

The Definitive Guide to Yii 2.0

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

Copyright 2014 Yii Software LLC.

Оглавление

1	Введение	1
1.1	Что такое Yii?	1
1.2	Обновление с версии 1.1	2
2	Первое знакомство	15
2.1	Установка Yii	15
2.2	Запуск приложения	20
2.3	Говорим «Привет»	24
2.4	Работа с формами	27
2.5	Работа с базами данных	32
2.6	Генерация кода при помощи Gii	38
2.7	Взгляд в будущее	46
3	Структура приложения	49
3.1	Обзор	49
3.2	Входные скрипты	50
3.3	Приложения	52
3.4	Компоненты приложения	65
3.5	Контроллеры	67
3.6	Виды	77
3.7	Фильтры	95
3.8	Виджеты	103
3.9	Модули	107
3.10	Ресурсы	113
3.11	Расширения	130
4	Обработка запросов	143
4.1	Предзагрузка	143
4.2	Работа с URL	144
4.3	Запросы	152
4.4	Ответы	155
4.5	Работа с URL	163
4.6	Обработка ошибок	170

4.7	Логгирование	175
5	Основные понятия	179
5.1	Компоненты	179
5.2	Свойства	181
5.3	События	183
5.4	Поведения	189
5.5	Конфигурации	195
5.6	Псевдонимы	201
5.7	Автозагрузка классов	203
5.8	Service Locator	205
5.9	Контейнер внедрения зависимостей	208
6	Работа с базами данных	217
7	Получение данных от пользователя	227
7.1	Проверка входящих данных	229
8	Отображение данных	245
8.1	Форматирование данных	245
8.2	Постраничное разделение данных	250
8.3	Сортировка	251
9	Безопасность	257
10	Кэширование	263
10.1	Кэширование	263
10.2	Кэширование данных	263
10.3	Кэширование фрагментов	271
10.4	Кэширование страниц	275
10.5	HTTP кэширование	276
11	Веб-сервисы REST	281
11.1	Быстрый старт	281
11.2	Ресурсы	285
11.3	Контроллеры	289
11.4	Маршрутизация	292
11.5	Форматирование ответа	294
11.6	Аутентификация	297
11.7	Ограничение частоты запросов	300
11.8	Версионирование	301
11.9	Обработка ошибок	304
12	Инструменты разработчика	307

13 Тестирование	311
14 Расширение Yii	317
15 Специальные темы	325
15.1 Шаблон приложения advanced	325
15.2 Интернационализация	334
16 Виджеты	349
17 Хелперы	353
17.1 ArrayHelper	355

Глава 1

Введение

1.1 Что такое Yii?

Yii – это высокопроизводительный компонентный PHP фреймворк, предназначенный для быстрой разработки современных веб приложений. Слово Yii (произносится как йи [ji:]) в китайском языке означает «простой и эволюционирующий». Также Yii может расшифровываться как акроним **Yes It Is!**

1.1.1 Для каких задач больше всего подходит Yii?

Yii – это универсальный фреймворк и может быть задействован во всех типах веб приложений. Благодаря его компонентной структуре и отличной поддержке кэширования, фреймворк особенно подходит для разработки таких крупных проектов как порталы, форумы, CMS, магазины или RESTful-приложения.

1.1.2 Сравнение Yii с другими фреймворками

Если вы уже знакомы с другими фреймворками, вам наверняка будет интересно сравнить их с Yii.

- Как и многие другие PHP фреймворки, для организации кода Yii использует модель MVC (Model-View-Controller).
- Yii придерживается философии простого и элегантного кода не пытаясь усложнять дизайн только ради следования каким-либо шаблонам проектирования.
- Yii является full-stack фреймворком и включает в себя проверенные и хорошо зарекомендовавшие себя возможности, такие как ActiveRecord для реляционных и NoSQL баз данных, поддержку REST API, многоуровневое кэширование и другие.
- Yii отлично расширяем. Вы можете настроить или заменить практически любую часть основного кода. Используя архитектуру рас-

ширений легко делиться кодом или использовать код сообщества.

- Одна из главных целей Yii – производительность.

Yii — не проект одного человека. Он поддерживается и развивается сильной командой¹ и большим сообществом разработчиков, которые ей помогают. Авторы фреймворка следят за тенденциями веб разработки и развитием других проектов. Наиболее подходящие возможности и лучшие практики регулярно внедряются в фреймворк в виде простых и элегантных интерфейсов.

1.1.3 Версии Yii

На данный момент существует две основные ветки Yii: 1.1 и 2.0. Ветка 1.1 является предыдущим поколением и находится в состоянии поддержки. Версия 2.0 – это полностью переписанный Yii, использующий последние технологии и протоколы, такие как Composer, PSR, пространства имен, трейты и многое другое. 2.0 — текущее поколение фреймворка. На этой версии будут сосредоточены основные усилия несколько следующих лет. Данное руководство именно о версии 2.0.

1.1.4 Требования к ПО и знаниям

Yii 2.0 требует PHP 5.4.0 и выше. Чтобы узнать требования для отдельных возможностей вы можете запустить скрипт проверки требований, который поставляется с каждым релизом фреймворка.

Для разработки на Yii потребуется общее понимание ООП так как фреймворк полностью следует этой парадигме. Также стоит изучить такие современные возможности PHP как пространства имён² и трейты³.

1.2 Обновление с версии 1.1

Между версиями 1.1 и 2.0 существует много различий, так как Yii был полностью переписан для версии 2.0. Таким образом, обновление с версии 1.1 не является таким же тривиальным как обновление между минорными версиями. В данном руководстве приведены основные различия между двумя версиями.

Если прежде вы не использовали Yii 1.1, вы можете сразу перейти к разделу «[Начало работы][start-installation.md]».

Также учтите, что в Yii 2.0 гораздо больше новых возможностей, чем описано далее. Настоятельно рекомендуется, изучить всё руководство. Вполне возможно, что то, что раньше приходилось разрабатывать самостоятельно теперь является частью фреймворка.

¹<http://www.yiiframework.com/about/>

²<http://www.php.net/manual/ru/language.namespaces.php>

³<http://www.php.net/manual/ru/language.oop5.traits.php>

1.2.1 Установка

Yii 2.0 широко использует Composer⁴, который является основным менеджером зависимостей для PHP. Установка как фреймворка, так и расширений, осуществляется через Composer. Подробно о установке Yii 2.0 вы можете узнать из раздела «Установка Yii». О том, как создавать расширения для Yii 2.0 или адаптировать уже имеющиеся расширения от версии 1.1, вы можете узнать из раздела «Создание расширений».

1.2.2 Требования PHP

Для работы Yii 2.0 необходим PHP 5.4 или выше. Данная версия включает большое количество улучшений по сравнению с версией 5.2, которая использовалась Yii 1.1. Таким образом, существует много различий в языке, которые вы должны принимать во внимание:

- Пространства имён⁵;
- Анонимные функции⁶;
- Использование короткого синтаксиса для массивов: `элементы[.]` вместо `arrayэлементы(.)`;
- Использование короткого `echo <?=` для вывода в файлах представлений. С версии PHP 5.4 данную возможность можно использовать не опасаясь;
- Классы и интерфейсы SPL⁷;
- Позднее статическое связывание (LSB)⁸;
- Классы для дат и времени⁹;
- Трейты¹⁰;
- Интернационализация (intl)¹¹; Yii 2.0 использует расширение PHP `intl` для различного функционала интернационализации.

1.2.3 Пространства имён

Одним из основных изменений в Yii 2.0 является использование пространств имён. Почти каждый класс фреймворка находится в пространстве имён, например, `yii\web\Request`. Префикс “C” в именах классов больше не используется. Имена классов соответствуют структуре директорий. Например, `yii\web\Request` указывает, что соответствующий класс находится в файле `web/Request.php` в директории фреймворка.

⁴<https://getcomposer.org/>

⁵<http://php.net/manual/ru/language.namespaces.php>

⁶<http://php.net/manual/ru/functions.anonymous.php>

⁷<http://php.net/manual/ru/book.spl.php>

⁸<http://php.net/manual/ru/language.oop5.late-static-bindings.php>

⁹<http://php.net/manual/ru/book.datetime.php>

¹⁰<http://php.net/manual/ru/language.oop5.traits.php>

¹¹<http://php.net/manual/ru/book.intl.php>

Благодаря загрузчику классов Yii, вы можете использовать любой класс фреймворка без необходимости непосредственно подключать его.

1.2.4 Компонент и объект

В Yii 2.0 класс `CComponent` из версии 1.1 был разделён на два класса: `yii\base\Object` и `yii\base\Component`. Класс `Object` является простым базовым классом, который позволяет использовать геттеры и сеттеры для свойств. Класс `Component` наследуется от класса `Object` и поддерживает события и поведения.

Если вашему классу не нужны события или поведения, вы можете использовать `Object` в качестве базового класса. В основном это относится к классам, представляющим собой базовые структуры данных.

1.2.5 Конфигурация объекта

Класс `Object` предоставляет единый способ конфигурирования объектов. Любой дочерний класс `Object` может определить конструктор (если нужно) как показано ниже. Это позволит конфигурировать его универсально:

```
class MyClass extends \yii\base\Object
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... инициализация до того, как конфигурация будет применена

        parent::__construct($config);
    }

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

        // ... инициализация после того, как конфигурация была применена
    }
}
```

В примере выше, последний параметр конструктора должен быть массивом конфигурации, который содержит пары в формате ключ-значение для инициализации свойств объекта. Вы можете переопределить метод `init()` для инициализации объекта после того, как конфигурация была применена к нему.

Следуя этому соглашению, вы сможете создавать и конфигурировать новые объекты с помощью массива конфигурации:

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
```

```
], [$param1, $param2]);
```

Более подробная информация о конфигурации представлена в разделе «Настройки».

1.2.6 События

В Yii 1, события создавались с помощью объявления метода `on` (например, `onBeforeSave`). В Yii2 вы можете использовать любое имя события. Вызывать события можно при помощи метода `trigger()`.

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

Для прикрепления обработчика события используйте метод `on()`.

