4 アドレス 192.168.10.128 も、制約の前にリストされているため、同様に許容されます。
- networks: ranges で使用する事が出来るネットワークのエイリアスの配列。配列の形式は、
  – キー - エイリアス名
  – 値 - 文字列の配列。文字列は、範囲、IP アドレス、または、他のエイリアスとすることが出来ます。 また、文字列は (negation オプションとは独立に) ! によって否定することが出来ます。

  デフォルトで、次のエイリアスが定義されています。
  — *: any
  — any: 0.0.0.0/0, ::/0
  — private: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, fd00::/8
  — multicast: 224.0.0.0/4, ff00::/8
  — linklocal: 169.254.0.0/16, fe80::/10
  — localhost: 127.0.0.0/8', ::1
  — documentation: 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, 2001:db8::/32
  — system: multicast, linklocal, localhost, documentation

  情報: このバリデータは、バージョン 2.0.7 以降で利用することが出来ます。

14.3.14 in

```
[
```

```
    // "level" が 1, 2 または 3 であるかどうかチェック
    ['level', 'in', 'range' => [1, 2, 3]],
]
```

このバリデータは、入力値が所与の値のリストにあるかどうかをチェックします。

- range: 与えられた値のリスト。この中に、入力値がなければならない。
- strict: 入力値と所与の値の比較が厳密でなければならない (型と値の両方が同じでなければならない) かどうか。 デフォルト値は false。
- not: 検証結果を反転すべきか否か。デフォルト値は false。 このプロパティが true に設定されているときは、入力値が所与の値のリストにない場合に検証が成功したとされます。
- allowArray: 入力値が配列であることを許可するかどうか。 このプロパティが true であるときに、入力値が配列である場合は、配列の全ての要素が所与の値のリストにある必要があり、そうでなければ検証は失敗します。

14.3.15 integer

```
[
    // "age" が整数であるかどうかチェック
    ['age', 'integer'],
]
```

このバリデータは入力値が整数であるかどうかをチェックします。

- max: 上限値 (その値を含む)。設定されていないときは、バリデータは上限をチェックしません。
- min: 下限値 (その値を含む)。設定されていないときは、バリデータは下限をチェックしません。

14.3.16 match

```
[
    // "username" が英字から始まり、英字、数字、アンダーバーだけで構成されているかどうかチェック
    ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']
]
```

このバリデータは、入力値が指定された正規表現に一致するかどうかをチェックします。

- pattern: 入力値が一致すべき正規表現。このプロパティを設定することは必須です。 そうしないと、例外が投げられます。
- not: 検証結果を反転すべきかどうか。 デフォルト値は false で、入力値がパターンに一致したときにだけ検証が成功することを意味します。 このプロパティが true に設定されているときは、入力値がパターンに一致しない場合にだけ検証が成功したと見なされます。

14.3.17 number

```
[
    // "salary" が数値であるかどうかチェック
    ['salary', 'number'],
]
```

このバリデータは、入力値が数値であるかどうかをチェックします。double バリデータと等価です。

- max: 上限値 (その値を含む)。設定されていないときは、バリデータは上限をチェックしません。
- min: 下限値 (その値を含む)。設定されていないときは、バリデータは下限をチェックしません。

14.3.18 required

```
[
    // "username" と "password" がともに空ではないことをチェックする
    [['username', 'password'], 'required'],
]
```

このバリデータは、入力値が提供されており、空ではないことをチェックします。

- requiredValue: 入力値として要求される値。このプロパティが設定されていない場合は、入力値が空ではいけないことを意味します。
- strict: 値を検証するときに、データ型をチェックするかどうか。デフォルト値は false。 requiredValue が設定されていない場合、このプロパティが true であるときは、バリデータは入力値が厳密な意味で null であるかどうかをチェックします。 一方、このプロパティが false であるときは、値が空か否かの判断に緩い規則を使います。 requiredValue が設定されている場合、このプロパティが true であるときは、 入力値と requiredValue を比較するときに型のチェックを行います。

  情報: 値が空であるか否かを決定する方法については、独立したトピックとして、 空の入力値を扱う のセクションでカバーされています。

14.3.19 safe

```
[
    // "description" を安全な属性としてマーク
    ['description', 'safe'],
]
```

このバリデータは検証を実行しません。 その代りに、このバリデータは、属性を 安全な属性 としてマークするために使われます。

14.3.20 string

```
[
    // "username" が、長さが 4 以上 24 以下の文字列であるかどうかチェック
    ['username', 'string', 'length' => [4, 24]],
]
```

このバリデータは、入力値が一定の長さを持つ有効な文字列であるかどうかをチェックします。

- length: 検証される入力文字列の長さの制限を指定します。 これは、次のいずれかの形式で指定することが出来ます。
  - 一つの整数: 文字列がちょうどその長さでなければならない、その長さ。
  - 一つの要素を持つ配列: 入力文字列の長さの最小値 (例えば、[8])。これは min を上書きします。
  - 二つの要素を持つ配列: 入力文字列の長さの最小値と最大値 (例えば、[8, 128])。 これは min と max の両方を上書きします。
- min: 入力文字列の長さの最小値。設定されていない時は、長さの下限値がないことを意味します。
- max: 入力文字列の長さの最大値。設定されていない時は、長さの上限値がないことを意味します。
- encoding: 検証される入力文字列の文字エンコーディング。設定されていない時は、 アプリケーションの charset の値が使われ、デフォルトでは UTF-8 となります。

14.3.21 trim

```
[
    // "username" と "email" の前後にあるホワイトスペースをトリムする
    [['username', 'email'], 'trim'],
]
```

このバリデータはデータの検証を実行しません。その代りに、入力値の前後にあるホワイト・スペースをトリムします。 入力値が配列であるときは、このバリデータによって無視されることに注意してください。

14.3.22 unique

```
[
    // a1 の入力値が a1 のカラムにおいてユニークである必要がある
    ['a1', 'unique'],

    // a1 の入力値が a2 のカラムにおいてユニークである必要がある
    ['a1', 'unique', 'targetAttribute' => 'a2'],

    // a1 と a2 の両方がユニークである必要がある。両者がともにエラー・メッセージを受け取る
    [['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],
```

```
    // a1 と a2 の両方がユニークである必要がある。a1 のみがエラー・メッセージを
    受け取る
    ['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a2 の値が a2 のカラム、a1 の値が a3 のカラムにおいてユニークである必要
    がある。a1 がエラー・メッセージを受け取る
    ['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],
]
```

このバリデータは、入力値がテーブルのカラムにおいてユニークであるかどうかをチェックします。 アクティブ・レコード モデルの属性に対してのみ働きます。 一つのカラムに対する検証か、複数のカラムに対する検証か、どちらかをサポートします。

- `targetClass`: 検証される入力値を探すために使用される アクティブ・レコード クラスの名前。 設定されていない場合は、現在検証されているモデルのクラスが使用されます。
- `targetAttribute`: `targetClass` において、入力値がユニークであることを検証するために使用される属性の名前。 設定されていない場合は、現在検証されている属性の名前が使用されます。 複数のカラムのユニーク性を同時に検証するために配列を使うことが出来ます。配列の値はユニーク性を検証するのに使用される属性であり、配列のキーはその入力値が検証される属性です。 キーと値が同じ場合は、値だけを指定することが出来ます。
- `filter`: 入力値のユニーク性をチェックするのに使用される DB クエリに適用される追加のフィルタ。 これには、文字列、または、追加のクエリ条件を表現する配列を使うことが出来ます (クエリ条件の書式については、`yii\db\Query::where()` を参照してください)。または、`function ($query)` というシグニチャを持つ無名関数でも構いません。`$query` は関数の中で修正できる `Query` オブジェクトです。

14.3.23 url

```
[
    // "website" が有効な URL であるかどうかをチェック。
    // URI スキームを持たない場合は、"website" 属性に "http://" を前置する
    ['website', 'url', 'defaultScheme' => 'http'],
]
```

このバリデータは、入力値が有効な URL であるかどうかをチェックします。

- `validSchemes`: 有効と見なされるべき URI スキームを指定する配列。デフォルト値は `['http', 'https']` であり、`http` と `https` の URL がともに有効と見なされることを意味します。
- `defaultScheme`: 入力値がスキームの部分を持たないときに前置されるデフォルトの URI スキーム。 デフォルト値は `null` であり、入力値を修正しないことを意味します。

- enableIDN: バリデータが IDN (国際化ドメイン名) を考慮すべきか否か。デフォルト値は false。 IDN の検証を使用するためには、intl PHP 拡張をインストールして有効化する必要があることに注意してください。 そうしないと、例外が投げられます。

  補足: このバリデータは URL スキームとホスト部分が正しいものであることを検証します。 URL の残りの部分はチェックしません。また、XSS や他の攻撃に対して防御するように設計されてもいません。 アプリケーション開発における脅威に対する防御について更に学習するためにセキュリティのベスト・プラクティス を参照して下さい。

14.4 国際化

国際化 (I18N) とは、工学的な変更を伴わずにさまざまな言語と地域に順応できるように、 ソフトウェア・アプリケーションを設計するプロセスを指します。 潜在的なユーザが世界中にいるウェブ・アプリケーションにとっては、このことは特に重要な意味を持ちます。 Yii は、全ての領域にわたる国際化機能を提供し、メッセージの翻訳、ビューの翻訳、日付と数字の書式設定をサポートします。

14.4.1 ロケールと言語

ロケール

ロケールとは、ユーザの言語、国、そして、ユーザが彼らのユーザ・インタフェイスにおいて目にすることを期待する すべての変異形式を定義する一連のパラメータです。 ロケールは、通常、言語 ID と地域 ID から成るロケール ID によって定義されます。

例えば、en-US という ID は、「英語とアメリカ合衆国」というロケールを意味します。

Yii アプリケーションで使用される全てのロケール ID は、一貫性のために、 ll-CC の形式に正規化されなければなりません。 ここで ll は ISO-639[12] に従った小文字二つまたは三つの言語コードであり、 CC は ISO-3166[13] に従った二文字の国コードです。 ロケールに関する更なる詳細は ICU プロジェクトのドキュメント[14] に述べられています。

言語

Yii では、”言語” という用語でロケールに言及することがしばしばあります。

[12] http://www.loc.gov/standards/iso639-2/

[13] http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html

[14] http://userguide.icu-project.org/locale#TOC-The-Locale-Concept

Yii のアプリケーションでは二つの言語を使用します。すなわち、

- ソース言語：ソース・コード中のテキスト・メッセージが書かれている言語。
- ターゲット言語：コンテントをエンド・ユーザに表示するのに使用されるべき言語。

いわゆるメッセージ翻訳サービスは、主として、テキスト・メッセージをソース言語からターゲット言語に翻訳するものです。

構成

アプリケーションの言語は、アプリケーションの構成情報で次のように構成することが出来ます。

```
return [
    // ターゲット言語を日本語に設定
    'language' => 'ja-JP',

    // ソース言語を英語に設定
    'sourceLanguage' => 'en-US',

    ......
];
```

ソース言語 のデフォルト値は en-US であり、合衆国の英語を意味します。 このデフォルト値は変えないことが 推奨 されます。 なぜなら、通常は、英語から他の言語への翻訳者を見つける方が、非英語から非英語への翻訳者を見つけるより、はるかに簡単だからです。

ターゲット言語 は、エンド・ユーザの言語選択など、 さまざまな要因に基づいて、動的に設定しなければならないことがよくあります。アプリケーションの構成情報で構成するかわりに、次の文を使ってターゲット言語を変更することが出来ます。

```
// ターゲット言語を中国語に変更
\Yii::$app->language = 'zh-CN';
```

> ヒント: ソース言語がコードの部分によって異なる場合は、メッセージ・ソースごとにソース言語をオーバーライドすることが出来ます。 これについては、次の説で説明します。

14.4.2 メッセージ翻訳

ソース言語からターゲット言語へ

メッセージ翻訳サービスは、テキスト・メッセージをある言語 (通常はソース言語) から別の言語 (通常は ターゲット言語) に翻訳するものです。

翻訳は、元のメッセージと翻訳されたメッセージを格納するメッセージ・ソースの中から、翻訳対象となったメッセージを探すことにより行

われます。 メッセージが見つかれば、対応する翻訳されたメッセージが返されます。メッセージが見つからなければ、元のメッセージが翻訳されずに返されます。

実装の仕方

メッセージ翻訳サービスを使用するためには、主として次の作業をする必要があります。

1. 翻訳する必要のある全てのテキスト・メッセージを `Yii::t()` メソッドの呼び出しの中に包む。
2. メッセージ翻訳サービスが翻訳されたメッセージを探すことが出来る一つまたは複数のメッセージ・ソースを構成する。
3. 翻訳者にメッセージを翻訳させて、それをメッセージ・ソースに格納する。

1. テキスト・メッセージを包む `Yii::t()` メソッドは次のように使います。

```
echo \Yii::t('app', 'This is a string to translate!');
```

ここで、二番目のパラメータが翻訳されるべきテキスト・メッセージを示し、 最初のパラメータはメッセージを分類するのに使用されるカテゴリ名を示します。

2. 一つまたは複数のメッセージ・ソースを構成する `Yii::t()` メソッドは `i18n` アプリケーション・コンポーネント の `translate` メソッドを呼んで実際の翻訳作業を実行します。 このコンポーネントはアプリケーションの構成情報の中で次のようにして構成することが出来ます。

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                //'sourceLanguage' => 'en-US',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

上記のコードにおいては、`yii\i18n\PhpMessageSource` によってサポートされるメッセージ・ソースが構成されています。

シンボル * によるカテゴリのワイルドカード　app* は、app で始まる全てのメッセージ・カテゴリが このメッセージ・ソースを使って翻訳されるべきであることを示しています。

3. 翻訳者にメッセージを翻訳させて、それをメッセージ・ソースに格納する　yii\i18n\PhpMessageSource クラスは、単純な PHP 配列を持つ複数の PHP ファイルを使用してメッセージ翻訳を格納します。 それらのファイルが、「ソース言語」のメッセージと「ターゲット言語」の翻訳とのマップを含みます。

> 情報: それらのファイルを message コマンド を使用して自動的に生成することが出来ます。 このセクションで後で紹介します。

PHP ファイルは、それぞれ、一つのカテゴリのメッセージに対応します。 デフォルトでは、ファイル名はカテゴリ名と同じでなければなりません。app/messages/nl-NL/main.ph の例を示します。

```
<?php

/**
* Translation map for nl-NL
*/
return [
    'welcome' => 'welkom'
];
```