```
$component->on($eventName, $handler);  
// убираем обработчик  
// $component->off($eventName, $handler);
```

Есть и другие улучшения по части событий, подробно описанные в в разделе «События».

1.2.7 Псевдонимы пути

В Yii 2.0 псевдонимы используются более широко и применяются как к путям в файловой системе, так и к URL. Теперь, для того, чтобы отличать псевдонимы от обычных путей и URL, требуется, чтобы имя псевдонима начиналось с символа `@`. Например, псевдоним `@yii` соответствует директории, в которую установлен Yii. Псевдонимы пути используются во многих местах. Например, значение свойства `yii\caching\FileCache::$cachePath` может быть как псевдонимом пути так и обычным путём к папке.

Псевдонимы пути тесно связаны с пространством имён классов. Рекомендуется определять псевдоним пути для каждого корневого пространства имён, что позволяет использовать загрузчик классов Yii без какой-либо дополнительной настройки. Например, так как `@yii` соответствует директории, в которую установлен фреймворк, класс `yii\webRequest` может быть загружен автоматически. Если вы используете сторонние библиотеки, например, из Zend Framework, вы можете определить псевдоним пути `@Zend` как директорию, в которую установлен этот фреймворк. После этого Yii будет способен автоматически загружать любой класс Zend Framework.

Подробнее о псевдонимах пути можно узнать из раздела «Псевдонимы пути».

1.2.8 Представления

Одним из основных изменений в Yii 2 является то, что специальная переменная `$this` в представлении, больше не соответствует текущему контроллеру или виджету. Вместо этого, `$this` теперь соответствует объекту *представления*, новой возможности введённой в версии 2.0. Объект представления имеет тип `yii\web\View`, который представляет собой часть *view* в шаблоне проектирования MVC. Если вы хотите получить доступ к контроллеру или виджету, используйте выражение `$this->context`.

Для рендеринга частичных представлений теперь используется метод `$this->render()`, а не `$this->renderPartial()`. Результат вызова метода `render` теперь должен быть выведен напрямую, так как `render` возвращает результат рендеринга, а не отображает его сразу:

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

Кроме использования РНР в качестве основного шаблонизатора, Yii 2.0 также предоставляет официальные расширения для двух популярных шаблонизаторов: Smarty и Twig. Шаблонизатор Prado больше не поддерживается. Для использования данных шаблонизаторов необходимо настроить компонент приложения `view` задав свойство `View::$renderers`. Подробнее об этом можно прочитать в разделе «Шаблонизаторы».

1.2.9 Модели

Yii 2.0 использует в качестве базового класса для моделей `yii\base\Model`, аналогичный классу `CModel` в версии 1.1. Класс `CFormModel` удалён. Вместо него для создания модели формы в Yii 2.0 вы должны напрямую наследоваться от `yii\base\Model`.

Появился новый метод `scenarios()` для объявления поддерживаемых сценариев, и для обозначения в каком сценарии атрибуты должны проверяться, считаться безопасными и т.п. Например,

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

В примере выше, объявлено два сценария: `backend` и `frontend`. Для `backend` сценария, оба атрибута `email` и `role` являются безопасными, и могут быть массово присвоены. Для сценария `frontend`, атрибут `email` может быть массово присвоен, а атрибут `role` нет. Оба атрибута `email` и `role` должны быть проверены с помощью правил валидации.

Метод `rules()` по-прежнему используется для объявления правил валидации. Обратите внимание, что в связи с появлением нового метода `scenarios()`, больше не поддерживается валидатор `unsafe`.

В большинстве случаев вам не нужно переопределять метод `scenarios()`, если метод `rules()` полностью указывает все существующие сценарии, и если нет надобности в объявлении атрибутов небезопасными.

Более детальная информация представлена в разделе «Модели».

1.2.10 Контроллеры

В качестве базового класса для контроллеров в Yii 2.0 используется `yii\web\Controller`, аналогичный `CController` в Yii 1.1. Базовым классом для всех действий является `yii\base\Action`.

Одним из основных изменений является то, что действие контроллера теперь должно вернуть результат вместо того, чтобы напрямую выводить его:

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundException;
    }
}
```

Более детальная информация представлена в разделе «Контроллеры».

1.2.11 Виджеты

В Yii 2.0 класс `yii\base\Widget` используется в качестве базового класса для виджетов, аналогично `CWidget` в Yii 1.1.

Для лучшей поддержки фреймворка в IDE, Yii 2.0 использует новый синтаксис для виджетов. Новые статические методы `begin()`, `end()`, и `widget()` используются следующим образом:

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

// Обратите внимание что вы должны выводить результат
echo Menu::widget(['items' => $items]);

// Указываем массив для конфигурации свойств объекта
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... поля формы ...
ActiveForm::end();
```

Более детальная информация представлена в разделе «Виджеты».

1.2.12 Темы

В Yii 2.0 темы работают совершенно по-другому. Теперь они основаны на механизме сопоставления путей исходного файла представления с темизированным файлом. Например, если используется сопоставление путей `['/web/views' => '/web/themes/basic']`, то темизированная версия файла представления `/web/views/site/index.php` будет находиться в `/web/themes/basic/site/index.php`. По этой причине темы могут быть применены к любому файлу представления, даже к представлению, отрендеренному внутри контекста контроллера или виджета. Также, больше не существует компонента `CThemeManager`. Вместо этого, `theme` является конфигурируемым свойством компонента приложения `view`.

Более детальная информация представлена в разделе «Темизация».

1.2.13 Консольные приложения

Консольные приложения теперь организованы как контроллеры, аналогично веб приложениям. Консольные контроллеры должны быть унаследованы от класса `yii\console\Controller`, аналогичного `CConsoleCommand` в версии 1.1.

Для выполнения консольной команды, используйте `yii маршрут<>`, где `маршрут<>` это маршрут контроллера (например, `sitemap/index`). Дополнительные анонимные аргументы будут переданы в качестве параметров соответствующему действию контроллера, в то время как именованные аргументы будут переданы в соответствие с объявлениями в `yii\console\Controller::options()`.

Yii 2.0 поддерживает автоматическую генерацию справочной информации из блоков комментариев.

Более детальная информация представлена в разделе «Консольные команды».

1.2.14 I18N

В Yii 2.0 встроенные форматтеры времени и чисел были убраны в пользу PECL расширения PHP intl¹².

Перевод сообщений теперь осуществляется через компонент приложения `i18n`. Данный компонент управляет множеством исходных хранилищ сообщений, что позволяет вам использовать разные хранилища для исходных сообщений в зависимости от категории сообщения.

Более детальная информация представлена в разделе «Интернационализация».

¹²<http://pecl.php.net/package/intl>

1.2.15 Фильтры действий

Фильтры действий теперь сделаны с помощью поведений. Для определения нового фильтра, унаследуйтесь от `yii\base\ActionFilter`. Для использования фильтра, прикрепите его к контроллеру в качестве поведения. Например, для использования фильтра `yii\filters\AccessControl`, следует сделать следующее:

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

Более детальная информация представлена в разделе «Фильтры».

1.2.16 Ресурсы

В Yii 2.0 представлена новая возможность *связка ресурсов*, которая заменяет концепт пакетов скриптов в Yii 1.1.

Связка ресурсов — это коллекция файлов ресурсов (например, JavaScript файлы, CSS файлы, файлы изображений, и т.п.) в определенной директории. Каждая связка ресурсов представлена классом, унаследованным от `yii\web\AssetBundle`. Связка ресурсов становится доступной через веб после её регистрации методом `yii\web\AssetBundle::register()`. В отличие от Yii 1.1, страница, регистрирующая связку ресурсов, автоматически будет содержать ссылки на JavaScript и CSS файлы, указанные в связке.

Более детальная информация представлена в разделе «Ресурсы».

1.2.17 Хелперы

В Yii 2.0 включено много широко используемых статичных классов.

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

Более детальная информация представлена в разделе «Хелперы».

1.2.18 Формы

Yii 2.0 вводит новое понятие *поле* для построения форм с помощью `yii\widgets\ActiveForm`. Поле — это контейнер, содержащий подпись, поле ввода, сообщение об ошибке и/или вспомогательный текст. Поле представлено объектом `ActiveField`. Используя поля, вы можете строить формы гораздо проще чем это было раньше:

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

Более детальная информация представлена в разделе «Работа с формами».

1.2.19 Построитель запросов

В версии 1.1, построение запроса было разбросано среди нескольких классов, включая `CDbCommand`, `CDbCriteria`, и `CDbCommandBuilder`. В Yii 2.0 запрос к БД представлен в рамках объекта `Query`, который может быть превращён в SQL выражение с помощью `QueryBuilder`. Например,

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

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

Лучшим способом использования данных методов является работа с `Active Record`.

Более детальная информация представлена в разделе «Построитель запросов».

1.2.20 Active Record

В Yii 2.0 внесено множество изменений в работу `Active Record`. Два основных из них включают в себя построение запросов и работу со связями.

Класс `CDbCriteria` версии 1.1 был заменен `yii\db\ActiveQuery`. Этот класс наследуется от `yii\db\Query` и таким образом получает все методы, необходимые для построения запроса. Чтобы начать строить запрос следует вызвать метод `yii\db\ActiveRecord::find()`:


```
// Получаем всех активных** клиентов и сортируем их по ID
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

Для объявления связи следует просто объявить геттер, который возвращает объект `ActiveQuery`. Имя свойства, определённое геттером, представляет собой название связи. Например, следующий код объявляет связь `orders` (в версии 1.1, вам нужно было бы объявить связи в одном месте — методе `relations()`):

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

Теперь вы можете использовать выражение `$customer->orders` для получения всех заказов клиента из связанной таблицы. Вы также можете использовать следующий код, чтобы применить нужные условия «на лету»:

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

Yii 2.0 осуществляет жадную загрузку связи не так, как это было в 1.1. В частности, в версии 1.1 для выбора данных из основной и связанной таблиц будет использован запрос `JOIN`. В Yii 2.0 будут выполнены два запроса без использования `JOIN`: первый запрос возвращает данные для основной таблицы, а второй, осуществляющий фильтрацию по первичным ключами основной таблицы — для связанной.

Вместо того, чтобы при выборке большого количества записей возвращать объекты `ActiveRecord`, вы можете использовать в построении запроса метод `asArray()`. Это заставит вернуть результат запроса в виде массива, что при большом количестве записей может существенно снизить затрачиваемое процессорное время и объём потребляемой памяти. Например:

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

Ещё одно изменение связано с тем, что вы больше не можете определять значения по умолчанию через `public` свойства. Вы должны установить их в методе `init` вашего класса, если это требуется.

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

Также в версии 1.1 были некоторые проблемы с переопределением конструктора `ActiveRecord`. Данные проблемы отсутствуют в версии 2.0. Обратите внимание, что при добавлении параметров в конструктор, вам, возможно, понадобится переопределить метод `yii\db\ActiveRecord::instantiate()`.

Существует также множество других улучшений в `ActiveRecord`. Подробнее о них можно узнать в разделе «Active Record».

1.2.21 Поведения Active Record

В версии 2.0 отсутствует базовый класс для поведений `CActiveRecordBehavior`. Если вам необходимо создать поведение для `Active Record`, стоит наследовать его класс напрямую от `yii\base\Behavior`. Если поведение должно реагировать на какие-либо события, необходимо перекрыть метод `events()` следующим образом:

```
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 и IdentityInterface

Класс `CWebUser` из версии 1.1 теперь заменён классом `yii\web\User`. Также больше не существует класса `CUserIdentity`. Вы должны реализовать интерфейс `yii\web\IdentityInterface`, что гораздо проще. Пример реализации представлен в шаблоне приложения `advanced`.

Более подробная информация представлена в разделах «Аутентификация», «Авторизация» и «Шаблон приложения `advanced`».

1.2.23 Разбор и генерация URL

Работа с URL в Yii 2.0 аналогична той, что была в версии 1.1. Основное изменение заключается в том, что теперь поддерживаются дополнительные параметры. Например, если у вас имеется правило, объявленное следующим образом, то оно совпадет с `post/popular` и `post/1/popular`. В версии 1.1, вам пришлось бы использовать два правила, для достижения того же результата.

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

Более детальная информация представлена в разделе «Разбор и генерация URL».

1.2.24 Использование Yii 1.1 вместе с 2.x

Информация об использовании кода для Yii 1.1 вместе с Yii 2.0 представлена в разделе «Одновременное использование Yii 1.1 и 2.0».

Глава 2

Первое знакомство

2.1 Установка Yii

Вы можете установить Yii двумя способами: используя Composer¹ или скачав архив. Первый способ предпочтительнее так как позволяет устанавливать новые расширения или обновить Yii одной командой.

Примечание: В отличие от Yii 1, после стандартной установки Yii 2 мы получаем как фреймворк, так и шаблон приложения.

2.1.1 Установка при помощи Composer

Если Composer еще не установлен это можно сделать по инструкции на getcomposer.org², или одним из нижеперечисленных способов. На Linux или Mac используйте следующую команду:

```
curl -s http://getcomposer.org/installer | php mv composer.phar /usr/local/bin/composer
```

На Windows, скачайте и запустите Composer-Setup.exe³.

В случае возникновения проблем или если вам необходима дополнительная информация, обращайтесь к документации Composer⁴.

Если у вас уже установлен Composer, обновите его при помощи `composer self-update`.

После установки Composer устанавливать Yii можно запустив следующую команду в папке доступной через веб:

```
composer global require "fxp/composer-asset-plugin:1.0.0"
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

¹<http://getcomposer.org/>

²<https://getcomposer.org/download/>

³<https://getcomposer.org/Composer-Setup.exe>

⁴<https://getcomposer.org/doc/>

Первая команда устанавливает `composer asset plugin`⁵, который позволяет управлять зависимостями пакетов `bower` и `npm` через `Composer`. Эту команду достаточно выполнить один раз. Вторая команда устанавливает `Yii` в директорию `basic`. Если хотите, можете выбрать другое имя директории.

`Composer` установит `Yii` (шаблонное приложение `basic`) в папку `basic`.

Примечание: В процессе установки `Composer` может запросить логин и пароль от `Github` потому как у `API Github` имеется ограничение на количество запросов. Это нормально потому как `Composer` в процессе работы запрашивает у `Github` большое количество информации для каждого пакета. Вход на `Github` повышает ограничение по запросам `API` и `Composer` может продолжить свою работу. Подробнее об этом можно прочитать в документации `Composer`⁶.

Подсказка: Если вы хотите установить последнюю нестабильную ревизию `Yii`, можете использовать следующую команду, в которой присутствует опция `stability`⁷:

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

Старайтесь не использовать нестабильную версию `Yii` на рабочих серверах потому как она может внезапно поломать код.

2.1.2 Установка из архива

Установка `Yii` из архива состоит из трёх шагов:

1. Скачайте архив с `yiiframework.com`⁸;
2. Распакуйте скачанный архив в папку, доступную из `Web`.
3. В файле `config/web.php` добавьте секретный ключ в значение `cookieValidationKey` (при установке через `Composer` это происходит автоматически):

```
// !!! insert a secret key in the following (if it is empty) - this is
      required by cookie validation
'cookieValidationKey' => 'enter your secret key here',
```

⁵<https://github.com/francoispluchino/composer-asset-plugin/>

⁶<https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens>

⁷<https://getcomposer.org/doc/04-schema.md#minimum-stability>

⁸<http://www.yiiframework.com/download/>

2.1.3 Другие опции установки

Выше приведены инструкции по установке Yii в виде базового приложения готового к работе. Это отличный вариант для небольших проектов или для тех, кто только начинает изучать Yii.

Есть два основных варианта такой установки:

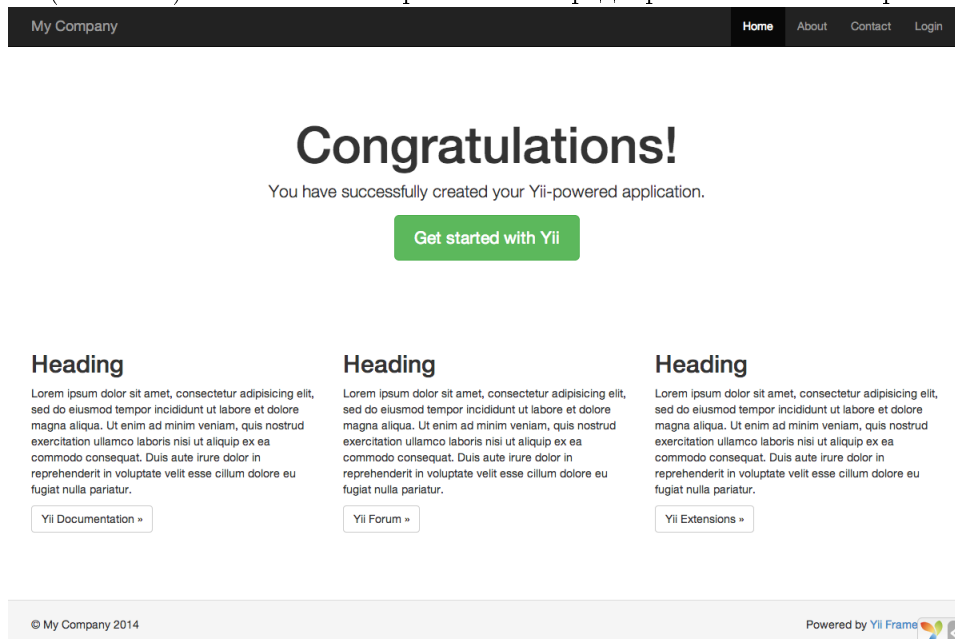
- Если вам нужен только сам фреймворк и вы хотели бы создать приложение с нуля, воспользуйтесь инструкцией, описанной в разделе «Создание приложения с нуля».
- Если хотите начать с более продвинутого приложения, хорошо подходящего для работы в команде, используйте шаблон приложения advanced.

2.1.4 Проверка установки

После установки приложение будет доступно по следующему URL:

```
http://localhost/basic/web/index.php
```

Здесь подразумевается, что вы установили приложение в директорию **basic** в корневой директории вашего веб сервера сервер работает локально (**localhost**). Вам может потребоваться предварительно его настроить.



Вы должны увидеть страницу приветствия «Congratulations!». Если нет — проверьте требования Yii одним из способов:

- Браузером перейдите по адресу `http://localhost/basic/requirements.php`
- Или выполните команду в консоли:

```
cd basic
```

php requirements.php

Для корректной работы фреймворка вам необходима установка PHP, соответствующая его минимальным требованиям. Основное требование — PHP версии 5.4 и выше. Если ваше приложение работает с базой данных, необходимо установить расширение PHP PDO⁹ и соответствующий драйвер (например, `pdo_mysql` для MySQL).

2.1.5 Настройка веб сервера

Информация: можете пропустить этот подраздел если вы только начали знакомиться с фреймворком и пока не разворачиваете его на рабочем сервере.

Приложение, установленное по инструкциям, приведённым выше, будет работать сразу как с Apache¹⁰, так и с Nginx¹¹ под Windows и Linux с установленным PHP 5.4 и выше. Yii 2.0 также совместим с HHVM¹². Тем не менее, в некоторых случаях поведение при работе с HHVM отличается от обычного PHP. Будьте внимательны.

На рабочем сервере вам наверняка захочется изменить URL приложения с `http://www.example.com/basic/web/index.php` на `http://www.example.com/index.php`. Для этого необходимо изменить корневую директорию в настройках веб сервера так, чтобы та указывала на `basic/web`. Дополнительно можно спрятать `index.php` следуя описанию в разделе «Разбор и генерация URL». Далее будет показано как настроить Apache и Nginx.

Информация: Устанавливая `basic/web` корневой директорией веб сервера вы защищаете от нежелательного доступа код и данные, находящиеся на одном уровне с `basic/web`. Это делает приложение более защищенным.

Информация: Если приложение работает на хостинге где нет доступа к настройкам веб сервера, то можно изменить структуру приложения как описано в разделе «Работа на Shared хостинге».

Рекомендуемые настройки Apache

Добавьте следующее в `httpd.conf` Apache или в конфигурационный файл виртуального хоста. Не забудьте заменить `path/to/basic/web` на корректный путь к `basic/web`.

⁹<http://www.php.net/manual/ru/pdo.installation.php>

¹⁰<http://httpd.apache.org/>

¹¹<http://nginx.org/>

¹²<http://hhvm.com/>


```
# Устанавливаем корневой директорией "basic/web"
DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">
    RewriteEngine on

    # Если запрашиваемая в URL директория или файл существуют обращаемся к
    # ним напрямую
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # Если нет - перенаправляем запрос на index.php
    RewriteRule . index.php

    # прочие... настройки...
</Directory>
```

Рекомендуемые настройки Nginx

PHP должен быть установлен как FPM SAPI¹³ для Nginx¹⁴. Используйте следующие параметры Nginx и не забудьте заменить `path/to/basic/web` на корректный путь к `basic/web` и `mysite.local` на ваше имя хоста.

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

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

    server_name mysite.local;
    root        /path/to/basic/web;
    index       index.php;

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

    location / {
        # Перенаправляем все запросы к несуществующим директориям и файлам
        # на index.php
        try_files $uri $uri/ /index.php?$args;
    }

    # раскомментируйте строки ниже во избежание обработки Yii обращений к
    # несуществующим статическим файлам
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    location ~ \.php$ {
```

¹³<http://php.net/manual/ru/install.fpm.php>

¹⁴<http://wiki.nginx.org/>

```
include fastcgi.conf;
fastcgi_pass 127.0.0.1:9000;
#fastcgi_pass unix:/var/run/php5-fpm.sock;
}