ファイルのマッピング　ただし、fileMap を構成して、別の命名方法によってカテゴリを PHP ファイルにマップすることも可能です。

上記の例では、(ja-JP がターゲット言語であると仮定すると) app/error のカテゴリは @app/messages/ja-JP/error.php という PHP ファイルにマップされます。 fileMap を構成しなければ、このカテゴリは @app/messages/ja-JP/app/error.php にマップされることになります。

他のストレージ・タイプ　翻訳メッセージを格納するのには、PHP ファイル以外に、 次のメッセージ・ソースを使うことも可能です。

- yii\i18n\GettextMessageSource - 翻訳メッセージを保持するのに GNU Gettext の MO ファイルまたは PO ファイルを使用する
- yii\i18n\DbMessageSource - 翻訳メッセージを保存するのにデータベース・テーブルを使用する

14.4.3 メッセージのフォーマット

メッセージを翻訳するときには、プレースホルダを埋め込んで、動的なパラメータ値で実行時に置き換えさせることが出来ます。 更には、パラメータ値をターゲット言語に応じてフォーマットさせるための特別なプ

レースホルダの構文を使うことも出来ます。 この項では、メッセージをフォーマットする様々な方法を説明します。

> 補足: 以下においては、メッセージ・フォーマットの理解を助けるために、原文にはない日本語への翻訳例 (とその出力結果) をコード・サンプルに追加しています。

メッセージ・パラメータ

翻訳対象となるメッセージには、一つまたは複数のパラメータ (プレースホルダとも呼びます) を埋め込んで、 与えられたパラメータ値で置き換えられるようにすることが出来ます。 様々なパラメータ値のセットを与えることによって、翻訳されるメッセージを動的に変化させることが出来ます。 次の例では、`'Hello, {username}!'` というメッセージの中のプレースホルダ `{username}` が `'Alexander'` と `'Qiang'` にそれぞれ置き換えられます。

```
$username = 'Alexander';
// username が "Alexander" になった翻訳メッセージを表示
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);

$username = 'Qiang';
// username が "Qiang" になった翻訳メッセージを表示
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);
```

プレースホルダを持つメッセージを翻訳する時には、プレースホルダはそのままにしておかなければなりません。 これは、プレースホルダは `Yii::t()` を呼んでメッセージを翻訳する時に、実際の値に置き換えられるものだからです。

```
// 日本語翻訳: '{username} さん、こんにちは!'
```

プレースホルダには、名前付きプレースホルダ と 序数プレースホルダ のどちらかを使用する事が出来ます。ただし、一つのメッセージに両方を使うことは出来ません。

上記の例は名前付きプレースホルダの使い方を示すものです。 すなわち、各プレースホルダは `{name}` という形式で書かれていますが、それに対して、キーが(波括弧なしの)プレースホルダ名であり、 値がそのプレースホルダを置き換える値である連想配列を渡す訳です。

序数プレースホルダは、0 ベースの整数の序数をプレースホルダ名として使います。 このプレースホルダは、`Yii::t()` の呼び出しに出現する順序に従って、パラメータ値によって置き換えられます。 次の例では、序数プレースホルダ `{0}`、`{1}` および `{2}` は、それぞれ、`$price`、`$count` および `$subtotal` の値によって置き換えられます。

```
$price = 100;
$count = 2;
$subtotal = 200;
echo \Yii::t('app', 'Price: {0}, Count: {1}, Subtotal: {2}', [$price, $count
    , $subtotal]);
```

```
// 日本語翻訳: 価格': {0}, 数量: {1}, 小計: {2}'
```

序数プレースホルダが一つだけの場合は、値を配列に入れずにそのまま指定することができます。

```
echo \Yii::t('app', 'Price: {0}', $price);
```

> ヒント: たいていの場合は名前付きプレースホルダを使うべきです。 と言うのは、翻訳者にとっては、パラメータ名がある方が、翻訳すべきメッセージ全体をより良く理解できるからです。

パラメータのフォーマット

メッセージのプレースホルダにフォーマットの規則を追加して指定し、パラメータ値がプレースホルダを置き換える前に適切にフォーマットされるようにすることが出来ます。 次の例では、price のパラメータ値の型は数値として扱われ、通貨の形式でフォーマットされます。

```
$price = 100;
echo \Yii::t('app', 'Price: {0,number,currency}', $price);
```

> 補足: パラメータのフォーマットには、intl PHP 拡張[15] のインストールが必要です。

プレースホルダにフォーマット規則を指定するためには、短い構文または完全な構文のどちらかを使うことが出来ます。

```
短い形式
: {name,type}完全な形式
: {name,type,style}
```

> 補足: {、}、'、# などの特殊な文字を使用する必要がある場合は、その部分の文字列を ' で囲んでください。
>
> ```
> echo Yii::t('app', "Example of string with ''-escaped characters
>     '': '{' '}' '{test}' {count,plural,other{''count'' value is
>     # '#{}'}}", ['count' => 3]);
> +```
> ```

このようなプレースホルダを指定する方法についての完全な説明は、ICU ドキュメント[16]を参照してください。以下では、よくある使用方法をいくつか示します。

[15] https://secure.php.net/manual/ja/intro.intl.php
[16] http://icu-project.org/apiref/icu4c/classMessageFormat.html

```
$sum = 12345;
echo \Yii::t('app', 'Balance: {0,number}', $sum);

// 日本語翻訳: 差引残高': {0,number}'
// 日本語出力: 差引残高': 12,345'
```

オプションのパラメータとして、integer、currency、percent のスタイルを指定することが出来ます。

```
$sum = 12345;
echo \Yii::t('app', 'Balance: {0,number,currency}', $sum);

// 日本語翻訳: 差引残高': {0,number,currency}'
// 日本語出力: 差引残高': ￥12,345'
```

または、数値をフォーマットするカスタム・パターンを指定することも出来ます。

```
$sum = 12345;
echo \Yii::t('app', 'Balance: {0,number,,000,000000}', $sum);

// 日本語翻訳: 差引残高': {0,number,,000,000000}'
// 日本語出力: 差引残高': 000,012345'
```

カスタムフォーマットで使用される文字については、 ICU API リファレンス[17] の “Special Pattern Characters” のセクションに記述されています。

数値は常に翻訳先のロケールに従ってフォーマットされます。 つまり、ロケールを変更せずに、小数点や桁区切りを変更することは出来ません。 それらをカスタマイズしたい場合は yii\i18n\Formatter::asDecimal() や yii\i18n\Formatter::asCurrency() を使うことが出来ます。

日付　パラメータ値は日付としてフォーマットされます。例えば、

```
echo \Yii::t('app', 'Today is {0,date}', time());

// 日本語翻訳: 今日は' {0,date} です。'
// 日本語出力: 今日は' 2015/01/07 です。'
```

オプションのパラメータとして、short、medium、long、そして full のスタイルを指定することが出来ます。

```
echo \Yii::t('app', 'Today is {0,date,short}', time());

// 日本語翻訳: 今日は' {0,date,short} です。'
// 日本語出力: 今日は' 2015/01/07 です。'
```

日付の値をフォーマットするカスタム・パターンを指定することも出来ます。

[17] http://icu-project.org/apiref/icu4c/classicu_1_1DecimalFormat.html

```
echo \Yii::t('app', 'Today is {0,date,yyyy-MM-dd}', time());

// 日本語翻訳: 今日は' {0,date,yyyy-MM-dd} です。'
// 日本語出力: 今日は' 2015-01-07 です。'
```

書式のリファレンス[18].

時刻　パラメータ値は時刻としてフォーマットされます。例えば、

```
echo \Yii::t('app', 'It is {0,time}', time());

// 日本語翻訳: 現在' {0,time} です。'
// 日本語出力: 現在' 22:37:47 です。'
```

オプションのパラメータとして、short、medium、long、そして full のスタイルを指定することが出来ます。

```
echo \Yii::t('app', 'It is {0,time,short}', time());

// 日本語翻訳: 現在' {0,time,short} です。'
// 日本語出力: 現在' 22:37 です。'
```

時刻の値をフォーマットするカスタム・パターンを指定することも出来ます。

```
echo \Yii::t('app', 'It is {0,date,HH:mm}', time());

// 日本語翻訳: 現在' {0,time,HH:mm} です。'
// 日本語出力: 現在' 22:37 です。'
```

書式のリファレンス[19].

綴り　パラメータ値は数値として取り扱われ、綴りとしてフォーマットされます。例えば、

```
// 出力例 : "42 is spelled as forty-two"
echo \Yii::t('app', '{n,number} is spelled as {n,spellout}', ['n' => 42]);

// 日本語翻訳: '{n,number} は、文字で綴ると {n,spellout} です。'
// 日本語出力: '42 は、文字で綴ると 四十二 です。'
```

デフォルトでは、数値は基数として綴られます。それを変更することは可能です。

```
// 出力例 : "I am forty-seventh agent"
echo \Yii::t('app', 'I am {n,spellout,%spellout-ordinal} agent', ['n' =>
    47]);

// 日本語翻訳: 私は'{n,spellout,%spellout-ordinalの工作員です。}'
// 日本語出力: 私は第四十七の工作員です。''
```

[18] http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html#details

[19] http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html#details

‘spellout,’ と ‘

あなたが使用しているロケールで利用可能なオプションのリストについては、 http://intl.rmcreative.ru/[20] の “Numbering schemas, Spellout” を参照してください。

序数 パラメータ値は数値として取り扱われ、順序を表す文字列としてフォーマットされます。例えば、

```
// 出力: "You are the 42nd visitor here!"
echo \Yii::t('app', 'You are the {n,ordinal} visitor here!', ['n' => 42]);
```

序数については、スペイン語などの言語では、さらに多くのフォーマットがサポートされています。

```
// 出力: ª"471"
echo \Yii::t('app', '{n,ordinal,%digits-ordinal-feminine}', ['n' => 471]);
```

‘ordinal,’ と ‘

あなたが使用しているロケールで利用可能なオプションのリストについては、 http://intl.rmcreative.ru/[21] の “Numbering schemas, Ordinal” を参照してください。

> 補足: 上記のソース・メッセージを、プレースホルダのスタイルを守って日本語に翻訳すると、’あなたはこのサイトの{n,ordinal}の訪問者です’ となります。 しかし、その出力結果は、’あなたはこのサイトの第42の訪問者です’ となり、意味は通じますが、日本語としては若干不自然なものになります。
>
> プレースホルダのスタイル自体も、翻訳の対象として、より適切なものに変更することが可能であることに注意してください。
>
> この場合も、’あなたはこのサイトの{n,plural,=1{最初} other{#番目}}の訪問者です’ のように翻訳するほうが適切でしょう。

継続時間 パラメータ値は秒数として取り扱われ、継続時間を表す文字列としてフォーマットされます。例えば、

```
// 出力: "You are here for 47 sec. already!"
echo \Yii::t('app', 'You are here for {n,duration} already!', ['n' => 47]);
```

継続時間については、さらに多くのフォーマットがサポートされています。

```
// 出力: '130:53:47'
echo \Yii::t('app', '{n,duration,%in-numerals}', ['n' => 471227]);
```

[20] http://intl.rmcreative.ru/
[21] http://intl.rmcreative.ru/

‘duration,’ と ‘

あなたが使用しているロケールで利用可能なオプションのリストについては、 http://intl.rmcreative.ru/[22] の “Numbering schemas, Duration” を参照してください。

> 補足: このソース・メッセージを ‘あなたはこのサイトに既に{n,duration}の間滞在しています’ と翻訳した場合の出力結果は、’あなたはこのサイトに既に47の間滞在しています’ となります。 これも、プレースホルダのスタイルも含めて全体を翻訳し直す方が良いでしょう。 どうも、ICU ライブラリは、ja_JP の数値関連の書式指定においては、割と貧弱な実装にとどまっている印象です。

複数形 言語によって、複数形の語形変化はさまざまに異なります。Yii は、さまざまな形式の複数形語形変化に対応したメッセージ翻訳のための便利な方法を提供しています。 それは、非常に複雑な規則に対しても、十分に機能するものです。 語形変化の規則を直接に処理する代りに、特定の状況における語形変化した言葉の翻訳を提供するだけで十分です。

```
// $n = 0 の場合の出力: "There are no cats!"
// $n = 1 の場合の出力: "There is one cat!"
// $n = 42 の場合の出: "There are 42 cats!"
echo \Yii::t('app', 'There {n,plural,=0{are no cats} =1{is one cat} other{
    are # cats}}!', ['n' => $n]);
```

上記の複数形規則の引数において、`=` はぴったりその値であることを意味します。従って、`=0` はぴったりゼロ、`=1` はぴったり 1 を表します。 `other` はそれ以外の数を表します。`#` は ターゲット言語に従ってフォーマットされた `n` の値によって置き換えられます。

複数形の規則が非常に複雑な言語もあります。 例えば、次のロシア語の例では、`=1` が `n = 1` にぴったりと一致するのに対して、`one` が `21` や `101` などに一致します。

```
Здесь
 {n,pluralкотов,=0{ нет} есть=1{ один кот} one{# кот} few{# кота} many{#
    котов} other{# кота}}!
```

これら `other`、`few`、`many` などの特別な引数の名前は言語によって異なります。 特定のロケールに対してどんな引数を指定すべきかを学ぶためには、http://intl.rmcreative.ru/[23] の “Plural Rules, Cardinal” を参照してください。 あるいは、その代りに、unicode.org の規則のリファレンス[24] を参照することも出来ます。

[22] http://intl.rmcreative.ru/
[23] http://intl.rmcreative.ru/
[24] http://cldr.unicode.org/index/cldr-spec/plural-rules

補足: 上記のロシア語のメッセージのサンプルは、主として翻訳メッセージとして使用されるものです。 アプリケーションの ソース言語 を `ru-RU` にしてロシア語から他の言語に翻訳するという設定にしない限り、オリジナルのメッセージとしては使用されることはありません。

`Yii::t()` の呼び出しにおいて、オリジナルのメッセージに対する翻訳が見つからない場合は、 ソース言語 の複数形規則がオリジナルのメッセージに対して適用されます。

文字列が以下のようなものである場合のために `offset` というパラメータがあります。

```
$likeCount = 2;
echo Yii::t('app', 'You {likeCount,plural,
    offset: 1
    =0{did not like this}
    =1{liked this}
    one{and one other person liked this}
    other{and # others liked this}
}', [
    'likeCount' => $likeCount
]);

// 出力: 'You and one other person liked this'
```

補足: 上記のソース・メッセージの日本語翻訳は以下のようなものになります。

'猫は{n, plural, =0{いません} other{#匹います}}。'

日本語では単数形と複数形を区別しませんので、たいていの場合、`=0` と `other` を指定するだけで事足ります。 横着をして、`{n, plural, ...}` を `{n, number}` に置き換えても、多分、大きな問題は生じないでしょう。

序数選択肢 <**span id="ordinal-selection"**> パラメータのタイプとして `selectordinal` を使うと、翻訳先ロケールの言語規則に基づいて序数のための文字列を選択することが出来ます。

```
$n = 3;
echo Yii::t('app', 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd}
    other{#th}} visitor', ['n' => $n]);
// 英語の出力
// You are the 3rd visitor

// ロシア語の翻訳
'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}} visitor
    ' => 'Вы {n, selectordinal, other{#-й}} посетитель',

// ロシア語の出力
// Вы 3-й посетитель
```

フォーマットは複数形で使われるものと非常に近いものです。 特定のロケールに対してどんな引数を指定すべきかを学ぶためには、http://intl.rmcreative.ru/[25] の “Plural Rules, Ordinal” を参照してください。 あるいは、その代りに、unicode.org の規則のリファレンス[26] を参照することも出来ます。

選択肢　パラメータのタイプとして `select` を使うと、パラメータの値に基づいて表現を選択することが出来ます。例えば、

```
// 出力: "Snoopy is a dog and it loves Yii!"
echo \Yii::t('app', '{name} is a {gender} and {gender,select,female{she}
    male{he} other{it}} loves Yii!', [
    'name' => 'Snoopy',
    'gender' => 'dog',
]);
```

上記の式の中で、`female` と `male` が `gender` が取り得る値であり、`other` がそれらに一致しない値を処理します。 それぞれの取り得る値の後には、波括弧で囲んで対応する表現を指定します。

> 補足: 日本語翻訳: ‘{name} は {gender} であり、{gender,select,female{彼女} male{彼} other{それ}}は Yii を愛しています。’
>
> 日本語出力: ‘Snoopy は dog であり、それは Yii を愛しています。’

デフォルトのメッセージ・ソースを指定する

構成されたカテゴリのどれにもマッチしないカテゴリのためのフォールバックとして使用される、デフォルトのメッセージ・ソースを指定することが出来ます。 これは、ワイルドカードのカテゴリ `*` を構成することによって可能になります。 そうするためには、アプリケーションの構成情報に次のように追加します。

```
// i18n コンポーネントを構成する

'i18n' => [
    'translations' => [
        '*' => [
            'class' => 'yii\i18n\PhpMessageSource'
        ],
    ],
],
```

こうすることで、個別に構成することなくカテゴリを使うことが可能になり、Yii 1.1 の振る舞いと同じになります。 カテゴリのメッセージは、デフォルトの翻訳の `basePath` すなわち `@app/messages` の下にあるファイルから読み込まれます。

[25] http://intl.rmcreative.ru/

[26] http://unicode.org/repos/cldr-tmp/trunk/diff/supplemental/language_plural_rules.html

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

この場合、メッセージは `@app/messages/<LanguageCode>/not_specified_category.php` から読み込まれます。

モジュールのメッセージを翻訳する

モジュール用のメッセージを翻訳したいけれども、全てのメッセージに対して一つの翻訳ファイルを使うことは避けたい、という場合には、次のようにすることが出来ます。

```
<?php

namespace app\modules\users;

use Yii;

class Module extends \yii\base\Module
{
    public $controllerNamespace = 'app\modules\users\controllers';

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        Yii::$app->i18n->translations['modules/users/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/modules/users/messages',
            'fileMap' => [
                'modules/users/validation' => 'validation.php',
                'modules/users/form' => 'form.php',
                ...
            ],
        ];
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('modules/users/' . $category, $message, $params,
    $language);
    }

}
```

上記の例では、マッチングのためにワイルドカードを使い、次に必要なファイルごとに各カテゴリをフィルタリングしています。 `fileMap` を使わずに、カテゴリを同じ名前のファイルにマップする規約を使って済ま

せることも出来ます。 以上のようにすれば、直接に `Module::t('validation', 'your custom validation message')` や `Module::t('form', 'some form label')` などを使用することが出来ます。

ウィジェットのメッセージを翻訳する

モジュールに適用できる同じ規則をウィジェットにも適用することが出来ます。例えば、

```
<?php

namespace app\widgets\menu;

use yii\base\Widget;
use Yii;

class Menu extends Widget
{

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        $i18n = Yii::$app->i18n;
        $i18n->translations['widgets/menu/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/widgets/menu/messages',
            'fileMap' => [
                'widgets/menu/messages' => 'messages.php',
            ],
        ];
    }

    public function run()
    {
        echo $this->render('index');
    }

    public static function t($category, $message, $params = [], $language =
    null)
    {
        return Yii::t('widgets/menu/' . $category, $message, $params,
    $language);
    }

}
```

`fileMap` を使わずに、カテゴリを同じ名前のファイルにマップする規約

を使って済ませることも出来ます。 これで、直接に `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` を使用することが出来ます。

> 補足: ウィジェットのためには i18n ビューも使うことが出来ます。コントローラのための同じ規則がウィジェットにも適用されます。

フレームワーク・メッセージを翻訳する

Yii には、検証エラーとその他いくつかの文字列に対するデフォルトの翻訳メッセージが付属しています。これらのメッセージは、全て 'yii' というカテゴリの中にあります。 場合によっては、あなたのアプリケーションのために、デフォルトのフレームワーク・メッセージの翻訳を修正したいことがあるでしょう。 そうするためには、`i18n` アプリケーション・コンポーネント を以下のように構成してください。

```
'i18n' => [
    'translations' => [
        'yii' => [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/messages'
        ],
    ],
],
```

これで、あなたの修正した翻訳を `@app/messages/<language>/yii.php` に置くことが出来ます。

欠落している翻訳の処理

ソースに翻訳が欠落している場合でも、Yii はリクエストされたメッセージの内容を表示します。 この振舞いは、原文のメッセージが正当かつ詳細なテキストである場合には、非常に好都合です。しかし、場合によっては、それだけでは十分ではありません。 リクエストされた翻訳がソースに欠落しているときに、何らかの特別な処理を実行する必要がある場合もあります。 そういう処理は、`yii\i18n\MessageSource` の `missingTranslation` イベントを使うことによって達成できます。

例えば、全ての欠落している翻訳に何か目立つマークを付けて、ページに表示されたときに簡単に見つけられるようにしましょう。 最初にイベント・ハンドラをセットアップする必要があります。これはアプリケーションの構成によって行うことが出来ます。

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
```

```
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
                'on missingTranslation' => ['app\components\
    TranslationEventHandler', 'handleMissingTranslation']
            ],
        ],
    ],
],
```

次に、私たち独自のイベント・ハンドラを実装する必要があります。

```
<?php

namespace app\components;

use yii\i18n\MissingTranslationEvent;

class TranslationEventHandler
{
    public static function handleMissingTranslation(MissingTranslationEvent
    $event)
    {
        $event->translatedMessage = "@MISSING: {$event->category}.{$event->
    message} FOR LANGUAGE {$event->language} @";
    }
}
```

このイベント・ハンドラによって yii\i18n\MissingTranslationEvent::$translatedMessage がセットされた場合は、それが翻訳結果として表示されます。

> 補足: 全てのメッセージ・ソースは、欠落した翻訳をそれぞれ独自に処理します。 いくつかのメッセージ・ソースを使っていて、それらが同じ方法で欠落した翻訳を取り扱うようにしたい場合は、対応するイベント・ハンドラを全てのメッセージ・ソースそれぞれに割り当てなければなりません。

message コマンドを使う

翻訳は php ファイル、.po ファイル、または database に保存することが出来ます。追加のオプションについてはそれぞれのクラスを参照してください。

まず最初に、構成情報ファイルを作成する必要があります。 どこに保存したいかを決めて、次のコマンドを発行してください。

```
./yii message/config-template path/to/config.php
```

作成されたファイルを開いて、あなたの要求に合わせてパラメータを修正します。特に、下記の項目に注意を払ってください。

- languages: あなたのアプリケーションが翻訳されるべき言語を表す配列。
- messagePath: メッセージファイルを保存するパス。これは、アプリケーションの構成情報で記述されている i18n の basePath と合致しなければなりません。

'./yii message/config' コマンドを使って、CLI 経由で、指定したオプションを持つ設定ファイルを動的に生成することも可能です。 例えば、languages と messagePath のパラメータは、次のようにして設定することが出来ます。

```
./yii message/config --languages=de,ja --messagePath=messages path/to/config
    .php
```

利用可能なオプションのリストを取得するためには、次のコマンドを実行します。

```
./yii help message/config
```

構成情報ファイルの編集が完了すれば、ついに、下記のコマンドを使ってメッセージを抽出することが出来ます。

```
./yii message path/to/config.php
```

また、オプションを指定して、抽出のパラメータを動的に変更することも出来ます。

これで、(あなたがファイル・ベースの翻訳を選択していた場合は) messagePath ディレクトリにファイルが出現します。

14.4.4 ビューの翻訳

個々のテキスト・メッセージを翻訳する代りに、ビュー・スクリプト全体を翻訳したい場合があるでしょう。 この目的を達するためには、ビューを翻訳して、ターゲット言語と同じ名前のサブ・ディレクトリに保存するだけで大丈夫です。 例えば、views/site/index.php というビューをターゲット言語 ru-RU に翻訳したい場合は、翻訳したビューを views/site/ru-RU/index.php というファイルとして保存します。 このようにすると、yii\base\View::renderFile() メソッド、または、このメソッドを呼び出す他のメソッド (例えば yii\base\Controller::render()) を呼んで views/site/index.php をレンダリングするたびに、 翻訳された views/site/ru-RU/index.php が代りにレンダリングされるようになります。

> 補足: ターゲット言語 が ソース言語 と同じ場合は、 翻訳されたビューの有無にかかわらず、オリジナルのビューがレンダリングされます。

14.4.5 数値と日付の値を書式設定する

詳細は データ・フォーマッタ のセクションを参照してください。

14.4.6 日付と数値をフォーマットする

詳細は データのフォーマット のセクションを参照して下さい。

14.4.7 PHP 環境をセットアップする

Yii は、yii\i18n\Formatter クラスの数値や日付の書式設定や、yii\i18n\MessageFormatter を使うメッセージのフォーマッティングなど、ほとんどの国際化機能を提供するために PHP intl 拡張[27] を使います。この二つのクラスは、intl がインストールされていない場合に備えて基本的な機能を提供するフォールバックを実装しています。 だだし、このフォールバックの実装は、英語がターゲット言語である場合にのみ十分に機能するものです。 従って、国際化機能が必要とされる場合は、intl をインストールすることが強く推奨されます。

PHP intl 拡張[28] は、さまざまに異なる全てのロケールについて知識と書式の規則を提供する ICU ライブラリ[29] に基礎を置いています。 ICU のバージョンが異なると、日付や数値のフォーマットの結果も異なる場合があります。 あなたのウェブ・サイトが全ての環境で同じ出力をすることを保証するためには、 全ての環境において同じバージョンの PHP intl 拡張 (従って同じバージョンの ICU) をインストールすることが推奨されます。

どのバージョンの ICU が PHP によって使われているかを知るために、次のスクリプトを走らせることが出来ます。このスクリプトは、使用されている PHP と ICU のバージョンを出力します。

```
<?php
echo "PHP: " . PHP_VERSION . "\n";
echo "ICU: " . INTL_ICU_VERSION . "\n";
echo "ICU Data: " . INTL_ICU_DATA_VERSION . "\n";
```

さらに、バージョン 49 以上の ICU を使用する事も推奨されます。そうすることによって、このドキュメントで説明されている全ての機能を使うことが出来るようになります。 例えば、49 未満の ICU は、複数形規則における # プレースホルダをサポートしていません。 入手できる ICU バージョン については、http://site.icu-project.org/download を参照してください。 バージョン番号の採番方式が 4.8 リリースの後に変更されたことに注意してください (すなわち、ICU 4.8、ICU 49、ICU 50、等々となっています)。

これに加えて、ICU ライブラリとともに出荷されるタイム・ゾーン・データベースの情報も古くなっている可能性があります。 タイム・ゾーン・データベースの更新に関する詳細は ICU マニュアル[30] を参照してください。 出力の書式設定には ICU タイム・ゾーン・データベースが

[27] https://secure.php.net/manual/ja/book.intl.php
[28] https://secure.php.net/manual/ja/book.intl.php
[29] http://site.icu-project.org/
[30] http://userguide.icu-project.org/datetime/timezone#TOC-Updating-the-Time-Zone-Data

使用されますが、PHP によって使われるタイム・ゾーン・データベースも影響する可能性があります。 PHP のタイム・ゾーン・データベースは、`timezonedb` pecl パッケージ[31] の最新版をインストールすることによって更新することが出来ます。

14.5 メール送信

補足: このセクションはまだ執筆中です。

Yii は電子メールの作成と送信をサポートしています。 ただし、フレームワークのコアが提供するのは、コンテント作成の機能と基本的なインタフェイスだけです。 実際のメール送信メカニズムはエクステンションによって提供されなければなりません。 と言うのは、メール送信はプロジェクトが異なるごとに異なる実装が必要とされるでしょうし、通常、外部のサービスやライブラリに依存するものだからです。

ごく一般的な場合であれば、yii2-swiftmailer[32] 公式エクステンションを使用することが出来ます。

14.5.1 構成

メール・コンポーネントの構成は、あなたが選んだエクステンションに依存します。 一般的には、アプリケーションの構成情報は次のようなものになる筈です。

```
return [
    //....
    'components' => [
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
            'useFileTransport' => false,
            'transport' => [
                'class' => 'Swift_SmtpTransport',
                'encryption' => 'tls',
                'host' => 'your_mail_server_host',
                'port' => 'your_smtp_port',
                'username' => 'your_username',
                'password' => 'your_password',
            ],
        ],
    ],
];
```

14.5.2 基本的な使用方法

いったん `mailer` コンポーネントを構成すれば、次のコードを使って電子メールのメッセージを送信することが出来るようになります。

[31] https://pecl.php.net/package/timezonedb
[32] https://www.yiiframework.com/extension/yiisoft/yii2-swiftmailer

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('メッセージの題')
    ->setTextBody('プレーンテキストのコンテント')
    ->setHtmlBody('<b>HTML のコンテント</b>')
    ->send();
```

上の例では、compose() メソッドでメール・メッセージのインスタンスを作成し、それに値を投入して送信しています。 必要であれば、このプロセスにもっと複雑なロジックを置くことも可能です。

```
$message = Yii::$app->mailer->compose();
if (Yii::$app->user->isGuest) {
    $message->setFrom('from@domain.com');
} else {
    $message->setFrom(Yii::$app->user->identity->email);
}
$message->setTo(Yii::$app->params['adminEmail'])
    ->setSubject('メッセージの題')
    ->setTextBody('プレーン・テキストのコンテント')
    ->send();
```