location ~ /\. (ht|svn|git) {
    deny all;
}
}
```

Используя данную конфигурацию установите `cgi.fix_pathinfo=0` в `php.ini` чтобы предотвратить лишние системные вызовы `stat()`.

Учтите, что используя HTTPS необходимо задавать `fastcgi_param HTTPS on`; чтобы Yii мог корректно определять защищенное соединение.

2.2 Запуск приложения

После установки Yii базовое приложение будет доступно либо по URL `http://hostname/basic/web/index.php`, либо по `http://hostname/index.php`, в зависимости от настроек Web сервера. Данный раздел - общее введение в организацию кода, встроенный функционал и обработку обращений приложением Yii.

Информация: далее в данном руководстве предполагается, что Yii установлен в директорию `basic/web`, которая, в свою очередь, установлена как корневой каталог в настройках Web сервера. В результате, обратившись по URL `http://hostname/index.php`, Вы получите доступ к приложению, расположенному в `basic/web`. Детальнее с процессом начальной настройки можно познакомиться в разделе Установка Yii.

2.2.1 Функциональность

Установленный шаблон простого приложения состоит из четырех страниц:

- домашняя страница, отображается при переходе по URL `http://hostname/index.php`
- страница “About” (“О нас”)
- на странице “Contact” находится форма обратной связи, на которой пользователь может обратиться к разработчику по e-mail
- на странице “Login” отображается форма авторизации. Попробуйте авторизоваться с логином/паролем “admin/admin”. Обратите внимание на изменение раздела “Login” в главном меню на “Logout”.

Эти страницы используют смежный хедер (шапка сайта) и футер (подвал). В “шапке” находится главное меню, при помощи которого пользо-

ватель перемещается по сайту. В “подвале” - копирайт и общая информация.

В самой нижней части окна Вы будете видеть системные сообщения Yii - журнал, отладочную информацию, сообщения об ошибках, запросы к базе данных и т.п. Выводом данной информации руководит встроенный отладчик, он записывает и отображает информацию о ходе выполнения приложения.

В дополнение к веб приложению имеется консольный скрипт с названием `yii`, который находится в базовой директории приложения. Этот скрипт может быть использован для выполнения фоновых задач и обслуживания приложения. Всё это описано в разделе Консольные команды.

2.2.2 Структура приложения Yii

Ниже приведен список основных директорий и файлов вашего приложения (считаем, что приложение установлено в директорию `basic`):

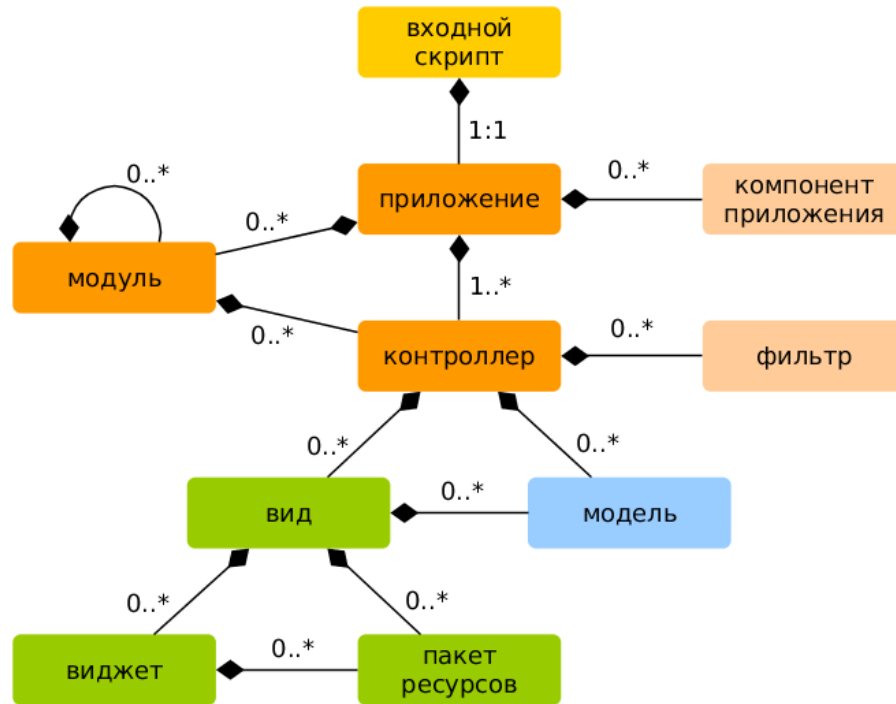
<code>basic/</code>	корневой каталог приложения
<code>composer.json</code>	используется Composerом ¹⁵ , содержит описание приложения
<code>config/</code>	конфигурационные файлы
<code> console.php</code>	конфигурация консольного приложения
<code> web.php</code>	конфигурация Web приложения
<code>commands/</code>	содержит классы консольных команд
<code>controllers/</code>	контроллеры
<code>models/</code>	модели
<code>runtime/</code>	файлы, которые генерирует Yii во время выполнения приложения логи(, кэш и тп..)
<code>vendor/</code>	содержит пакеты Composerа ¹⁵ и, собственно, сам фреймворк Yii
<code>views/</code>	виды приложения
<code>web/</code>	корневая директория Web приложения. Содержит файлы, доступные через Web
<code> assets/</code>	скрипты, используемые приложением (js, css)
<code> index.php</code>	точка входа в приложение Yii. С него начинается выполнение приложения
<code>yii</code>	скрипт выполнения консольного приложения Yii

В целом, приложение Yii можно разделить на две категории файлов: расположенные в `basic/web` и расположенные в других директориях. Первая категория доступна через Web (например, браузером), вторая не доступна из вне и не должна быть доступной т.к. содержит служебную информацию.

В Yii реализована схема проектирования модель-вид-контроллер (MVC)¹⁵, которая соответствует структуре директорий приложения. В директории `models` находятся Модели, в `views` расположены Видеы, а в каталоге `controllers` все Контроллеры приложения.

¹⁵<http://ru.wikipedia.org/wiki/Model-View-Controller>

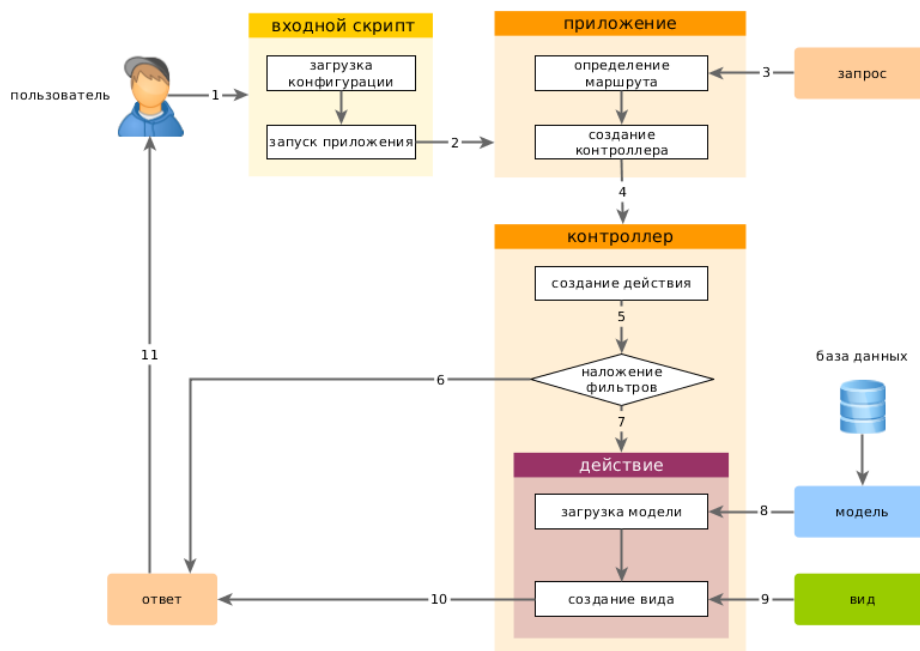
Диаграмма ниже демонстрирует внутреннее устройство приложения.



В каждом приложении Yii есть точка входа в приложение, `web/index.php` это единственный PHP-скрипт доступный для выполнения из Web. Он принимает входящий запрос и создает экземпляр приложения. Приложение обрабатывает входящие запросы при помощи компонентов и отправляет запрос контроллеру. Виджеты используются в Видах для построения динамических интерфейсов сайта.

2.2.3 Жизненный цикл пользовательского запроса

На диаграмме показано как приложение обрабатывает запрос.



1. Пользователь обращается к точке входа `web/index.php`.
2. Скрипт загружает конфигурацию `configuration` и создает экземпляр приложения для дальнейшей обработки запроса.
3. Приложение определяет маршрут запроса при помощи компонента приложения `запрос`.
4. Приложение создает экземпляр контроллера для выполнения запроса.
5. Контроллер, в свою очередь, создает действие и накладывает на него фильтры.
6. Если хотя бы один фильтр дает сбой, выполнение приложения останавливается.
7. Если все фильтры пройдены - приложение выполняется.
8. Действие загружает модель данных. Вероятнее всего из базы данных.
9. Действие генерирует вид, отображая в нем данные (в т.ч. и полученные из модели).
10. Сгенерированный вид приложения передается как компонент ответа.

11. Компонент “ответ” отправляет готовый результат работы приложения браузеру пользователя.

2.3 Говорим «Привет»

В этом разделе рассмотрим как создать новую страницу с надписью «Привет». В процессе решения задачи вы создадите действие контроллера и представление:

- Приложение обработает запрос и передаст управление соответствующему действию;
- Действие, в свою очередь, отобразит представление с надписью “Привет” конечному пользователю.

С помощью данного руководства вы изучите

- Как создавать действие, чтобы отвечать на запросы;
- Как создавать представление, чтобы формировать содержимое ответа;
- Как приложение отправляет запросы к действию.

2.3.1 Создание Действия

Для нашей задачи потребуется действие `say`, которое читает параметр `message` из запроса и отображает его значение пользователю. Если в запросе не содержится параметра `message`, то действие будет выводить «Привет».

Информация: Действия могут быть запущены непосредственно пользователем и сгруппированы в контроллеры. Результатом выполнения действия является ответ, который получает пользователь.

Действия объявляются в контроллерах. Для простоты, вы можете объявить действие `say` в уже существующем контроллере `SiteController`, который определен в файле класса `controllers/SiteController.php`:

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // существующий... код...

    public function actionSay($message = 'Привет')
    {
```

```
        return $this->render('say', ['message' => $message]);  
    }  
}
```

В приведенном коде действие `say` объявлено как метод `actionSay` в классе `SiteController`. Yii использует префикс `action` чтобы различать методы-действия и обычные методы. Название после префикса `action` считается идентификатором соответствующего действия.

Информация: Идентификаторы действий задаются в нижнем регистре. Если идентификатор состоит из нескольких слов, они соединяются дефисами, то есть `create-comment`. Имена методов действий получаются путём удаления дефисов из идентификатора, преобразования первой буквы каждого слова в верхний регистр и добавления префикса `action`. Например, идентификатор действия `create-comment` соответствует методу `actionCreateComment`.

Метод действия принимает параметр `$message`, который по умолчанию равен `"Привет"`. Когда приложение получает запрос и определяет, что действие `say` ответственно за его обработку, параметр заполняется одноимённым значением из запроса.

Внутри метода действия, для вывода отображения представления с именем `say`, используется метод `render()`. Для того, чтобы вывести сообщение, в отображение передаётся параметр `message`. Результат отображения при помощи `return` передаётся приложению, которое отдаёт его пользователю.

2.3.2 Создание представления

Представления являются скриптами, которые используются для формирования тела ответа. Для нашего приложения вы создадите представление `say`, которое будет выводить параметр `message`, полученный из метода действия:

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

Представление `say` должно быть сохранено в файле `views/site/say.php`. Когда метод `render()` вызывается в действии, он будет искать РНР файл с именем вида `views/ControllerID/ViewName.php`.

Стоит отметить, что в коде выше параметр `message` экранируется для HTML перед выводом. Это обязательно так как параметр приходит от

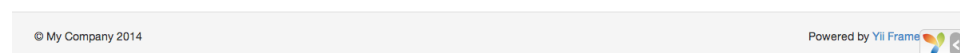
пользователя, который может попытаться провести XSS атаку¹⁶ путём вставки зловредного JavaScript кода.

Вы можете дополнить представление `say` HTML тегами, текстом или кодом PHP. Фактически, представление `say` является простым PHP скриптом, который выполняется методом `render()`. Содержимое, выводимое скриптом представления, будет передано пользователю приложением.

2.3.3 Попробуем

После создания действия и представления вы можете перейти на новую страницу по следующему URL:

```
http://hostname/index.php?r=site/say&messageПриветмир=+
```



Будет отображена страница с надписью «Привет мир». Она использует ту же шапку и футер, что и остальные страницы приложения. Если вы не укажете параметр `message`, то увидите на странице «Привет». Это происходит потому, как `message` передаётся в метод `actionSay()` и значение по умолчанию — «Привет».

Информация: Новая страница использует ту же шапку и футер, что и другие страницы, потому что метод `render()` автоматически вставляет результат представления `say` в, так называемый, макет `views/layouts/main.php`.