> 補足: すべての mailer エクステンションは、二つの主要なクラス、すなわち、Mailer と Message のセットとして提供されます。 Mailer は常に Message のクラス名と仕様を知っています。 Message オブジェクトのインスタンスを直接に作成しようとしてはいけません。常に compose() メソッドを使って作成してください。

いくつかのメッセージを一度に送信することも出来ます。

```
$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);
```

メール・エクステンションの中には、単一のネットワーク・メッセージを使うなどして、この手法の恩恵を享受することが出来るものもいくつかあるでしょう。

14.5.3 メールのコンテントを作成する

Yii は実際のメール・メッセージを特別なビュー・ファイルによって作成することを許容しています。 デフォルトでは、それらのファイルは @app/mail というパスに配置されなければなりません。

以下はメール・ビュー・ファイルの内容の例です。

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;

/* @var $this \yii\web\View ビュー・コンポーネントのインスタンス */
/* @var $message \yii\mail\BaseMessage 新しく作成されたメール・メッセージのイ
    ンスタンス */

?>
<h2ワン・クリックで私たちのサイトのホームページを訪問することが出来ます></h2>
<?= Html::a('ホームページへ', Url::home('http')) ?>
```

ビュー・ファイルによってメッセージを作成するためには、単に compose() メソッドにビューの名前を渡すだけで十分です。

```
Yii::$app->mailer->compose('home-link') // ここでビューのレンダリング結果が
    メッセージのボディになります
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('メッセージの題')
    ->send();
```

ビュー・ファイルの中で利用できる追加のビュー・パラメータを compose() メソッドに渡すことができます。

```
Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

HTML と平文テキストのメッセージ・コンテントに違うビューを指定することが出来ます。

```
Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);
```

ビュー名をスカラーの文字列として渡した場合は、そのレンダリング結果は HTML ボディとして使われます。 そして、平文テキストのボディは HTML のボディから全ての HTML 要素を削除することによって作成されます。

ビューのレンダリング結果はレイアウトで包むことが出来ます。レイアウトは、yii\mail\BaseMailer::$htmlLayout と yii\mail\BaseMailer::$textLayout を使ってセットアップすることが可能です。 レイアウトは、通常のウェブ・アプリケーションのレイアウトと同じように働きます。 レイアウトは、メールの CSS スタイルや、その他の共有されるコンテントをセットアップするために使うことが出来ます。

```
<?php
use yii\helpers\Html;

/* @var $this \yii\web\View ビュー・コンポーネントのインスタンス */
```

```
/* @var $message \yii\mail\MessageInterface 作成されるメッセージ */
/* @var $content string メイン・ビューのレンダリング結果 */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/
    TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?= Yii::
    $app->charset ?>" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">よろしくお願いします。><?= Yii::$app->name ?> チー
    ム</div>
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

14.5.4 ファイルの添付

attach() メソッド、attachContent() メソッドを使って、メッセージにファイルを添付することが出来ます。

```
$message = Yii::$app->mailer->compose();

// ローカル・ファイル・システムからファイルを添付する
$message->attach('/path/to/source/file.pdf');

// 添付ファイルをその場で生成する
$message->attachContent('添付される内
    容', ['fileName' => 'attach.txt', 'contentType' => 'text/plain']);
```

14.5.5 画像の埋め込み

embed() メソッドを使って、メッセージのコンテントに画像を埋め込むことが出来ます。 このメソッドは添付ファイルの ID を返しますので、それを img タグで使わなければなりません。 このメソッドはビュー・ファイルによってメッセージのコンテントを作成するときに簡単に使うことが出来ます。

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' => '/path/to/
    image.jpg'])
```

```
    // ...
    ->send();
```

そして、ビュー・ファイルの中では、次のコードを使うことが出来ます。

```
<img src="<?= $message->embed($imageFileName); ?>">
```

14.5.6 テストとデバッグ

開発者は、実際にどのようなメールがアプリケーションによって送信されたか、その内容はどのようなものであったか、等をチェックしなければならないことが多くあります。 Yii は、そのようなチェックが出来ることを yii\mail\BaseMailer::useFileTransport によって保証しています。このオプションを有効にすると、メールのメッセージ・データが、通常のように送信される代りに、ローカル・ファイルに強制的に保存されます。 ファイルは、yii\mail\BaseMailer::fileTransportPath、デフォルトでは @runtime/mail の下に保存されます。

> 補足: メッセージをファイルに保存するか、実際の受信者に送信するか、どちらかを選ぶことが出来ますが、両方を同時に実行することは出来ません。

メール・メッセージのファイルは通常のテキストエディタで開くことが出来ますので、実際のメッセージ・ヘッダやコンテントなどを閲覧することが出来ます。 このメカニズムは、アプリケーションのデバッグや単体テストを実行する際に、真価を発揮するでしょう。

> 補足: メール・メッセージのファイルの内容は \yii\mail\MessageInterface::toString() によって作成されますので、 あなたのアプリケーションで使用している実際のメール・エクステンションに依存したものになります。

14.5.7 あなた自身のメール・ソリューションを作成する

あなた自身のカスタム・メール・ソリューションを作成するためには、二つのクラスを作成する必要があります。 すなわち、一つは Mailer であり、もう一つは Message です。 yii\mail\BaseMailer と yii\mail\BaseMessage をあなたのソリューションの基底クラスとして使うことが出来ます。 これらのクラスが、このガイドで説明された基本的なロジックを既に持っています。しかし、それを使用することは強制されていません。 yii\mail\MailerInterface と yii\mail\MessageInterface のインタフェイスを実装すれば十分です。 そして、あなたのソリューションをビルドするために、全ての抽象メソッドを実装しなければなりません。

14.6 パフォーマンス・チューニング

あなたのウェブ・アプリケーションのパフォーマンスに影響を及ぼす要因は数多くあります。 環境の要因もありますし、あなたのコードに関係する要因もあります。また、Yii そのものに関係する要因もあります。 このセクションでは要因のほとんどを列挙して、どのようにそれらを修正すればあなたのアプリケーションのパフォーマンスを向上させることが出来るかを説明します。

14.6.1 PHP 環境を最適化する

PHP 環境を正しく構成することは非常に重要です。最大のパフォーマンスを得るためには、

- 最新の安定した PHP バージョンを使うこと。使用する PHP のメジャー・リリースを上げると、顕著なパフォーマンスの改善がもたらされることがあります。
- Opcache[33] (PHP 5.5 以降) または APC[34] (PHP 5.4) を使って、 バイト・コード・キャッシュを有効にすること。 バイト・コード・キャッシュによって、リクエストが入ってくるたびに PHP スクリプトを解析してインクルードする時間の浪費を避けることが出来ます。
- `realpath()` キャッシュをチューニングする[35].

14.6.2 デバッグ・モードを無効にする

本番環境でアプリケーションを実行するときには、デバッグ・モードを無効にしなければなりません。 Yii は、`YII_DEBUG` という名前の定数の値を使って、デバッグ・モードを有効にすべきか否かを示します。 デバッグ・モードが有効になっているときは、Yii はデバッグ情報の生成と記録のために時間を余計に費やします。

エントリ・スクリプト の冒頭に次のコード行を置くことによってデバッグ・モードを無効にすることが出来ます。

```
defined('YII_DEBUG') or define('YII_DEBUG', false);
```

> 情報: `YII_DEBUG` のデフォルト値は `false` です。 従って、アプリケーション・コードの他のどこかでこのデフォルト値を変更していないと確信できるなら、単に上記の行を削除してデバッグ・モードを無効にしても構いません。

[33] https://secure.php.net/opcache
[34] https://secure.php.net/apc
[35] https://github.com/samdark/realpath_cache_tuner

14.6.3 キャッシュのテクニックを使う

さまざまなキャッシュのテクニックを使うと、あなたのアプリケーションのパフォーマンスを目に見えて改善することが出来ます。 たとえば、あなたのアプリケーションが Markdown 形式のテキスト入力をユーザに許可している場合、解析済みの Markdown のコンテントをキャッシュすることを考慮してください。 そうすれば、リクエストごとに毎回同じ Markdown テキストの解析を繰り返すことを回避できるでしょう。 Yii によって提供されているキャッシュのサポートについて学ぶためには キャッシュ のセクションを参照してください。

14.6.4 スキーマ・キャッシュを有効にする

スキーマ・キャッシュは、アクティブ・レコード を使おうとする場合には、いつでも有効にすべき特別なキャッシュ機能です。 ご存じのように、アクティブ・レコードは、賢いことに、あなたがわざわざ記述しなくても、 DB テーブルに関するスキーマ情報 (カラムの名前、カラムのタイプ、外部キー制約など) を自動的に検出します。 アクティブ・レコードはこの情報を取得するために追加の SQL クエリを実行しています。 スキーマ・キャッシュを有効にすると、取得されたスキーマ情報はキャッシュに保存されて将来のクエリで再利用されるようになります。

スキーマ・キャッシュを有効にするためには、アプリケーションの構成情報 の中で、 `cache` アプリケーション・コンポーネント にスキーマ情報を保存するように構成し、`yii\db\Connection::$enableSchemaCache` を `true` に設定します。

```
return [
    // ...
    'components' => [
        // ...
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase',
            'username' => 'root',
            'password' => '',
            'enableSchemaCache' => true,

            // スキーマ・キャッシュの持続時間
            'schemaCacheDuration' => 3600,

            // スキーマ情報を保存するのに使用されるキャッシュ・コンポーネントの
    名前
            'schemaCache' => 'cache',
        ],
    ],
];
```

14.6.5 アセットを結合して最小化する

複雑なウェブ・ページでは、多数の CSS や JavaScript のアセット・ファイルをインクルードすることがよくあります。 HTTP リクエストの回数、および、これらのアセットの全体としてのダウンロード・サイズを削減するために、アセットを単一のファイルに結合して、それを圧縮することを考慮すべきです。 これによって、ページのロードにかかる時間とサーバの負荷を大きく削減することが出来ます。 詳細については、アセット のセクションを参照してください。

14.6.6 セッションのストレージを最適化する

デフォルトでは、セッションのデータはファイルに保存されます。 これは、session_write_close() が呼ばれる (Yii では Yii::$app->session->close() によって呼び出されます) か、あるいはリクエストの処理が終了して、セッションが閉じられる時点まで、ファイルが開かれるのをロックするという実装になっています。 セッション・ファイルがロックされている間は、同じセッションを使用しようとする全てのリクエストはブロックされ、最初のリクエストがセッション・ファイルを解放するのを待たなければなりません。 開発時はこれでも構いません。おそらく、小さなプロジェクトでも、これで大丈夫でしょう。 しかし、大量のリクエストを並列処理するとなると、データベースのような、もっと洗練されたストレージを使う方が良いでしょう。 Yii はさまざまなセッション・ストレージのサポートを内蔵しています。 これらのストレージは、アプリケーションの構成情報 の中で session コンポーネントを次のように構成することによって使用することが出来ます。

```
return [
    // ...
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',

            // デフォルトの 'db' 以外の DB コンポーネントを使用したい場合は
            // 以下を設定する
            // 'db' => 'mydb',

            // デフォルトの session テーブルをオーバーライドするためには以下を
    設定する
            // 'sessionTable' => 'my_session',
        ],
    ],
];
```

上記の構成は、セッション・データの保存にデータベース・テーブルを使用するものです。 デフォルトでは、db アプリケーション・コンポーネントをデータベース接続として使用し、セッション・データを session テーブルに保存します。 ただし、前もって session テーブルを次のよう

に作っておく必要があります。

```
CREATE TABLE session (
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

yii\web\CacheSession を使って、セッションをキャッシュに保存することも出来ます。 理論上、サポートされている キャッシュ・ストレージ のどれでも使うことが出来ます。 ただし、キャッシュ・ストレージの中には、容量の上限に達したときにキャッシュされたデータをフラッシュするものがあることに注意してください。 この理由により、主として容量の上限が無い種類のキャッシュ・ストレージを使用すべきです。

あなたのサーバに Redis[36] がある場合は、yii\redis\Session によって redis をセッション・ストレージとして使用することを強く推奨します。

14.6.7 データベースを最適化する

DB クエリの実行とデータベースからのデータ取得がウェブ・アプリケーションのパフォーマンスの主たるボトルネックになることがよくあります。 データ・キャッシュ の使用によってパフォーマンスの劣化を緩和することは出来ますが、問題を完全に解決することは出来ません。 データベースが膨大なデータを抱えている場合、キャッシュされたデータが無効化されたときに最新のデータを取得するためのコストは、 データベースとクエリが適切に設計されていないと、法外なものになり得ます。

DB クエリのパフォーマンスを向上させるための一般的なテクニックは、フィルタの対象になるテーブル・カラムにインデックスを作成することです。 例えば、username によってユーザのレコードを検索する必要があるなら、username に対してインデックスを作成するべきです。 ただし、インデックスを付けると SELECT クエリを非常に速くすることが出来る代りに、INSERT、UPDATE、または DELTE のクエリが遅くなることに注意してください。

複雑な DB クエリについては、クエリの解析と準備の時間を節約するために、データベース・ビューを作成することが推奨されます。

最後にもう一つ大事なことですが、SELECT クエリで LIMIT を使ってください。 こうすることで、大量のデータが返されて、PHP のために確保されたメモリを使い尽くすということがなくなります。

14.6.8 プレーンな配列を使う

アクティブ・レコード は非常に使い勝手のよいものですが、データベースから大量のデータを取得する必要がある場合は、プレーンな配列を使うほどには効率的ではありません。 そういう場合は、アクティブ・レ

[36] http://redis.io/

コードを使ってデータを取得する際に `asArray()` を呼んで、 取得したデータがかさばるアクティブ・レコードのオブジェクトではなく配列として表現されるようにすることを考慮するのが良いでしょう。 例えば、

```
class PostController extends Controller
{
    public function actionIndex()
    {
        $posts = Post::find()->limit(100)->asArray()->all();

        return $this->render('index', ['posts' => $posts]);
    }
}
```

上記において、`$posts` は、テーブル行の配列としてデータを代入されることになります。各行はプレーンな配列になります。 `$i` 番目の行の `title` カラムにアクセスするためには、`$posts[$i]['title']` という式を使うことが出来ます。

クエリを構築するのに DAO を使って、データをプレーンな配列に取得することも出来ます。

14.6.9 Composer オートローダを最適化する

Composer のオートローダは、ほとんどのサードパーティのクラス・ファイルをインクルードするのに使われますので、 次のコマンドを実行して Composer のオートローダを最適化することを考慮すべきです。

```
composer dumpautoload -o
```

さらに加えて、 authoritative class maps[37] および APCu cache[38] の使用を検討して下さい。 ただし、この二つの最適化があなたの特定のケースに適切である場合もあれば、そうでない場合もあります。

14.6.10 オフラインでデータを処理する

リクエストが何らかのリソース集約的な操作を必要とするものである場合は、そういう操作が終るまでユーザを待たせずに、 オフラインモードで操作を実行する方策を考えるべきです。

オフラインでデータを処理するための方法が二つあります。すなわち、プルとプッシュです。

プルの方法では、リクエストが何らかの複雑な操作を必要とするたびに、タスクを作成してデータベースなどの持続的ストレージに保存します。 そうしておいて、別の独立したプロセス (例えばクロンジョブ) を使い、タスクを引き出して処理します。 この方法は、実装は容易ですが、いくつかの欠点があります。 例えば、タスクのプロセスはストレージか

[37] https://getcomposer.org/doc/articles/autoloader-optimization.md#optimization-level-2-a-authoritative-class-maps

[38] https://getcomposer.org/doc/articles/autoloader-optimization.md#optimization-level-2-b-apcu-cache

ら定期的にタスクを引き出さなければなりません。 引き出す間隔が長すぎると、タスクの処理に大きな遅延が生じます。しかし、間隔が短すぎると、オーバーヘッドが大きくなります。

プッシュの方法では、タスクを管理するのにメッセージ・キュー (例えば、RabbitMQ、ActiveMQ、Amazon SQS など) を使用します。 新しいタスクがキューに入れられるたびに、タスクを処理するプロセスが起動されたり通知を受けたりして、タスク処理がトリガされます。

14.6.11 パフォーマンス・プロファイリング

あなたは、あなたのコードをプロファイルして、パフォーマンスのボトルネックを発見し、それに応じた適切な手段を講じるべきです。 次のプロファイリング・ツールが役に立つでしょう。

- Yii のデバッグ・ツールバーとデバッガ[39]
- Blackfire[40]
- XHProf[41]
- XDebug プロファイラ[42]

14.6.12 アプリケーションをスケーラブルなものにする覚悟を決める

何をやっても助けにならないときは、あなたのアプリケーションをスケーラブルにすることを試みましょう。良い導入記事が Configuring a Yii 2 Application for an Autoscaling Stack (Yii 2 アプリケーションを自動スケール環境のために構成する)[43] の中で提供されています。

14.7 共有ホスティング環境

共有ホスティング環境では、たいてい、構成やディレクトリ構造について大きな制約があります。 それでも、ほとんどの場合、少し調整をすれば、Yii 2.0 を共有ホスティング環境で走らせることが可能です。

14.7.1 ベーシック・プロジェクト・テンプレートを配備する

通例、共有ホスティング環境では、一つのウェブ・ルートしかありませんので、可能であればベーシック・プロジェクト・テンプレートを使用して下さい。 まず、Yii をインストールする のセクションを参照して、プロジェクト・テンプレートをローカル環境にインストールします。 そ

[39] https://github.com/yiisoft/yii2-debug/blob/master/docs/guide-ja/README.md
[40] https://blackfire.io/
[41] https://secure.php.net/manual/ja/book.xhprof.php
[42] http://xdebug.org/docs/profiler
[43] https://github.com/samdark/yii2-cookbook/blob/master/book/scaling.md

して、ローカル環境でアプリケーションが動くようにした後で、共有ホスティング環境でホスト出来るようにいくつかの修正を行います。

ウェブ・ルートの名前を変える

FTP またはその他の手段であなたの共有ホストに接続します。おそらく、下記のようなディレクトリが見えるでしょう。

```
config
logs
www
```

上記で `www` はウェブ・サーバのウェブ・ルート・ディレクトリです。別の名前かもしれません。よくある名前は、`www`、`htdocs`、`public_html` です。

私たちのベーシック・プロジェクト・テンプレートではウェブ・ルートの名前は `web` になっています。 あなたのウェブ・サーバにアップロードする前に、ローカルのウェブ・ルートの名前をあなたのサーバに適合するように変更します。 すなわち、`web` から `www` や `public_html` など、何であれ、あなたの共有ホストのウェブ・ルートの名前に変更します。

FTP ルート・ディレクトリは書き込み可能

ルート・レベルのディレクトリ、すなわち、`config`、`logs`、`www` があるディレクトリに対して書き込みが出来るのであれば、 `assets`、`commands` などをそのままルート・レベルのディレクトリにアップロードします。

ウェブ・サーバのための追加設定

使用されているウェブ・サーバが Apache である場合は、次の内容を持つ `.htaccess` ファイルを `web` (または `public_html` など、要するに、`index.php` があるディレクトリ) に追加する必要があります。

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on

# ディレクトリかファイルが存在すれば、それを直接使う
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# それ以外は、index.php にフォワードする
RewriteRule . index.php
```

nginx の場合は、追加の構成ファイルは必要がない筈です。

必要条件をチェックする

Yii を走らせるためには、あなたのウェブ・サーバは Yii の必要条件を満たさなければなりません。最低限の必要条件は PHP 5.4 です。 必要条件をチェックするために、`requirements.php` をルート・ディレクトリからウェブ・ルート・ディレクトリにコピーして、 `http://example.com/requirements.php` という URL を使ってブラウザ経由で走らせます。後でファイルを削除するのを忘れないでください。

14.7.2 アドバンスト・プロジェクト・テンプレートを配備する

アドバンスト・プロジェクト・テンプレートを共有ホストに配備することは、ベーシック・プロジェクト・テンプレートを配備するのに比べると少しトリッキーにはなりますが、可能です。 アドバンスト・プロジェクト・テンプレートのドキュメント[44] で説明されている指示に従って下さい。

14.8 テンプレートエンジンを使う

デフォルトでは、Yii は PHP をテンプレート言語として使いますが、Twig[45] や Smarty[46] などの他のレンダリング・エンジンをサポートするように Yii を構成することが出来ます。

`view` コンポーネントがビューのレンダリングに責任を持っています。このコンポーネントのビヘイビアを構成することによって、カスタム・テンプレート・エンジンを追加することが出来ます。

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // twig のオプションの配列
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
```

[44] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-ja/topic-shared-hosting.md

[45] http://twig.sensiolabs.org/

[46] http://www.smarty.net/

```
                ],
                // ...
            ],
        ],
    ],
]
```

上記のコードにおいては、Smarty と Twig の両者がビュー・ファイルによって使用可能なものとして構成されています。 しかし、これらのエクステンションをプロジェクトで使うためには、`composer.json` ファイルも修正して、これらのエクステンションを含める必要があります。

```
"yiisoft/yii2-smarty": "~2.0.0",
"yiisoft/yii2-twig": "~2.0.0",
```

上のコードを `composer.json` の `require` セクションに追加します。変更をファイルに保存した後、コマンドラインで `composer update --prefer-dist` を実行することによってエクステンションをインストールすることが出来ます。

具体的にテンプレート・エンジンを使用する方法については、それぞれのドキュメントで詳細を参照してください。

- Twig ガイド[47]
- Smarty ガイド[48]

14.9 サードパーティのコードを扱う

時々、Yii アプリケーションの中でサードパーティのコードを使用する必要があることがあります。 あるいは、サードパーティのシステムの中で Yii をライブラリとして使用したいこともあるでしょう。このセクションでは、こういう目的をどうやって達成するかを説明します。

14.9.1 Yii の中でサードパーティのライブラリを使う

Yii アプリケーションの中でサードパーティのライブラリを使うために主として必要なことは、 そのライブラリのクラスが適切にインクルードされること、または、オートロード可能であることを保証することです。

Composer パッケージを使う

多くのサードパーティ・ライブラリは Composer[49] パッケージの形式でリリースされています。 そのようなライブラリは、次の二つの簡単なステップを踏むことによって、インストールすることが出来ます。

1. アプリケーションの `composer.json` ファイルを修正して、どの Composer パッケージをインストールしたいかを指定する。

[47] https://www.yiiframework.com/extension/yiisoft/yii2-twig/doc/guide/
[48] https://www.yiiframework.com/extension/yiisoft/yii2-smarty/doc/guide/
[49] https://getcomposer.org/

2. `composer install` を実行して、指定したパッケージをインストールする。

インストールされた Composer パッケージ内のクラスは、Composer のオートローダを使ってオートロードすることが出来ます。 アプリケーションの エントリ・スクリプト に、 Composer のオートローダをインストールするための下記の行があることを確認してください。

```
// Composer のオートローダをインストール
require __DIR__ . '/../vendor/autoload.php';

// Yii クラス・ファイルをインクルード
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';
```

ダウンロードしたライブラリを使う

ライブラリが Composer パッケージとしてリリースされていない場合は、そのライブラリのインストールの指示に従ってインストールしなければなりません。 たいていの場合は、リリース・ファイルを手動でダウンロードし、`BasePath/vendor` ディレクトリの下に解凍する必要があります。ここで `BasePath` は、アプリケーションの base path を表すものです。

ライブラリがそれ自身のオートローダを持っている場合は、それをアプリケーションの エントリ・スクリプト でインストールすることが出来ます。 複数のオートローダ・クラスの中で Yii のクラス・オートローダが優先されるように、 ライブラリのオートローダは `Yii.php` ファイルをインクルードする前にインストールすることを推奨します。

ライブラリがクラスオートローダを提供していない場合でも、クラスの命名規約が PSR-4[50] に従っている場合は、ライブラリのクラスをオートロードするのに Yii のクラス・オートローダを使うことが出来ます。必要なことは、ライブラリのクラスによって使われている全てのルート名前空間に対して ルート・エイリアス を宣言することだけです。 例えば、ライブラリを `vendor/foo/bar` ディレクトリの下にインストールしたとしましょう。 そしてライブラリのクラスは `xyz` ルート名前空間の下にあるとします。 この場合、アプリケーションの構成情報において、次のコードを含めれば良いのです。

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

上記のどちらにも当てはまらない場合、おそらくそのライブラリは、クラス・ファイルを探して適切にインクルードするために、PHP の include path 設定に依存しているのでしょう。 この場合は、PHP include path の設定に関するライブラリの指示に従うしかありません。

[50] http://www.php-fig.org/psr/psr-4/

最悪の場合として、ライブラリが全てのクラス・ファイルを明示的にインクルードすることを要求している場合は、 次の方法を使ってクラスを必要に応じてインクルードすることが出来るようになります。

- ライブラリに含まれるクラスを特定する。
- アプリケーションの エントリ・スクリプト において、 クラスと対応するファイル・パスを `Yii::$classMap` としてリストアップする。例えば、

```
Yii::$classMap['Class1'] = 'path/to/Class1.php';
Yii::$classMap['Class2'] = 'path/to/Class2.php';
```

14.9.2 サードパーティのシステムで Yii を使う

Yii は数多くの優れた機能を提供していますので、サードパーティのシステム (例えば、WordPress、Joomla、または、他の PHP フレームワークを使って開発されるアプリケーション) を開発したり機能拡張したりするのをサポートするために Yii の機能のいくつかを使用したいことがあるでしょう。 例えば、`yii\helpers\ArrayHelper` クラスや アクティブ・レコード をサードパーティのシステムで使いたいことがあるでしょう。 この目的を達するためには、主として、二つのステップを踏む必要があります。 すなわち、Yii のインストールと、Yii のブートストラップです。

サードパーティのシステムが Composer を使って依存を管理している場合は、 単に下記のコマンドを実行すれば Yii をインストールすることが出来ます。

```
composer require yiisoft/yii2
```

データベース抽象レイヤなど、アセットに関係しない Yii の機能だけを使用したい場合は、 Bower および NPM のパッケージのインストールを阻止する特別な composer パッケージが必要になります。 詳細については cebe/assetfree-yii2[51] を参照して下さい。

Composer に関する更なる情報や、インストールの過程で出現しうる問題に対する解決方法については、 一般的な Composer によるインストール のセクションを参照してください。

そうでない場合は、Yii のリリースを ダウンロード[52] して、 `BasePath/vendor` ディレクトリに解凍してください。

次に、サードパーティのシステムのエントリ・スクリプトを修正します。次のコードをエントリ・スクリプトの先頭に追加してください。

```
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

$yiiConfig = require __DIR__ . '/../config/yii/web.php';
new yii\web\Application($yiiConfig); // ここで run() を呼ばない
```

ごらんのように、上記のコードは典型的な Yii アプリケーションの エントリ・スクリプト と非常に良く似ています。 唯一の違いは、アプリケー

[51] https://github.com/cebe/assetfree-yii2
[52] http://www.yiiframework.com/download/

ションのインスタンスが作成された後に run() メソッドが呼ばれない、という点です。 run() を呼ぶと Yii がリクエスト処理のワークフローを制御するようになりますが、 この場合はリクエストを処理する別のアプリケーションが既に存在していますので、これは必要ではないからです。

Yii アプリケーションでの場合と同じように、サードパーティ・システムが走っている環境に基づいて Yii のアプリケーション・インスタンスを構成する必要があります。 例えば、アクティブ・レコード の機能を使うためには、サードパーティ・システムによって使用されている DB 接続の設定を使って db アプリケーション・コンポーネント を構成しなければなりません。

これで、Yii によって提供されているほとんどの機能を使うことが出来ます。 例えば、アクティブ・レコード・クラスを作成して、それを使ってデータベースを扱うことが出来ます。

14.9.3 Yii 2 を Yii 1 とともに使う

あなたが Yii 1 を前から使っている場合は、たぶん、稼働中の Yii 1 アプリケーションを持っているでしょう。 アプリケーション全体を Yii 2 で書き直す代りに、Yii 2 でのみ利用できる機能を使ってアプリケーションを機能拡張したいこともあるでしょう。 このことは、以下に述べるようにして、実現できます。

> 補足: Yii 2 は PHP 5.4 以上を必要とします。 あなたのサーバと既存のアプリケーションが PHP 5.4 以上をサポートしていることを確認しなければなりません。

最初に、直前の項 で述べられている指示に従って、Yii 2 を既存のアプリケーションにインストールします。

次に、アプリケーションのエントリ・スクリプトを以下のように修正します。

```
// カスタマイズされた Yii クラスをインクルード 下記で説明()
require __DIR__ . '/../components/Yii.php';

// Yii 2 アプリケーションの構成
$yii2Config = require __DIR__ . '/../config/yii2/web.php';
new yii\web\Application($yii2Config); // ここで run() を呼ばない。yii2 app は
    サービス・ロケータとしてのみ使用される。

// Yii 1 アプリケーションの構成
$yii1Config = require __DIR__ . '/../config/yii1/main.php';
Yii::createWebApplication($yii1Config)->run();
```

Yii 1 と Yii 2 の両者が Yii クラスを持っているため、二つを結合するカスタム・バージョンを作成する必要があります。 上記のコードでカスタマイズされた Yii クラス・ファイルをインクルードしていますが、これは下記のようにして作成することが出来ます。