¹⁶http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D1%81%D0%B0%D0%B9%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%B8%D0%BD%D0%B3

Параметр `r` требует дополнительных пояснений. Он связан с маршрутом (route), который представляет собой уникальный идентификатор, указывающий на действие. Его формат `ControllerID/ActionID`. Когда приложение получает запрос, оно проверяет параметр `r` и, используя `ControllerID`, определяет какой контроллер следует использовать для обработки запроса. Затем, контроллер использует часть `ActionID`, чтобы определить какое действие выполняет реальную работу. В нашем случае маршрут `site/say` будет соответствовать контроллеру `SiteController` и его действию `say`. В результате, для обработки запроса будет вызван метод `SiteController::actionSay()`.

Информация: Как и действия, контроллеры также имеют идентификаторы, которые однозначно определяют их в приложении. Идентификаторы контроллеров используют те же правила именования, что и идентификаторы действий. Имена классов контроллеров получаются путём удаления дефисов из идентификатора, преобразования первой буквы каждого слова в верхний регистр и добавления в конец `Controller`. Например, идентификатор контроллера `post-comment` соответствует имени класса контроллера `PostCommentController`.

2.3.4 Резюме

В этом разделе вы затронули тему контроллеров и представлений в паттерне MVC. Вы создали действие как часть контроллера, обрабатывающего запросы, и представление, участвующее в формировании ответа. В этом процессе никак не была задействована модель, так как в качестве данных выступает лишь простой параметр `message`.

Также вы познакомились с концепцией маршрутизации, которая является связующим звеном между запросом пользователя и действием контроллера.

В следующем разделе вы узнаете как создавать модели и добавлять новые страницы с HTML формами.

2.4 Работа с формами

В данном разделе мы обсудим получение данных от пользователя. На странице будет располагаться форма с полями для ввода имени и email. Полученные данные будут показаны на странице для их подтверждения.

Чтобы достичь этой цели, помимо создания действия и двух представлений вы создадите модель.

В данном руководстве вы изучите:

- Как создать модель для данных, введённых пользователем;
- Как объявить правила проверки введённых данных;

- Как создать HTML форму в представлении.

2.4.1 Создание модели

В файле `models/EntryForm.php` создайте класс модели `EntryForm` как показано ниже. Он будет использоваться для хранения данных, введенных пользователем. Подробно о именовании файлов классов читайте в разделе «Автозагрузка классов».

```
<?php

namespace app\models;

use yii\base\Model;

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

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

Данный класс расширяет класс `yii\base\Model`, который является частью фреймворка и обычно используется для работы с данными форм.

Класс содержит 2 публичных свойства `name` и `email`, которые используются для хранения данных, введенных пользователем. Он также содержит метод `rules()`, который возвращает набор правил проверки данных. Правила, объявленные в коде выше означают следующее:

- Поля `name` и `email` обязательны для заполнения;
- В поле `email` должен быть правильный адрес email.

Если объект `EntryForm` заполнен пользовательскими данными, то для их проверки вы можете вызвать метод `validate()`. В случае неудачной проверки свойство `hasErrors` станет равным `true`. С помощью `errors` можно узнать, какие именно ошибки возникли.

2.4.2 Создание действия

Далее создайте действие `entry` в контроллере `site`, точно так же, как вы делали это ранее.

```
<?php

namespace app\controllers;
```

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

class SiteController extends Controller
{
    // существующий... код...

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

        if ($model->load(Yii::$app->request->post()) && $model->validate())
        {
            // данные в $model удачно проверены

            // делаем что-то полезное с $model ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // либо страница отображается первый раз, либо есть ошибка в
            // данных
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

Действие создает объект `EntryForm`. Затем оно пытается заполнить модель данными из массива `$_POST`, доступ к которому обеспечивает `Yii` при помощи `yii\web\Request::post()`. Если модель успешно заполнена, то есть пользователь отправил данные из HTML формы, то для проверки данных будет вызван метод `validate()`.

Если всё в порядке, действие отобразит представление `entry-confirm`, которое показывает пользователю введенные им данные. В противном случае будет отображено представление `entry`, которое содержит HTML форму и ошибки проверки данных, если они есть.

Информация: `Yii::$app` представляет собой глобально доступный экземпляр-одиночку приложения (singleton). Одновременно это Service Locator, дающий доступ к компонентам вроде `request`, `response`, `db` и так далее. В коде выше для доступа к данным из `$_POST` был использован компонент `request`.

2.4.3 Создание представления

В заключение, создаём два представления с именами `entry-confirm` и `entry`, которые отображаются действием `entry` из предыдущего подраздела.

Представление `entry-confirm` просто отображает имя и email. Оно должно быть сохранено в файле `views/site/entry-confirm.php`.

```
<?php
use yii\helpers\Html;
?>
<pВы> ввели следующую информацию:</p>

<ul>
    <li><label>Name</label>: <?= Html::encode($model->name) ?></li>
    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>
</ul>
```

Представление `entry` отображает HTML форму. Оно должно быть сохранено в файле `views/site/entry.php`.

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

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

Для построения HTML формы представление использует мощный виджет `ActiveForm`. Методы `begin()` и `end()` выводят открывающий и закрывающий теги формы. Между этими вызовами создаются поля ввода при помощи метода `field()`. Первым идёт поле для “name”, вторым — для “email”. Далее для генерации кнопки отправки данных вызывается метод `yii\helpers\Html::submitButton()`.

2.4.4 Попробуем

Чтобы увидеть всё созданное в работе, откройте в браузере следующий URL:

```
http://hostname/index.php?r=site/entry
```

Вы увидите страницу с формой и двумя полями для ввода. Перед каждым полем имеется подпись, которая указывает, какую информацию следует вводить. Если вы нажмёте на кнопку отправки без ввода данных или если вы введёте email в неверном формате, вы увидите сообщение с ошибкой рядом с каждым проблемным полем.

The screenshot shows a web form for 'My Company'. The header bar contains the company name and navigation links: Home, About, Contact, and Login. The form has two input fields: 'Name' and 'Email'. Both fields are empty and have a red border, indicating they are required. Below each field is a red error message: 'Name cannot be blank' and 'Email cannot be blank'. A blue 'Submit' button is located below the email field. The footer bar contains the copyright notice '© My Company 2014' and the text 'Powered by Yii Frame' with a logo.

После ввода верных данных и их отправки, вы увидите страницу с данными, которые вы только что ввели.

The screenshot shows the result of a successful form submission. The header bar is the same as the previous screenshot. The main content area displays the message 'You have entered the following information:' followed by a bulleted list of the submitted data: 'Name: Qiang Xue' and 'Email: tester@example.com'. The footer bar is also the same as the previous screenshot.

Как работает вся эта «магия»

Вы, скорее всего, задаётесь вопросом о том, как же эта HTML форма работает на самом деле. Весь процесс может показаться немного волшебным: то как показываются подписи к полям, ошибки проверки данных

при некорректном вводе и то что всё это происходит без перезагрузки страницы.

Да, проверка данных на самом деле происходит и на стороне клиента при помощи JavaScript и на стороне сервера. `yii\widgets\ActiveForm` достаточно продуман, чтобы взять правила проверки, которые вы объявили в `EntryForm`, преобразовать их в JavaScript код и использовать его для проведения проверок. На случай отключения JavaScript в браузере валидация проводится и на стороне сервера как показано в методе `actionEntry()`. Это даёт уверенность в том, что данные корректны при любых обстоятельствах.

Подписи для полей генерируются методом `field()`, на основе имён свойств модели. Например, подпись `Name` генерируется для свойства `name`. Вы можете модифицировать подписи следующим образом:

```
<?= $form->field($model, 'name')->label('Ваше имя') ?>
<?= $form->field($model, 'email')->label('Ваш Email') ?>
```

Информация: В Yii есть множество виджетов, позволяющих быстро строить сложные и динамичные представления. Как вы узнаете позже, разрабатывать новые виджеты очень просто. Многие из представлений можно вынести в виджеты, чтобы использовать это повторно в других местах и упростить тем самым разработку в будущем.

2.4.5 Резюме

В данном разделе вы попробовали каждую часть шаблона проектирования MVC. Вы изучили как создавать классы моделей для обработки и проверки пользовательских данных.

Также, вы изучили как получать данные от пользователя и как показать данные пользователю. Это задача может занимать в процессе разработки значительное время. Yii предоставляет мощные виджеты, которые делают задачу максимально простой.

В следующем разделе вы изучите как работать с базами данных, что требуется в большинстве приложений.

2.5 Работа с базами данных

Этот раздел расскажет о том, как создать новую страницу, отображающую данные по странам, полученные из таблицы `countries` базы данных. Для достижения этой цели вам будет необходимо настроить подключение к базе данных, создать класс Active Record, определить action, и создать view.

Изучив эту часть, вы научитесь:

- Настраивать подключение к БД
- Определять класс Active Record
- Запрашивать данные, используя класс Active Record
- Отображать данные во view с использованием пагинации

Обратите внимание, чтобы усвоить этот раздел, вы должны иметь базовые знания и навыки использования баз данных. В частности, вы должны знать, как создать базу данных, и как выполнять SQL запросы, используя клиентские инструменты для работы с БД.

2.5.1 Подготавливаем базу данных

Для начала, создайте базу данных под названием `yii2basic`, из которой вы будете получать данные в вашем приложении. Вы можете создать базу данных SQLite, MySQL, PostgreSQL, MSSQL или Oracle, так как Yii имеет встроенную поддержку для многих баз данных. Для простоты, в дальнейшем описании будет подразумеваться MySQL.

После этого создайте в базе данных таблицу `country`, и добавьте в неё немного демонстрационных данных. Вы можете запустить следующую SQL инструкцию, чтобы сделать это:

```
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',18886000);  
INSERT INTO 'country' VALUES ('BR','Brazil',170115000);  
INSERT INTO 'country' VALUES ('CA','Canada',1147000);  
INSERT INTO 'country' VALUES ('CN','China',1277558000);  
INSERT INTO 'country' VALUES ('DE','Germany',82164700);  
INSERT INTO 'country' VALUES ('FR','France',59225700);  
INSERT INTO 'country' VALUES ('GB','United Kingdom',59623400);  
INSERT INTO 'country' VALUES ('IN','India',1013662000);  
INSERT INTO 'country' VALUES ('RU','Russia',146934000);  
INSERT INTO 'country' VALUES ('US','United States',278357000);
```

На данный момент у вас есть база данных под названием `yii2basic`, и внутри неё таблица `country` с тремя столбцами, содержащими десять строк данных.

2.5.2 Настраиваем подключение к БД

Перед продолжением убедитесь, что у вас установлены PHP-расширение PDO¹⁷ и драйвер PDO для используемой вами базы данных (н-р `pdo_mysql` для MySQL). Это базовое требование в случае использования вашим

¹⁷<http://www.php.net/manual/en/book.pdo.php>

приложением реляционной базы данных. После того, как они установлены, откройте файл `config/db.php` и измените параметры на верные для вашей базы данных. По умолчанию этот файл содержит следующее:

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

Файл `config/db.php` - типичный конфигурационный инструмент, базирующийся на файлах. Данный конфигурационный файл определяет параметры, необходимые для создания и инициализации экземпляра `yii\db\Connection`, через который вы можете делать SQL запросы к подразумеваемой базе данных.

Подключение к БД, настроенное выше, доступно в коде приложения через выражение `Yii::$app->db`.

Информация: файл `config/db.php` будет подключен главной конфигурацией приложения `config/web.php`, описывающей то, как экземпляр приложения должен быть инициализирован. Для детальной информации, пожалуйста, обратитесь к разделу Конфигурации.

2.5.3 Создаём потомка Active Record

Чтобы представлять и получать данные из таблицы `country`, создайте класс - потомок `Active Record`, под названием `Country`, и сохраните его в файле `models/Country.php`.