```
$yii2path = '/path/to/yii2';
require $yii2path . '/BaseYii.php'; // Yii 2.x

$yii1path = '/path/to/yii1';
require $yii1path . '/YiiBase.php'; // Yii 1.x

class Yii extends \yii\BaseYii
{
    // YiiBase (1.x) のコードをここにコピー・ペースト
}

spl_autoload_unregister(array('YiiBase','autoload'));
spl_autoload_register(array('Yii','autoload'));

Yii::$classMap = include($yii2path . '/classes.php');
// Yii 2 オートローダを Yii 1 によって登録
Yii::registerAutoloader(['yii\BaseYii', 'autoload']);
// 依存注入コンテナを作成
Yii::$container = new yii\di\Container;
```

以上です。 これで、あなたのコードのどの部分においても、`Yii::$app` を使って Yii 2 アプリケーション・インスタンスにアクセスすることが出来、`Yii::app()` によって Yii 1 アプリケーション・インスタンスを取得することが出来ます。

```
echo get_class(Yii::app()); // 'CWebApplication' を出力
echo get_class(Yii::$app);  // 'yii\web\Application' を出力
```

14.10 Yii をマイクロ・フレームワークとして使う

Yii はベーシック・テンプレートやアドバンスト・テンプレートに含まれる機能なしで使うことが簡単にできます。言葉を換えれば、Yii は既にマイクロ・フレームワークです。Yii を使うためにテンプレートによって提供されているディレクトリ構造を持つことは要求されていません。

このことは、アセットやビューなどの事前定義されたテンプレート・コードを必要としない場合には、特に好都合です。そのような場合の一つが JSON API です。以下に続くセクションで、どのようにしてそれを実現するかを示します。

14.10.1 Yii をインストールする

プロジェクト・ファイルのためのディレクトリを作成し、ワーキング・ディレクトリをそのパスに変更します。例で使用されているコマンドは UNIX ベースのものですが、同様のコマンドが Windows にもあります。

```
mkdir micro-app
cd micro-app
```

> 補足: 続けるためには Composer についての知識が多少必要です。Composer の使い方をまだ知らない場合は、時間を取って、Composer Guide[53] を読んでください。

micro-app ディレクトリの下に composer.json ファイルを作成し、あなたの好みのエディタを使って、下記を追加します。

```
{
    "require": {
        "yiisoft/yii2": "~2.0.0"
    },
    "repositories": [
        {
            "type": "composer",
            "url": "https://asset-packagist.org"
        }
    ]
}
```

ファイルを保存して composer install コマンドを実行します。これによって、フレームワークがその全ての依存とともにインストールされます。

14.10.2 プロジェクトの構造を作成する

フレームワークをインストールしたら、次は、アプリケーションの エントリ・ポイント を作成します。エントリ・ポイントは、アプリケーションを開こうとしたときに、一番最初に実行されるファイルです。セキュリティ上の理由により、エントリ・ポイントを置くディレクトリは別にして、それをウェブ・ルートとします。

web ディレクトリを作成して、下記の内容を持つ index.php をそこに置きます。

```
<?php

// 実運用サーバに配備するときは次の行をコメント・アウトする2
defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

$config = require __DIR__ . '/../config.php';
(new yii\web\Application($config))->run();
```

また config.php という名前のファイルを作成し、アプリケーションの全ての構成情報をそこに含ませます。

```
<?php
return [
    'id' => 'micro-app',
```

[53] https://getcomposer.org/doc/00-intro.md

```
    // アプリケーションの basePath は ‘micro-app‘ ディレクトリになります
    'basePath' => __DIR__,
    // この名前空間からアプリケーションは全てのコントローラを探します
    'controllerNamespace' => 'micro\controllers',
    // 'micro' 名前空間からのクラスのオートロードを可能にするためにエイリアスを
    設定します
    'aliases' => [
        '@micro' => __DIR__,
    ],
];
```

> 情報: 構成情報を index.php ファイルに持つことも出来ますが、別のファイルに持つことを推奨します。 そうすれば、後で示しているように、同じ構成情報をコンソール・アプリケーションから使うことが出来ます。

これであなたのプロジェクトはコーディングの準備が出来ました。プロジェクトのディレクトリ構造を決定するのは、名前空間に注意する限り、あなた次第です。

14.10.3 最初のコントローラを作成する

controllers ディレクトリを作成し、SiteController.php というファイルを追加します。 これが、パス情報を持たないリクエストを処理する、デフォルトのコントローラです。

```
<?php

namespace micro\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return 'こんにちは!';
    }
}
```

このコントローラに違う名前を使いたい場合は、名前を変更して yii\base\Application::$defaultRoute をそれに応じて変更します。 例えば、DefaultController であれば、'defaultRoute' => 'default/index' と変更します。

この時点で、プロジェクトの構造は次のようになっています。

```
micro-app/|-
 composer.json|-
 config.php|-
 web/
```

```
   |- index.php|-
controllers/
   |- SiteController.php
```

まだウェブ・サーバをセットアップしていない場合は、ウェブ・サーバの構成ファイル例 を参照すると良いでしょう。 もう一つのオプションは、PHP の内蔵ウェブ・サーバを利用する `yii serve` コマンドを使うことです。 `micro-app/` ディレクトリから、次のコマンドを実行します。

```
vendor/bin/yii serve --docroot=./web
```

アプリケーションの URL をブラウザで開くと、`SiteController::actionIndex()` で返された “こんにちは!” という文字列が表示される筈です。

> 情報: 私たちの例では、アプリケーションのデフォルトの名前空間 `app` を `micro` に変更しています。 これは、あなたがその名前に縛られていないことを示すためです(万一あなたが縛られていると思っている場合を考えて)。 そして、コントローラの名前空間 を修正し、正しいエイリアスを設定しています。

14.10.4 REST API を作成する

私たちの “マイクロ・フレームワーク” の使い方を示すために、記事のための簡単な REST API を作成しましょう。‘ この API が何らかのデータを提供するためには、まず、データベースが必要です。 データベース接続の構成をアプリケーション構成に追加します。

```
'components' => [
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'sqlite:@micro/database.sqlite',
    ],
],
```

> 情報: ここでは話を簡単にするために sqlite データベースを使用します。他のオプションについては データベースのガイド を参照してください。

次に、データベース・マイグレーション を作成して、記事のテーブルを作成します。 既に述べたように、独立した構成情報ファイルがあることを確認してください。 下記のコンソール・コマンドを実行するためには、それが必要です。 次のコマンドを実行すると、データベース・マイグレーション・ファイルが作成され、そして、マイグレーションがデータベースに適用されます。

```
vendor/bin/yii migrate/create --appconfig=config.php create_post_table --
    fields="title:string,body:text"
vendor/bin/yii migrate/up --appconfig=config.php
```

`models` ディレクトリを作成し、`Post.php` ファイルをそのディレクトリに置きます。以下がそのモデルのためのコードです。

```php
<?php

namespace micro\models;

use yii\db\ActiveRecord;

class Post extends ActiveRecord
{
    public static function tableName()
    {
        return '{{post}}';
    }
}
```

> 情報: ここで作成されたモデルは ActiveRecord クラスのもので、post テーブルのデータを表します。 詳細な情報は アクティブ・レコードのガイド を参照してください。

私たちの API で記事データへのアクセスを提供するために、controllers に PostController を追加します。

```php
<?php

namespace micro\controllers;

use yii\rest\ActiveController;

class PostController extends ActiveController
{
    public $modelClass = 'micro\models\Post';

    public function behaviors()
    {
        // 動作のために認証済みユーザであることを要求する rateLimiter を削除
        $behaviors = parent::behaviors();
        unset($behaviors['rateLimiter']);
        return $behaviors;
    }
}
```

この時点で私たちの API は以下の URL を提供します。

- /index.php?r=post - 全ての記事をリストする
- /index.php?r=post/view&id=1 - ID 1 の記事を表示する
- /index.php?r=post/create - 記事を作成する
- /index.php?r=post/update&id=1 - ID 1 の記事を更新する
- /index.php?r=post/delete&id=1 - ID 1 の記事を削除する

ここから開始して、あなたのアプリケーションの開発を更に進めるために、次のガイドを読むと良いでしょう。

- API は今のところ入力として URL エンコードされたフォームデータだけを理解します。 本物の JSON API にするためには、yii\web\JsonParser を構成する必要があります。

- URL をもっと馴染みやすいものにするためには、ルーティングを構成しなければなりません。 方法を知るためには REST のルーティングのガイド を参照してください。
- 更に参照すべき文書を知るために 先を見通す のセクションを読んでください。

Chapter 15

ウィジェット

Chapter 16

ヘルパ

16.1 ヘルパ

補足: このセクションはまだ執筆中です。

Yii は、一般的なコーディングのタスク、例えば、文字列や配列の操作、HTML コードの生成などを手助けする多くのクラスを提供しています。 これらのヘルパ・クラスは `yii\helpers` 名前空間の下に編成されており、すべてスタティックなクラス (すなわち、スタティックなプロパティとメソッドのみを含み、インスタンス化すべきでないクラス) です。

ヘルパ・クラスは、そのスタティックなメソッドの一つを直接に呼び出すことによって使用します。例えば、

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

補足: ヘルパ・クラスをカスタマイズする ことをサポートするために、Yii はコア・ヘルパ・クラスのすべてを二つのクラスに分割しています。 すなわち、基底クラス (例えば `BaseArrayHelper`) と具象クラス (例えば `ArrayHelper`) です。 ヘルパを使うときは、具象クラスのみを使うべきであり、基底クラスは決して使ってはいけません。

16.1.1 コア・ヘルパ・クラス

以下のコア・ヘルパ・クラスが Yii のリリースにおいて提供されています。

- 配列ヘルパ
- Console
- FileHelper
- FormatConverter
- Html ヘルパ

- HtmlPurifier
- Imagine (yii2-imagine エクステンションによって提供)
- Inflector
- Json
- Markdown
- StringHelper
- Url ヘルパ
- VarDumper

16.1.2 ヘルパ・クラスをカスタマイズする

コア・ヘルパ・クラス (例えば `yii\helpers\ArrayHelper`) をカスタマイズするためには、 そのヘルパに対応する基底クラス (例えば `yii\helpers\BaseArrayHelper`) を拡張するクラスを作成して、名前空間も含めて、対応する具象クラス (例えば `yii\helpers\ArrayHelper`) と同じ名前を付けます。 このクラスが、フレームワークのオリジナルの実装を置き換えるものとしてセットアップされます。

次の例は、`yii\helpers\ArrayHelper` クラスの `merge()` メソッドをカスタマイズする方法を示すものです。

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // あなた独自の実装
    }
}
```

あなたのクラスを `ArrayHelper.php` という名前のファイルに保存します。このファイルはどこに置いても構いません。例えば、`@app/components` に置くことにしましょう。

次に、アプリケーションの エントリ・スクリプト で、 次のコード行を `yii.php` ファイルをインクルードする行の後に追加して、Yii クラス・オートローダ に、 フレームワークから本来のヘルパ・クラスをロードする代りに、あなたのカスタム・クラスをロードすべきことを教えます。

```
Yii::$classMap['yii\helpers\ArrayHelper'] = '@app/components/ArrayHelper.php
    ';
```

ヘルパ・クラスのカスタマイズは、ヘルパの既存の関数の振る舞いを変更したい場合にだけ役立つものであることに注意してください。 アプリケーションの中で使用する関数を追加したい場合は、 そのための独立したヘルパを作成する方が良いでしょう。

16.2 配列ヘルパ

PHP の充実した配列関数[1] への追加として、 Yii の配列ヘルパは、配列をさらに効率的に扱うことを可能にするスタティックなメソッドを提供しています。

16.2.1 値を取得する

配列、オブジェクト、またはその両方から成る複雑な構造から標準的な PHP を使って値を取得することは、非常に面倒くさい仕事です。 最初に `isset` でキーの存在をチェックしなければならず、次に、キーが存在していれば値を取得し、存在していなければ、 デフォルト値を提供しなければなりません。

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name :
    null;
```

Yii はこのための非常に便利なメソッドを提供しています。

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

メソッドの最初の引数は、どこから値を取得しようとしているかを指定します。 二番目の引数は、データの取得の仕方を指定します。これは、以下の一つとすることが出来ます。

- 値を読み出すべき配列のキーまたはオブジェクトのプロパティの名前。
- ドットで分割された配列のキーまたはオブジェクトのプロパティ名のセット。上の例で使用した形式です。
- 値を返すコールバック。

コールバックは次の形式でなければなりません。

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

三番目のオプションの引数はデフォルト値であり、指定されない場合は `null` となります。以下のようにして使用します。

```
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

[1] https://secure.php.net/manual/ja/book.array.php

16.2.2 値を設定する

```
$array = [
    'key' => [
        'in' => ['k' => 'value']
    ]
];

ArrayHelper::setValue($array, 'key.in', ['arr' => 'val']);
// `$array` で値を書くためのパスは配列として指定することも出来ます
ArrayHelper::setValue($array, ['key', 'in'], ['arr' => 'val']);
```

結果として、`$array['key']['in']` の初期値は新しい値によって上書きされます。

```
[
    'key' => [
        'in' => ['arr' => 'val']
    ]
]
```

パスが存在しないキーを含んでいる場合は、キーが作成されます。

```
// `$array['key']['in']['arr0']` が空でなければ、値が配列に追加される
ArrayHelper::setValue($array, 'key.in.arr0.arr1', 'val');

// `$array['key']['in']['arr0']` の値を完全に上書きしたい場合は
ArrayHelper::setValue($array, 'key.in.arr0', ['arr1' => 'val']);
```

結果は以下のようになります

```
[
    'key' => [
        'in' => [
            'k' => 'value',
            'arr0' => ['arr1' => 'val']
        ]
    ]
]
```

16.2.3 配列から値を取り除く

値を取得して、その直後にそれを配列から削除したい場合は、`remove` メソッドを使うことが出来ます。

```
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

このコードを実行した後では、`$array` には `['options' => [1, 2]]` が含まれ、`$type` は `A` となります。 `getValue` メソッドとは違って、`remove` は単純なキー名だけをサポートすることに注意してください。

16.2.4 キーの存在をチェックする

ArrayHelper::keyExists は、大文字と小文字を区別しないキーの比較をサポートすることを除いて、 array_key_exists[2] と同じ動作をします。例えば、

```
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) || !ArrayHelper::
    keyExists('username', $data2, false)) {
    echo "username を提供してください。";
}
```

16.2.5 カラムを取得する

データ行またはオブジェクトの配列から、あるカラムの値を取得する必要があることがよくあります。良くある例は、ID のリストの取得です。