```
<?php
namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

Класс `Country` наследуется от `yii\db\ActiveRecord`. Вам не нужно писать ни строчки кода внутри него! С кодом, приведённым выше, `Yii` свяжет имя таблицы с именем класса.

Информация: Если нет возможности задать прямой зависимости между именем таблицы и именем класса, вы можете пе-

реопределить метод `yii\db\ActiveRecord::tableName()`, чтобы явно задать имя связанной таблицы.

Используя класс `Country`, вы можете легко манипулировать данными в таблице `country`, как показано в этих фрагментах:

```
use app\models\Country;

// получаем все строки из таблицы "country" и сортируем их по "name"
$countries = Country::find()->orderBy('name')->all();

// получаем строку с первичным ключом "US"
$country = Country::findOne('US');

// отобразит "United States"
echo $country->name;

// меняем имя страны на "U.S.A." и сохраняем в базу данных
$country->name = 'U.S.A.';
$country->save();
```

Информация: Active Record - мощный способ доступа и манипулирования данными БД в объектно-ориентированном стиле. Вы можете найти подробную информацию в разделе Active Record. В качестве альтернативы, вы также можете взаимодействовать с базой данных, используя более низкоуровневый способ доступа, называемый Data Access Objects.

2.5.4 Создаём Action

Для того, чтобы показать данные по странам конечным пользователям, вам надо создать новый action. Вместо размещения нового action'a в контроллере `site`, как вы делали в предыдущих разделах, будет иметь больше смысла создать новый контроллер специально для всех действий, относящихся к данным по странам. Назовите новый контроллер `CountryController`, и создайте action `index` внутри него, как показано ниже.

```
<?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,
        ]);
    }
}

```

Сохраните код выше в файле `controllers/CountryController.php`.

Action `index` вызывает `Country::find()`. Данный метод Active Record строит запрос к БД и извлекает все данные из таблицы `country`. Чтобы ограничить количество стран, возвращаемых каждым запросом, запрос разбивается на страницы с помощью объекта `yii\data\Pagination`. Объект `Pagination` служит двум целям: *Устанавливает пункты `offset` и `limit` для SQL инструкции, представленной запросом, чтобы она возвращала только одну страницу данных за раз (в нашем случае максимум 5 строк на страницу)*. Он используется во `view` для отображения пагинатора, состоящего из набора кнопок с номерами страниц, это будет разъяснено в следующем подразделе.

В конце кода action `index` выводит `view` с именем `index`, и передаёт в него данные по странам вместе с информацией о пагинации.

2.5.5 Создаём View

Первым делом создайте поддиректорию с именем `country` внутри директории `views`. Эта папка будет использоваться для хранения всех `view`, выводимых контроллером `country`. Внутри директории `views/country` создайте файл с именем `index.php`, содержащий следующий код:

```

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

```

```
<?php endforeach; ?>
</ul>

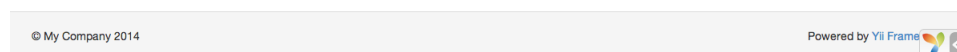
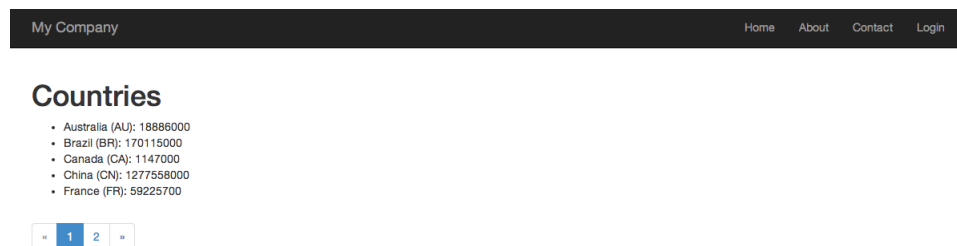
<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

View имеет 2 части относительно отображения данных по странам. В первой части предоставленные данные по странам выводятся как неупорядоченный HTML-список. Во второй части выводится виджет `yii\widgets\LinkPager`, используя информацию о пагинации, переданную из action во view. Виджет `LinkPager` отображает набор постраничных кнопок. Клик по любой из них обновит данные по странам в соответствующей странице.

2.5.6 Испытываем в действии

Чтобы увидеть, как работает весь вышеприведённый код, перейдите по следующей ссылке в своём браузере:

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



В начале вы увидите страницу, показывающую пять стран. Под странами вы увидите пагинатор с четырьмя кнопками. Если вы кликните по кнопке "2", то увидите страницу, отображающую другие пять стран из базы данных: вторая страница записей. Посмотрев внимательней, вы увидите, что URL в браузере тоже сменилось на

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

За кадром, `Pagination` предоставляет всю необходимую функциональность для постраничной разбивки набора данных: *В начале `Pagination`*

показывает первую страницу, которая отражает *SELECT* запрос стран с параметрами *LIMIT 5 OFFSET 0*. Как результат, первые пять стран будут получены и отображены. Виджет **LinkPager** выводит кнопки страниц используя URL'ы, созданные **Pagination**. Эти URL'ы будут содержать параметр запроса **page**, который представляет различные номера страниц. * Если вы кликните по кнопке "2", сработает и обработается новый запрос для маршрута *country/index*. Таким образом новый запрос стран будет иметь параметры *LIMIT 5 OFFSET 5* и вернет следующие пять стран для отображения.

2.5.7 Резюме

В этом разделе вы научились работать с базой данных. Также вы научились получать и отображать данные с постраничной разбивкой с помощью `yii\data\Pagination` и `yii\widgets\LinkPager`.

В следующем разделе вы научитесь использовать мощный инструмент генерации кода, называемый **Gii**, чтобы с его помощью быстро осуществлять некоторые частоиспользуемые функции, такие, как операции Create-Read-Update-Delete (CRUD) для работы с данными в таблице базы данных. На самом деле код, который вы только что написали, в **Yii** может быть полностью сгенерирован автоматически с использованием **Gii**.

2.6 Генерация кода при помощи Gii

В этом разделе мы опишем, как использовать **Gii** для автоматической генерации кода, реализующего некоторые общие функции вебсайта. Для достижения этой цели всё, что вам нужно, это просто ввести необходимую информацию в соответствии с инструкциями, отображаемыми на веб-страницах **Gii**.

В этом руководстве вы узнаете:

- Как активировать **Gii** в приложении;
- Как использовать **Gii** для создания **Active Record** класса;
- Как использовать **Gii** для генерации кода, реализующего **CRUD** для таблицы БД.
- Как настроить код, генерируемый **Gii**.

2.6.1 Запускаем Gii

Gii представлен в **Yii** как модуль. Вы можете активировать **Gii**, настроив его в свойстве **modules**. В зависимости от того, каким образом вы создали приложение, вы можете удостовериться в наличии следующего кода в конфигурационном файле `config/web.php`,

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

Приведенная выше конфигурация показывает, что находясь в режиме разработки, приложение должно включать в себя модуль с именем `gii`, который реализует класс `yii\gii\Module`.

Если вы посмотрите входной скрипт `web/index.php` вашего приложения, вы увидите следующую строку, устанавливающую константу `YII_ENV_DEV` в значение `true`.

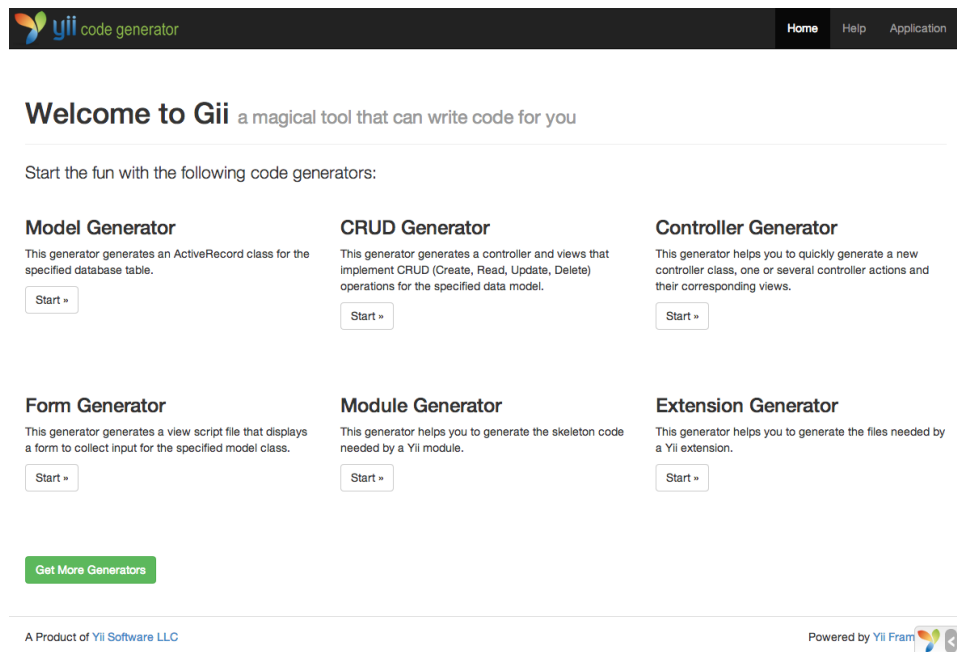
```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Благодаря этой строке ваше приложение находится в режиме разработки, и Gii уже активирован в соответствии с описанной выше конфигурацией. Теперь вы можете получить доступ к Gii по следующему адресу:

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

Замечание: Если вы пытаетесь получить доступ к Gii не с локального хоста, по умолчанию, в целях обеспечения безопасности, доступ будет запрещён. Вы можете изменить настройки Gii, чтобы добавить разрешённые IP адреса, как указано ниже