```
$array = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

結果は ['123', '345'] となります。

追加の変形が要求されたり、値の取得方法が複雑であったりする場合は、 無名関数を二番目の引数として指定することが出来ます。

```
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

16.2.6 配列を再インデックスする

指定されたキーに従って配列にインデックスを付けるために、index メソッドを使うことが出来ます。 入力値は、多次元配列であるか、オブジェクトの配列でなければなりません。 $key は、サブ配列のキーの名前、オブジェクトのプロパティの名前、または、キーとして使用される値を返す無名関数とすることが出来ます。

$groups 属性はキーの配列であす。これは、入力値の配列を一つまたは複数のサブ配列にグループ化するために キーとして使用されます。

特定の要素の $key 属性またはその値が null であるとき、$groups が定義されていない場合は、その要素は破棄されて、結果には入りません。

[2] https://secure.php.net/manual/ja/function.array-key-exists.php

そうではなく、$groups が指定されている場合は、 配列の要素はキー無しで結果の配列に追加されます。

例えば、

```
$array = [
    ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
    ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
];
$result = ArrayHelper::index($array, 'id');
```

結果は、id 属性の値をキーとする連想配列になります。

```
[
    '123' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    '345' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
    // 元の配列の番目の要素は、同じ2 id であるため、最後の要素によって上書きされます
]
```

$key として無名関数を渡しても同じ結果になります。

```
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

id を3番目の引数として渡すと、$array を id によってグループ化することが出来ます。

```
$result = ArrayHelper::index($array, null, 'id');
```

結果は、最初のレベルが id でグループ化され、第2のレベルはインデックスされていない連想配列になります。

```
[
    '123' => [
        ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
    ],
    '345' => [ // このインデックスを持つ全ての要素が結果の配列に入る
        ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
        ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
    ]
]
```

無名関数を配列のグループ化に使うことも出来ます。

```
$result = ArrayHelper::index($array, 'data', [function ($element) {
    return $element['id'];
}, 'device']);
```

結果は、最初のレベルが id でグループ化され、第2のレベルが device でグループ化され、 第3のレベルが data でインデックスされた連想配列になります。

```
[
    '123' => [
        'laptop' => [
```

```
            'abc' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
        ]
    ],
    '345' => [
        'tablet' => [
            'def' => ['id' => '345', 'data' => 'def', 'device' => 'tablet']
        ],
        'smartphone' => [
            'hgi' => ['id' => '345', 'data' => 'hgi', 'device' => '
    smartphone']
        ]
    ]
]
```

16.2.7 マップを作成する

多次元配列またはオブジェクトの配列からマップ (キー・値 のペア) を作成するためには `map` メソッドを使うことが出来ます。 `$from` と `$to` のパラメータで、マップを構成するキー名またはプロパティ名を指定します。オプションで、グループ化のためのフィールド `$group` に従って、マップをグループ化することも出来ます。例えば、

```
$array = [
    ['id' => '123', 'name' => 'aaa', 'class' => 'x'],
    ['id' => '124', 'name' => 'bbb', 'class' => 'x'],
    ['id' => '345', 'name' => 'ccc', 'class' => 'y'],
];

$result = ArrayHelper::map($array, 'id', 'name');
// 結果は次のようになります
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// 結果は次のようになります
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
//         '345' => 'ccc',
//     ],
// ]
```

16.2.8 多次元配列の並べ替え

multisort メソッドは、オブジェクトの配列または入れ子にされた配列を、一つまたは複数のキーによって並べ替えることを手助けします。例えば、

```
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

並べ替えの後には、$data に次のデータが入っています。

```
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

並べ替えで参照するキーを指定する二番目の引数は、一つのキーであれば文字列、複数のキーであれば配列を取ることが出来ます。 さらに、次のような無名関数でも構いません。

```
ArrayHelper::multisort($data, function($item) {
    // 存在していれば 'age' で、さもなくば 'name' でソート
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

三番目の引数は並べ替えの順序です。 一つのキーによる並べ替えの場合は、SORT_ASC か SORT_DESC のいずれかです。 複数の値による並べ替えの場合は、並べ替えの順序の配列を渡して、値ごとに違う順序で並べ替えることが出来ます。

最後の引数は並べ替えのフラグで、 PHP の sort()[3] 関数に渡されるのと同じ値を取ることが出来ます。

16.2.9 配列の型を検出する

配列が添字配列であるか連想配列であるかを知ることが出来ると便利です。例を挙げましょう。

```
// キーは指定されていない
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// 全てのキーは文字列
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

[3] https://secure.php.net/manual/ja/function.sort.php

16.2.10 値を HTML エンコード / デコードする

文字列の配列の中にある特殊文字を HTML エンティティにエンコード、または、HTML エンティティからデコードするために、下記の関数を使うことが出来ます。

```
$encoded = ArrayHelper::htmlEncode($data);
$decoded = ArrayHelper::htmlDecode($data);
```

デフォルトでは、値だけがエンコードされます。二番目の引数を `false` として渡すことによって、配列のキーもエンコードすることが出来ます。エンコードにはアプリケーションの文字セットが使用されますが、三番目の引数によってそれを変更することも出来ます。

16.2.11 配列をマージする

`ArrayHelper::merge()` を使って、二つまたはそれ以上の配列を再帰的に一つの配列にマージすることが出来ます。 各配列に同じ文字列のキー値を持つ要素がある場合は、 (array_merge_recursive()[4] とは違って)後のものが前のものを上書きします。 両方の配列が、同じキーを持つ配列型の要素を持っている場合は、再帰的なマージが実行されます。 添字型の要素については、後の配列の要素が前の配列の要素の後に追加されます。 `yii\helpers\UnsetArrayValue` オブジェクトを使って前の配列にある値を非設定に指定したり、 `yii\helpers\ReplaceArrayValue` オブジェクトを使って再帰的なマージでなく前の値の上書きを強制したりすることが出来ます。

例えば、

```
$array1 = [
    'name' => 'Yii',
    'version' => '1.1',
    'ids' => [
        1,
    ],
    'validDomains' => [
        'example.com',
        'www.example.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
        'dev' => 'dev@example.com',
    ],
];

$array2 = [
    'version' => '2.0',
    'ids' => [
        2,
    ],
```

[4] https://secure.php.net/manual/ja/function.array-merge-recursive.php

```
    'validDomains' => new \yii\helpers\ReplaceArrayValue([
        'yiiframework.com',
        'www.yiiframework.com',
    ]),
    'emails' => [
        'dev' => new \yii\helpers\UnsetArrayValue(),
    ],
];

$result = ArrayHelper::merge($array1, $array2);
```

結果は次のようになります。

```
[
    'name' => 'Yii',
    'version' => '2.0',
    'ids' => [
        1,
        2,
    ],
    'validDomains' => [
        'yiiframework.com',
        'www.yiiframework.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
    ],
]
```

16.2.12 オブジェクトを配列に変換する

オブジェクトまたはオブジェクトの配列を配列に変換する必要があることがよくあります。 最もよくあるのは、REST API によってデータ配列を提供するなどの目的で、アクティブ・レコード・モデルを変換する場合です。そうするために、次のコードを使うことが出来ます。

```
$posts = Post::find()->limit(10)->all();
$data = ArrayHelper::toArray($posts, [
    'app\models\Post' => [
        'id',
        'title',
        // 結果配列のキー名 => プロパティの値
        'createTime' => 'created_at',
        // 結果配列のキー名 => 無名関数が返す値
        'length' => function ($post) {
            return strlen($post->content);
        },
    ],
]);
```

最初の引数が変換したいデータです。この例では、`Post` AR モデルを変換しようとしています。

二番目の引数は、クラスごとの変換マップです。ここでは、Post モデルの変換マップを設定しています。 変換マップの配列が、一連のマップを含んでいます。各マップは以下のいずれかの形式を取ります。

- フィールド名 - そのままインクルードされる。
- キー/値 のペア - 配列のキー名にしたい文字列と、値を取得すべきモデルのカラムの名前。
- キー/値 のペア - 配列のキー名にしたい文字列と、値を返すコールバック。

単一のモデルに対する上記の変換の結果は以下のようになります。

```
[
    'id' => 123,
    'title' => 'test',
    'createTime' => '2013-01-01 12:00AM',
    'length' => 301,
]
```

特定のクラスについて、配列に変換するデフォルトの方法を提供するためには、 そのクラスの Arrayable インタフェイスを実装することが出来ます。

16.2.13 配列の中にあるかどうか調べる

ある要素が配列の中に存在するかどうか、また、一連の要素が配列のサブセットであるかどうか、ということを調べる必要がある場合がよくあります。 PHP は in_array() を提供していますが、これはサブセットや \Traversable なオブジェクトをサポートしていません。

この種のチェックを助けるために、yii\helpers\ArrayHelper は isIn() および isSubset() を in_array()[5] と同じシグニチャで提供しています。

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new ArrayObject(['a']));

// true
ArrayHelper::isSubset(new ArrayObject(['a', 'c']), new ArrayObject(['a', 'b'
    , 'c']));
```

16.3 Html ヘルパ

全てのウェブ・アプリケーションは大量の HTML マークアップを生成します。 マークアップが静的な場合は、PHP と HTML を一つのファイルに混ぜる[6] ことによって効率よく生成することが可能ですが、マークアップを動的にするとなると、何らかの助けが無ければ、処理がトリッキー

[5] https://secure.php.net/manual/en/function.in-array.php
[6] https://secure.php.net/manual/ja/language.basic-syntax.phpmode.php

になってきます。 Yii はそのような手助けを Html ヘルパの形式で提供します。 これは、よく使われる HTML タグとそのオプションやコンテントを処理するための一連のスタティック・メソッドを提供するものです。

> 補足: あなたのマークアップがおおむね静的なものである場合は、HTML を直接に使用する方が適切です。 何でもかんでも Html ヘルパの呼び出しでラップする必要はありません。

16.3.1 基礎

動的な HTML を文字列の連結によって構築していると、あっという間に乱雑なコードになります。 そのため、Yii はタグのオプションを操作し、それらのオプションに基づいてタグを構築する一連のメソッドを提供します。

タグを生成する

タグを生成するコードは次のようなものです。

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

最初の引数はタグの名前です。二番目の引数は、開始タグと終了タグの間に囲まれることになるコンテントです。 `Html::encode` を使っていることに注目してください。 これは、必要な場合には HTML を使うことが出来るように、コンテントが自動的にはエンコードされないからです。 三番目の引数は HTML のオプション、言い換えると、タグの属性です。この配列で、キーは `class`、`href`、`target` などの属性の名前であり、値は属性の値です。

上記のコードは次の HTML を生成します。

```
<p class="username">samdark</p>
```

開始タグまたは終了タグだけが必要な場合は、`Html::beginTag()` または `Html::endTag()` のメソッドを使うことが出来ます。

オプションは多くの Html ヘルパのメソッドとさまざまなウィジェットで使用されます。 その全ての場合において、いくつか追加の処理がなされることを知っておいてください。

- 値が `null` である場合は、対応する属性はレンダリングされません。
- 値が真偽値である属性は、真偽値属性 (boolean attributes)[7] として扱われます。
- 属性の値は `Html::encode()` を使って HTML エンコードされます。
- 属性の値が配列である場合は、次のように処理されます。
  - 属性が `yii\helpers\Html::$dataAttributes` にリストされているデータ属性である場合、例えば `data` や `ng` である場合は、 値の配列にある要素の一つ一つについて、属性のリ

[7] http://www.w3.org/TR/html5/infrastructure.html#boolean-attributes

ストがレンダリングされます。 例えば、'data' => ['id' => 1, 'name' => 'yii'] は data-id="1" data-name="yii" を生成します。 また、'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok'] は data-params='{"id":1,"name":"yii"}' data-status="ok" を生成します。 後者の例において、下位の配列に対して JSON 形式が使用されていることに注意してください。
- 属性がデータ属性でない場合は、値は JSON エンコードされます。 例えば、['params' => ['id' => 1, 'name' => 'yii'] は params='{"id":1,"name":"yii"}' を生成します。

CSS のクラスとスタイルを形成する

HTML タグのオプションを構築する場合、たいていは、デフォルトの値から始めて必要な修正をする、という方法をとります。 CSS クラスを追加または削除するために、次のコードを使用することが出来ます。

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Pwede na', $options);

// $type が 'success' の場合、次のようにレンダリングされる
// <div class="btn btn-success">Pwede na</div>
```

配列形式を使って複数の CSS クラスを指定することも出来ます。

```
$options = ['class' => ['btn', 'btn-default']];

echo Html::tag('div', 'Save', $options);
// '<div class="btn btn-default">Save</div>' をレンダリングする
```

クラスを追加・削除する際にも配列形式を使うことが出来ます。

```
$options = ['class' => 'btn'];

if ($type === 'success') {
    Html::addCssClass($options, ['btn-success', 'btn-lg']);
}

echo Html::tag('div', 'Save', $options);
// '<div class="btn btn-success btn-lg">Save</div>' をレンダリングする
```

Html::addCssClass() はクラスの重複を防止しますので、同じクラスが二度出現するかも知れないと心配する必要はありません。

```
$options = ['class' => 'btn btn-default'];

Html::addCssClass($options, 'btn-default'); // クラス 'btn-default' は既に存
    在する
```

```
echo Html::tag('div', 'Save', $options);
// '<div class="btn btn-default">Save</div>' をレンダリングする
```

CSS のクラスオプションを配列形式で指定する場合には、名前付きのキーを使ってクラスの論理的な目的を示すことが出来ます。 この場合、`Html::addCssClass()` で同じキーを持つクラスを指定しても無視されます。

```
$options = [
    'class' => [
        'btn',
        'theme' => 'btn-default',
    ]
];

Html::addCssClass($options, ['theme' => 'btn-success']); // 'theme' キーは既
    に使用されている

echo Html::tag('div', 'Save', $options);
// '<div class="btn btn-default">Save</div>' をレンダリングする
```

CSS のスタイルも `style` 属性を使って、同じように設定することが出来ます。

```
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// style="width: 100px; height: 200px; position: absolute;" となる
Html::addCssStyle($options, 'height: 200px; position: absolute;');

// style="position: absolute;" となる
Html::removeCssStyle($options, ['width', 'height']);
```

`addCssStyle()` を使うときには、CSS プロパティの名前と値に対応する「キー-値」ペアの配列か、 または、`width: 100px; height: 200px;` のような文字列を指定することが出来ます。 この二つの形式は、`cssStyleFromArray()` と `cssStyleToArray()` を使って、双方向に変換することが出来ます。 `removeCssStyle()` メソッドは、削除すべきプロパティの配列を受け取ります。 プロパティが一つだけである場合は、文字列で指定することも出来ます。