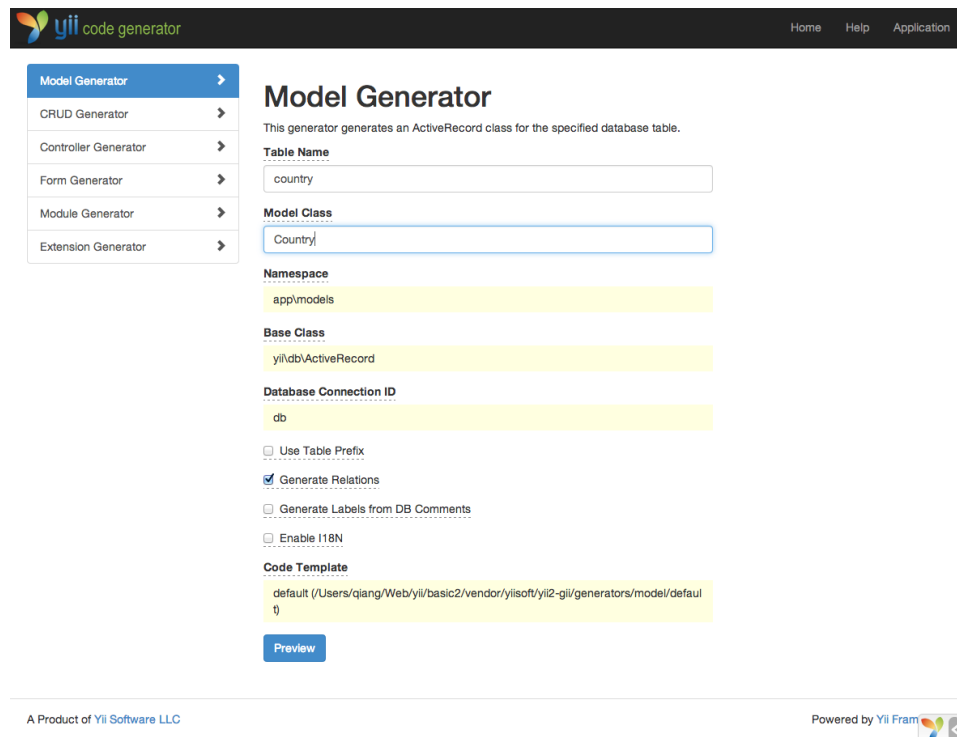
```
'gii' => [  
    'class' => 'yii\gii\Module',  
    'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'] //  
    регулируйте в соответствии со своими нуждами  
],
```



2.6.2 Генерация класса Active Record

Чтобы использовать Gii для генерации класса Active Record, выберите “Генератор модели” (нажав на ссылку на главной странице Gii). И заполните форму следующим образом:

- Имя таблицы: `country`
- Класс модели : `Country`



Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name

country

Model Class

Country

Namespace

app\models

Base Class

yii\db\ActiveRecord

Database Connection ID

db

☐ Use Table Prefix

☒ Generate Relations

☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template

default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default
t)

Preview

A Product of Yii Software LLC

Powered by Yii Framework

Затем нажмите на кнопку “Предварительный просмотр”. Вы увидите, что `models/Country.php` перечислен в результатах создаваемых файлов классов. Вы можете нажать на имя файла класса для просмотра его содержимого.

Если вы уже создали такой же файл и хотите перезаписать его, нажмите кнопку `diff` рядом с именем файла, чтобы увидеть различия между генерируемым кодом и существующей версией.

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name
country

Model Class
Country

Namespace
app\models

Base Class
yii\db\ActiveRecord

Database Connection ID
db

☐ Use Table Prefix

☒ Generate Relations

☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template
default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

[Preview](#) [Generate](#)

Click on the above **Generate** button to generate the files selected below:

☒ Create ☒ Unchanged ☒ Overwrite

Code File	Action
models/Country.php	overwrite

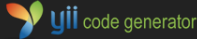
Для перезаписи существующего файла установите флажок рядом с “overwrite” и нажмите кнопку “Generate”. Для создания нового файла вы можете просто нажать “Generate”.

После этого вы увидите страницу подтверждения, указывающую на то, что код был успешно сгенерирован. Если файл существовал до этого, вы также увидите сообщение о том, что он был перезаписан заново сгенерированным кодом.

2.6.3 Создание CRUD кода

CRUD расшифровывается как Create, Read, Update и Delete, предоставляющий четыре основные функции, выполняемые над данными на большинстве веб-сайтов. Чтобы создать функциональность CRUD используя Gii, выберите “CRUD Генератор” (нажав на ссылку на главной странице Gii). Для нашей таблицы «country» заполните полученную форму следующим образом:

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

 Home Help Application

Model Generator >

CRUD Generator >

Controller Generator >

Form Generator >

Module Generator >

Extension Generator >

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Model Class

Search Model Class

Controller Class

View Path

Base Controller Class

yii\web\Controller

Widget Used in Index Page

GridView

☐ **Enable I18N**

Code Template

default (C:\dev\yii2\extensions\gii\generators\crud\default)

Preview

A Product of Yii Software LLC

Powered by [Yii Framework](#)

Затем нажмите на кнопку “Preview”. Вы увидите список файлов, которые будут созданы, как показано ниже.

Base Controller Class

yii\web\Controller

Widget Used in Index Page

GridView

☐ Enable I18N

Code Template

default (C:\dev\yii2\extensions\gii\generators\crud\default)

Preview Generate

Click on the above **Generate** button to generate the files selected below:

☒ Create ☒ Unchanged ☒ Overwrite

Code File	Action	
controllers/CountryController.php	create	<input checked="" type="checkbox"/>
models/CountrySearch.php	create	<input checked="" type="checkbox"/>
views/country/_form.php	create	<input checked="" type="checkbox"/>
views/country/_search.php	create	<input checked="" type="checkbox"/>
views/country/create.php	create	<input checked="" type="checkbox"/>
views/country/index.php	create	<input checked="" type="checkbox"/>
views/country/update.php	create	<input checked="" type="checkbox"/>
views/country/view.php	create	<input checked="" type="checkbox"/>

Если вы уже создали файлы `controllers/CountryController.php` и `views/country/index.php` (в разделе о базах данных), установите флажок “overwrite”, чтобы заменить их. (Предыдущие версии не поддерживают CRUD полностью)

2.6.4 Испытываем в действии

Чтобы увидеть как всё это работает, перейдите по следующему URL, используя ваш браузер:

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

Вы увидите таблицу, показывающую страны из таблицы БД. Вы можете сортировать, а также фильтровать данные, указывая условия фильтрации в заголовках столбцов.

Для каждой отображающейся в таблице страны вы можете просмотреть подробную информацию, обновить или удалить её. Вы также можете нажать на кнопку “Создать страну” в верхней части таблицы для получения формы создания новой страны.

My Company































HomeAboutContactLogin

[Home](#) / [Countries](#)

Countries

Create Country

Showing 1-10 of 10 items.

#	Code	Name	Population	
	<input type="text"/>	<input type="text"/>	<input type="text"/>	
1	AU	Australia	18886000	  
2	BR	Brazil	170115000	  
3	CA	Canada	1147000	  
4	CN	China	1277558000	  
5	DE	Germany	82164700	  
6	FR	France	59225700	  
7	GB	United Kingdom	59623400	  
8	IN	India	1013662000	  
9	RU	Russia	146934000	  
10	US	United States	278357000	  

[<](#) [1](#) [>](#)

© My Company 2014

Powered by Yii Frame 

My Company

HomeAboutContactLogin

[Home](#) / [Countries](#) / [United States](#) / [Update](#)

Update Country: United States

Code

US

Name


United States

Population

278357000

Update

© My Company 2014

Powered by Yii Frame 

Ниже приведен список файлов, созданных с помощью Gii, в том случае, если вы захотите исследовать реализацию этих функций, или изменить их:

- Контроллер: `controllers/CountryController.php`

- Модели: `models/Country.php` и `models/CountrySearch.php`
- Вид: `views/country/*.php`

Информация: Gii разработан как тонконастраиваемый и расширяемый инструмент генерации кода. Используя его с умом, вы можете значительно ускорить скорость разработки приложений. Для более подробной информации, пожалуйста, обратитесь к разделу Gii.

2.6.5 Заключение

В этом разделе вы узнали, как использовать Gii для генерации кода, реализующего полную функциональность CRUD для данных, хранящихся в таблице базы данных.

2.7 Взгляд в будущее

В итоге вы создали полноценное приложение на Yii и узнали, как реализовать некоторые наиболее часто используемые функции, такие, как получение данных от пользователя при помощи HTML форм, выборки данных из базы данных и их отображения в разбитом на страницы виде. Так же вы узнали, как использовать Gii для автоматической генерации кода, что превращает программирование в настолько простую задачу, как простое заполнение какой-либо формы. В этом разделе мы обобщим ресурсы о Yii, которые помогут вам быть более продуктивным при использовании Yii.

- Документация
 - Подробное руководство: как следует из названия, руководство точно определяет, как Yii должен работать и дает вам общие указания по его использованию. Это самый важный учебник по Yii, который вы должны прочитать, прежде чем писать различный Yii код.
 - Описание классов: определяет использование каждого класса, представленного в Yii. Им следует пользоваться, когда вы пишете код и хотите разобраться в использовании конкретного класса, метода, свойства.
 - Вики статьи: написаны пользователями Yii на основе их собственного опыта. Большинство из них составлены для сборника рецептов, показывая, как решить конкретные проблемы с использованием Yii. Причём качество этих статей может быть таким же хорошим, как в Подробном руководстве. Они полезны тем, что охватывают более широкие темы и часто могут предоставить вам готовые решения для дальнейшего использования.

- Книги
- Расширения¹⁸: Yii гордится библиотекой из тысяч внесённых пользователями расширений, которые могут быть легко подключены в ваши приложения и сделать разработку приложений ещё быстрее и проще.
- Сообщество
 - Форум¹⁹
 - GitHub²⁰
 - Facebook²¹
 - Twitter²²
 - LinkedIn²³

¹⁸<http://www.yiiframework.com/extensions/>

¹⁹<http://www.yiiframework.com/forum/>

²⁰<https://github.com/yiisoft/yii2>

²¹<https://www.facebook.com/groups/yiitalk/>

²²<https://twitter.com/yiiframework>

²³<https://www.linkedin.com/groups/yii-framework-1483367>

Глава 3

Структура приложения

3.1 Обзор

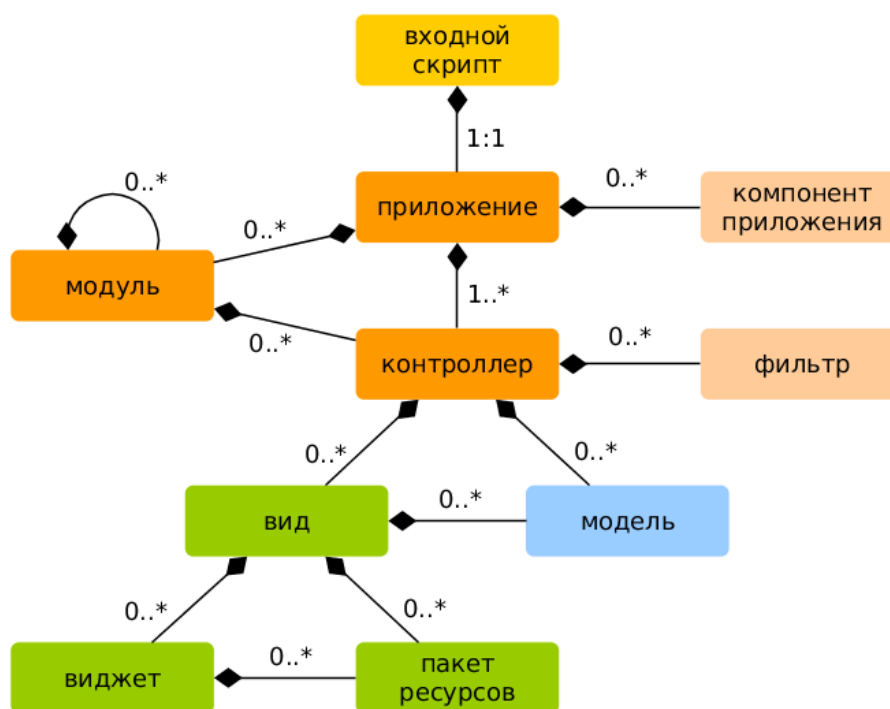
Yii приложения организованы согласно шаблону проектирования модель-представление-поведение (MVC)¹. Модели представляют собой данные, бизнес логику и бизнес правила; представления отвечают за отображение информации, в том числе и на основе данных, полученных из моделей; контроллеры принимают входные данные от пользователя и преобразовывают их в понятный для моделей формат и команды, а также отвечают за отображение нужного представления.