コンテントをエンコードおよびデコードする

コンテントが適切かつ安全に HTML として表示されるためには、コンテント内の特殊文字がエンコードされなければなりません。 特殊文字のエンコードとデコードは、PHP では htmlspecialchars[8] と htmlspecialchars_decode[9] によって行われます。 これらのメソッドを直接使用する場合の問題は、文字エンコーディングと追加のフラグを毎回指定しな

[8] https://secure.php.net/manual/ja/function.htmlspecialchars.php
[9] https://secure.php.net/manual/ja/function.htmlspecialchars-decode.php

ければならないことです。 フラグは毎回同じものであり、文字エンコーディングはセキュリティ問題を防止するためにアプリケーションのそれと一致すべきものですから、 Yii は二つのコンパクトかつ使いやすいメソッドを用意しました。

```
$userName = Html::encode($user->name);
echo $userName;

$decodedUserName = Html::decode($userName);
```

16.3.2 フォーム

フォームのマークアップを扱う仕事は、極めて面倒くさく、エラーを生じがちなものです。 このため、フォームのマークアップの仕事を助けるための一群のメソッドがあります。

> 補足: モデルを扱っており、検証が必要である場合は、ActiveForm を使うことを検討してください。

フォームを作成する

フォームを開始するためには、次のように beginForm() メソッドを使うことが出来ます。

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' => '
    multipart/form-data']) ?>
```

最初の引数は、フォームが送信されることになる URL です。これは Url ::to() によって受け入れられる Yii のルートおよびパラメータの形式で指定することが出来ます。 第二の引数は使われるメソッドです。post がデフォルトです。第三の引数はフォームタグのオプションの配列です。上記の場合では、POST リクエストにおけるフォーム・データのエンコーディング方法を multipart/form-data に変更しています。 これはファイルをアップロードするために必要とされます。

フォーム・タグを閉じるのは簡単です。

```
<?= Html::endForm() ?>
```

ボタン

ボタンを生成するためには、次のコードを使うことが出来ます。

```
<?= Html::button('押してね !', ['class' => 'teaser']) ?>
<?= Html::submitButton('送信', ['class' => 'submit']) ?>
<?= Html::resetButton('リセット', ['class' => 'reset']) ?>
```

最初の引数は、三つのメソッドのどれでも、ボタンのタイトルであり、第二の引数はオプションです。 タイトルはエンコードされませんので、エンド・ユーザからデータを取得する場合は Html::encode() を使ってエンコードしてください。

インプット・フィールド

インプットのメソッドには二つのグループがあります。 一つは active から始まるものでアクティブ・インプットと呼ばれます。もう一方は active から始まらないものです。 アクティブ・インプットは、データを指定されたモデルと属性から取得しますが、通常のインプットでは、データは直接に指定されます。

最も汎用的なメソッドは以下のものです。

```
タイプ、インプットの名前、値、オプション

<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>
```

```
    タイプ、モデル、モデルの属性名、オプショ
    ン


<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

インプットのタイプが前もって判っている場合は、ショートカットメソッドを使う方が便利です。

- yii\helpers\Html::buttonInput()
- yii\helpers\Html::submitInput()
- yii\helpers\Html::resetInput()
- yii\helpers\Html::textInput(), yii\helpers\Html::activeTextInput()
- yii\helpers\Html::hiddenInput(), yii\helpers\Html::activeHiddenInput()
- yii\helpers\Html::passwordInput() / yii\helpers\Html::activePasswordInput()
- yii\helpers\Html::fileInput(), yii\helpers\Html::activeFileInput()
- yii\helpers\Html::textarea(), yii\helpers\Html::activeTextarea()

ラジオとチェックボックスは、メソッドのシグニチャの面で少し異なっています。

```
<?= Html::radio('agree', true, ['label' => '同意します']) ?>
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement']) ?>

<?= Html::checkbox('agree', true, ['label' => '同意します']) ?>
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement']) ?>
```

ドロップダウン・リストとリスト・ボックスは、次のようにしてレンダリングすることが出来ます。

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels,
    'id', 'name')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels, 'id
    ', 'name')) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels, 'id'
    , 'name')) ?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id', '
    name')) ?>
```

最初の引数はインプットの名前、第二の引数は現在選択されている値です。そして第三の引数は「キー-値」のペアであり、配列のキーはリストの値、配列の値はリストのラベルです。

複数の選択肢を選択できるようにしたい場合は、チェックボックス・リストが最適です。

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels, 'id'
    , 'name')) ?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels, '
    id', 'name')) ?>
```

そうでない場合は、ラジオ・リストを使います。

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id', '
    name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels, 'id',
     'name')) ?>
```

ラベルとエラー

インプットと同じように、ラベルを生成するメソッドが二つあります。モデルからデータを取るアクティブなラベルと、データを直接受け入れるアクティブでないラベルです。

```
<?= Html::label('ユーザ名', 'username', ['class' => 'label username']) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username']) ?>
```

一つまたは複数のモデルから取得したエラーを要約として表示するためには、次のコードを使うことが出来ます。

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

個別のエラーを表示するためには、次のようにします。

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```

インプットの名前と値

モデルに基づいてインプット・フィールドの名前、ID、値を取得するメソッドがあります。 これらは主として内部的に使用されるものですが、場合によっては重宝します。

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// 私の最初の投稿''
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

上記において、最初の引数はモデルであり、第二の引数は属性を示す式です。これは最も単純な形式においては属性名ですが、配列の添字を前および/または 後に付けた属性名とすることも出来ます。配列の添字は主として表形式データ入力のために使用されます。

- `[0]content` は、表形式データ入力で使われます。表形式入力の最初のモデルの “content” 属性を表します。
- `dates[0]` は、"dates” 属性の最初の配列要素を表します。
- `[0]dates[0]` は、表形式入力の最初のモデルの “dates” 属性の最初の配列要素を表します。

前後の添字なしに属性名を取得するためには、次のコードを使うことが出来ます。

```
// dates
echo Html::getAttributeName('dates[0]');
```

16.3.3 スタイルとスクリプト

埋め込みのスタイルとスクリプトをラップするタグを生成するメソッドが二つあります。

```
<?= Html::style('.danger { color: #f00; }', ['media' => 'print']) ?>これは次
    の

 HTML を生成します。

<style media="print">.danger { color: #f00; }</style>


<?= Html::script('alertこんにちは("!");') ?>これは次の

 HTML を生成します。

<script>alert("こんにちは!");</script>
```

CSS ファイルの外部スタイルをリンクしたい場合は、次のようにします。

```
<?= Html::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>これは次の

 HTML を生成します。

<!--[if IE 5]>
    <link href="http://example.com/css/ie5.css" />
<![endif]-->
```

最初の引数は URL であり、第二の引数はオプションの配列です。通常のオプションに加えて、次のものを指定することが出来ます。

- `condition` - 指定された条件を使って `<link` を条件付きコメントで囲みます。 条件付きコメントなんて、使う必要が無くなっちゃえば良いのにね ;)

- noscript - true に設定すると <link を <noscript> タグで囲むことができます。 この場合、JavaScript がブラウザでサポートされていないか、ユーザが JavaScript を無効にしたときだけ、CSS がインクルードされます。

JavaScript ファイルをリンクするためには、次のようにします。

```
<?= Html::jsFile('@web/js/main.js') ?>
```

CSS と同じように、最初の引数はインクルードされるファイルへのリンクを指定するものです。オプションを第二の引数として渡すことが出来ます。 オプションに置いて、cssFile のオプションと同じように、condition を指定することが出来ます。

16.3.4 ハイパーリンク

ハイパーリンクを手軽に生成できるメソッドがあります。

```
<?= Html::a('プロファイ
    ル', ['user/view', 'id' => $id], ['class' => 'profile-link']) ?>
```

最初の引数はタイトルです。これはエンコードされませんので、エンド・ユーザから取得したデータを使う場合は、Html::encode() でエンコードする必要があります。 第二の引数が、<a タグの href に入ることになるものです。 どのような値が受け入れられるかについて、詳細は Url::to() を参照してください。 第三の引数は、タグのプロパティの配列です。

mailto リンクを生成する必要があるときは、次のコードを使うことが出来ます。

```
<?= Html::mailto('連絡先', 'admin@example.com') ?>
```

16.3.5 画像

イメージタグを生成するためには次のようにします。

```
<?= Html::img('@web/images/logo.png', ['alt' => '私のロゴ']) ?>これは次の

 HTML を生成します。

<img src="http://example.com/images/logo.png" alt="私のロゴ" />
```

最初の引数は、エイリアス 以外にも、ルートとパラメータ、または URL を受け入れることが出来ます。Url::to() と同様です。

16.3.6 リスト

順序なしリストは、次のようにして生成することが出来ます。

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
```

```
        ['class' => 'post']
    );
}]) ?>
```

順序付きリストを生成するためには、代りに `Html::ol()` を使ってください。

16.4 Url ヘルパ

Url ヘルパは URL を管理するための一連のスタティック・メソッドを提供します。

16.4.1 よく使う URL を取得する

よく使う URL を取得するために使うことが出来るメソッドが二つあります。すなわち、ホーム URL と、現在のリクエストのベース URL を取得するメソッドです。 ホーム URL を取得するためには、次のようにします。

```
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

パラメータが渡されない場合は、生成される URL は相対 URL になります。 パラメータとして `true` を渡せば、現在のスキーマの絶対 URL を取得することが出来ます。または、スキーマ (`http`, `https`) を明示的に指定しても構いません。

現在のリクエストのベース URL を取得するためには、次のようにします。

```
$relativeBaseUrl = Url::base();
$absoluteBaseUrl = Url::base(true);
$httpsAbsoluteBaseUrl = Url::base('https');
```

このメソッドの唯一のパラメータは、`Url::home()` の場合と全く同じ動作をします。

16.4.2 URL を生成する

与えられたルートへの URL を生成するためには、`Url::toRoute()` メソッドを使います。 このメソッドは、`yii\web\UrlManager` を使って URL を生成します。

```
$url = Url::toRoute(['product/view', 'id' => 42]);
```

ルートは、文字列として指定することが出来ます (例えば、`site/index`)。または、生成される URL に追加のクエリ・パラメータを指定したい場合は、配列を使うことも出来ます。配列の形式は、以下のようにしなければなりません。

```
// /index.php?r=site%2Findex&param1=value1&param2=value2 を生成
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

アンカーの付いた URL を生成したい場合は、# パラメータを持つ配列を使うことが出来ます。例えば、

```
// /index.php?r=site%2Findex&param1=value1#name を生成
['site/index', 'param1' => 'value1', '#' => 'name']
```

ルートは、絶対ルートか相対ルートかのどちらかです。絶対ルートは先頭にスラッシュを持ち (例えば /site/index)、相対ルートは持ちません (例えば site/index または index)。 相対ルートは次の規則に従って絶対ルートに変換されます。

- ルートが空文字列である場合は、現在の ルート が使用されます。
- ルートがスラッシュを全く含まない場合は (例えば index)、カレント・コントローラのアクション ID であると見なされて、 カレント・コントローラの uniqueId が前置されます。
- ルートが先頭にスラッシュを含まない場合は (例えば site/index)、カレント・モジュールに対する相対ルートと見なされて、 カレント・モジュールの uniqueId が前置されます。

バージョン 2.0.2 以降では、エイリアス の形式でルートを指定することが出来ます。 その場合は、エイリアスが最初に実際のルートに変換され、そのルートが上記の規則に従って絶対ルートに変換されます。

以下に、このメソッドの使用例をいくつか挙げます。

```
// /index.php?r=site%2Findex
echo Url::toRoute('site/index');

// /index.php?r=site%2Findex&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100      エイリア
    ス "@postEdit" は "post/edit" と定義されていると仮定
echo Url::toRoute(['@postEdit', 'id' => 100]);

// http://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', 'https');
```

もうひとつ、toRoute() と非常によく似た Url::to() というメソッドがあります。 唯一の違いは、このメソッドはルートを配列として指定することを要求する、という点です。文字列が与えられた場合は、URL として扱われます。

最初の引数は、次のいずれかを取り得ます。

- 配列: URL を生成するために toRoute() が呼び出されます。 例えば、['site/index']、['post/index', 'page' => 2]。 ルートの指定方法の詳細については toRoute() を参照してください。

- `@` で始まる文字列: これはエイリアスとして扱われ、 エイリアスに対応する文字列が返されます。
- 空文字列: 現在リクエストされている URL が返されます。
- 通常の文字列: その通りのものとして扱われます。

`$scheme` (文字列または `true`) が指定された場合は、ホスト情報 (`yii\web\UrlManager::$hostInfo` から取得されます) を伴う絶対 URL が返されます。 `$url` が既に絶対 URL であった場合には、スキームが指定されたものに置き換えられます。

下記にいくつかの用例を挙げます。

```
// /index.php?r=site%2Findex
echo Url::to(['site/index']);

// /index.php?r=site%2Findex&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100      エイリア
    ス "@postEdit" が "post/edit" と定義されていると仮定
echo Url::to(['@postEdit', 'id' => 100]);

// 現在リクエストされている URL
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// http://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

バージョン 2.0.3 以降では、`yii\helpers\Url::current()` を使って、現在リクエストされているルートと GET パラメータに基づいて URL を生成することが出来ます。 `$params` パラメータを渡して、GET パラメータの中のいくつかを修正したり削除したり、または新しい GET パラメータを追加したりすることが出来ます。 例えば、

```
// $_GET が ['id' => 123, 'src' => 'google'] であり、現在のルート
    が "post/view" であると仮定

// /index.php?r=post%2Fview&id=123&src=google
echo Url::current();

// /index.php?r=post%2Fview&id=123
echo Url::current(['src' => null]);
// /index.php?r=post%2Fview&id=100&src=google
echo Url::current(['id' => 100]);
```

16.4.3 URL を記憶する

URL を記憶して、後に続く一連のリクエストの一つを処理するときに、記憶した URL を使わなければならないという場合があります。 これは、次のようにして達成することが出来ます。

```
// 現在の URL を記憶する
Url::remember();

// 指定された URL を記憶する。引数の形式は Url::to() を参照。
Url::remember(['product/view', 'id' => 42]);

// 指定された名前で URL を記憶する。
Url::remember(['product/view', 'id' => 42], 'product');
```

次のリクエストで、記憶された URL を次のようにして取得することが出来ます。

```
$url = Url::previous();
$productUrl = Url::previous('product');
```

16.4.4 相対 URL かどうかチェックする

URL が相対 URL であること、すなわち、URL がホスト情報の部分を持っていないことを確かめるために、次のコードを使うことが出来ます。

```
$isRelative = Url::isRelative('test/it');
```