Кроме MVC, Yii приложения также имеют следующие сущности:

- входные скрипты: это РНР скрипты, которые доступны напрямую конечному пользователю приложения. Они ответственны за запуск и обработку входящего запроса;
- приложения: это глобально доступные объекты, которые осуществляют корректную работу различных компонентов приложения и их координацию для обработки запроса;
- компоненты приложения: это объекты, зарегистрированные в приложении и предоставляющие различные возможности для обработки текущего запроса;
- модули: это самодостаточные пакеты, которые включают в себя полностью все средства для MVC. Приложение может быть организовано с помощью нескольких модулей;
- фильтры: это код, который должен быть выполнен до и после обработки запроса контроллерами;
- виджеты: это объекты, которые могут включать в себя представления. Они могут содержать различную логику и быть использованы в различных представлениях.

Ниже на диаграмме представлена структурная схема приложения:

¹<http://ru.wikipedia.org/wiki/Model-View-Controller>



3.2 Входные скрипты

Входные скрипты это первое звено в процессе начальной загрузки приложения. Приложение (веб приложение или консольное приложение) включает единый входной скрипт. Конечные пользователи делают запросы к входному скрипту, который создает объекты приложения и перенаправляет запрос к ним.

Входные скрипты для веб приложений должны быть сохранены в папках, доступных из веб, таким образом они могут быть доступны конечным пользователям. Такие скрипты обычно именуются `index.php`, но так же могут использовать другие имена, которые могут быть распознаны используемыми веб-серверами.

Входные скрипты для консольных приложений обычно расположены в базовой папке приложений и имеют название `yii` (с суффиксом `.php`). Они должны иметь права на выполнение, таким образом пользователи смогут запускать консольные приложения через команду `./yii маршрут<> аргументы[] опции[]`.

Входные скрипты в основном делают следующую работу:

- Объявляют глобальные константы;
- Регистрируют загрузчик классов Composer²;

²<http://getcomposer.org/doc/01-basic-usage.md#autoloading>

- Подключают файл класса `Yii`;
- Загружают конфигурацию приложения;
- Создают и конфигурируют объект приложения;
- Вызывают метод `yii\base\Application::run()` приложения для обработки входящего запроса.

3.2.1 Веб приложения

Ниже представлен код входного скрипта для базового шаблона приложения.

```
<?php

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

// регистрация загрузчика классов Composer
require(__DIR__ . '/../vendor/autoload.php');

// подключение файла класса Yii
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

// загрузка конфигурации приложения
$config = require(__DIR__ . '/../config/web.php');

// создание и конфигурация приложения, а также вызов метода для обработки
// входящего запроса
(new yii\web\Application($config))->run();
```

3.2.2 Консольные приложения

Ниже представлен аналогичный код входного скрипта консольного приложения:

```
#!/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);

// fcgi не имеет констант STDIN и STDOUT, они определяются по умолчанию
defined('STDIN') or define('STDIN', fopen('php://stdin', 'r'));
defined('STDOUT') or define('STDOUT', fopen('php://stdout', 'w'));

// регистрация загрузчика классов Composer
require(__DIR__ . '/../vendor/autoload.php');
```

```
// подключение файла класса Yii
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');

// загрузка конфигурации приложения
$config = require(__DIR__ . '/config/console.php');

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

3.2.3 Объявление констант

Входные скрипты являются наилучшим местом для объявления глобальных констант. Yii поддерживают следующие три константы:

- `YII_DEBUG`: указывает работает ли приложение в отладочном режиме. Находясь в отладочном режиме, приложение будет собирать больше информации в логи и покажет детальный стек вызовов если возникнет исключение. По этой причине, отладочный режим должен быть использован только в процессе разработки. По-умолчанию значение `YII_DEBUG` равно `false`;
- `YII_ENV`: указывает в каком окружении запущено приложение. Данная тема подробно описана в разделе Конфигурации. По-умолчанию значение `YII_ENV` равно `'prod'`, означающие, что приложение запущено в производственном режиме;
- `YII_ENABLE_ERROR_HANDLER`: указывает нужно ли включать имеющийся в Yii обработчик ошибок. По-умолчанию значение данной константы равно `true`.

При определении константы, мы обычно используем следующий код:

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

который равнозначен коду, приведенному ниже:

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

Первый способ является более кратким и понятным.

Константы должны быть определены как можно раньше, в самом начале входного скрипта, таким образом они могут оказать влияние, когда остальные PHP файлы будут подключены.

3.3 Приложения

Приложения это объекты, которые управляют всей структурой и жизненным циклом прикладной системы Yii. Каждая прикладная система Yii включает в себя один объект приложения, который создается во входном скрипте и глобально доступен через `\Yii::$app`.

Информация: В зависимости от контекста, когда мы говорим “приложение”, это может означать как объект приложения так и приложение как прикладную систему в целом.

Существует два вида приложений: **веб приложения** и **консольные приложения**. Как можно догадаться по названию, первый тип в основном занимается обработкой веб запросов, в то время как последний - консольных команд.

3.3.1 Конфигурации приложения

Когда входной скрипт создаёт приложение, он загрузит конфигурацию и применит её к приложению, например:

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

// загрузка конфигурации приложения
$config = require(__DIR__ . '/../config/web.php');

// создание объекта приложения и его конфигурирование
(new yii\web\Application($config))->run();
```

Также как и обычные конфигурации, конфигурации приложения указывают как следует инициализировать свойства объектов приложения. Из-за того, что конфигурация приложения часто является очень сложной, она разбивается на несколько конфигурационных файлов, например, `web.php` - файл в приведённом выше примере.

3.3.2 Свойства приложений

Существует много важных свойств приложения, которые вы настраиваете в конфигурациях приложения. Эти свойства обычно описывают окружение, в котором работает приложение. Например, приложение должно знать каким образом загружать контроллеры, где хранить временные файлы, и т. д. Ниже мы рассмотрим данные свойства.

Обязательные свойства

В любом приложении, вы должны настроить минимум два свойства: `id` и `basePath`.

id Свойство `id` это уникальный индекс приложения, который отличает его от других приложений. В основном это используется внутрисистемно. Хотя это и не обязательно, но для лучшей совместимости рекомендуется использовать буквенно-цифровые символы при указании индекса приложения.

basePath Свойство **basePath** указывает на корневую директорию приложения. Эта директория содержит весь защищенный исходный код приложения. В данной директории обычно могут находиться поддиректории **models**, **views**, **controllers**, содержащие код, соответствующий шаблону проектирования MVC.

Вы можете задать свойство **basePath** используя путь к директории или используя псевдоним пути. В обоих случаях, указанная директория должна существовать, иначе будет выброшено исключение. Путь будет нормализован функцией **realpath()**.

Свойство **basePath** часто используется для указания других важных путей (например, путь к директории **runtime**, используемой приложением). По этой причине, псевдоним пути **@app** предустановлен и содержит данный путь. Производные пути могут быть получены с использованием этого псевдонима пути (например, **@app/runtime** указывает на временную директорию **runtime**).

Важные свойства

Свойства, указанные в этом подразделе, часто нуждаются в преднастройке т.к. они разнятся от приложения к приложению.

aliases Это свойство позволяет настроить вам множество псевдонимов в рамках массива. Ключами массива являются имена псевдонимов, а значениями массива - соответствующие значения пути. Например,

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

Это свойство доступно таким образом, чтобы вы могли указывать псевдонимы в рамках конфигурации приложения, а не вызовов метода **Yii::setAlias()**.

bootstrap Данное свойство является очень удобным, оно позволяет указать массив компонентов, которые должны быть загружены в процессе **начальной загрузки** приложения. Например, если вы хотите, чтобы модуль производил тонкую настройку URL правил, вы можете указать его ID в качестве элемента данного свойства.

Каждый из элементов данного свойства, может быть указан в одном из следующих форматов:

- ID, указанный в компонентах;
- ID модуля, указанный в модулях;
- название класса;

- массив конфигурации;
- анонимная функция, которая создаёт и возвращает компонент.

Например,

```
[
    'bootstrap' => [
        // ID компонента приложения или модуля
        'demo',

        // название класса
        'app\components\Profiler',

        // массив конфигурации
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ],

        // анонимная функция
        function () {
            return new app\components\Profiler();
        }
    ],
]
```

Info: Если ID модуля такой же, как идентификатор компонента приложения, то в процессе начальной загрузки будет использован компонент приложения. Если Вы вместо этого хотите использовать модуль, то можете указать его при помощи анонимной функции похожей на эту: `'php [`

```
function () {
    return Yii::$app->getModule('user');
},
```

`] ,`

В процессе начальной загрузки, каждый компонент будет создан. Если класс компонента имеет интерфейс `yii\base\BootstrapInterface`, то также будет вызван метод `bootstrap()`.

Еще одним практическим примером является конфигурация базового шаблона приложения, в котором модули `debug` и `gii` указаны как `bootstrap` компоненты, когда приложение находится в отладочном режиме.

```
if (YII_ENV_DEV) {
    // настройка конфигурации для окружения 'dev'
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
```

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

Примечание: Указывание слишком большого количества компонентов в `bootstrap` приведет к снижению производительности приложения, потому что для каждого запроса одно и то же количество компонентов должно быть загружено. Таким образом вы должны использовать начальную загрузку разумно.

catchAll Данное свойство поддерживается только **веб приложениями**. Оно указывает действие контроллера, которое должно обрабатывать все входящие запросы от пользователя. В основном это используется, когда приложения находится в режиме обслуживания и должно обрабатывать все запросы через одно действие.

Конфигурация это массив, первый элемент которого, определяет маршрут действия. Остальные элементы в формате пара ключ-значение задают дополнительные параметры, которые должны быть переданы действию (методу контроллера `actionXXX`). Например,

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

components Данное свойство является наиболее важным. Оно позволяет вам зарегистрировать список именованных компонентов, называемых компоненты приложения, которые Вы можете использовать в других местах. Например,

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

Каждый компонент приложения указан массивом в формате ключ-значение. Ключ представляет собой ID компонента приложения, в то время как значение представляет собой название класса или конфигурацию.

Вы можете зарегистрировать любой компонент в приложении, позже этот компонент будет глобально доступен через выражение `\Yii::$app->ComponentID`.

Более подробная информация приведена в разделе Компоненты приложения.

controllerMap Данное свойство позволяет вам задавать соответствия(mapping) между ID контроллера и произвольным классом контроллера. По-умолчанию, Yii задает соответствие между ID контроллера и его классом согласно данному соглашению (таким образом, ID `post` будет соответствовать `app\controllers\PostController`). Задавая эти свойства вы можете переопределить соответствия для необходимых контроллеров. В приведенном ниже примере, `account` будет соответствовать контроллеру `app\controllers\UserController`, в то время как `article` будет соответствовать контроллеру `app\controllers\PostController`.

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

Ключами данного свойства являются ID контроллеров, а значениями являются соответствующие названия классов(полное название класса с пространством имен) контроллера или конфигурация.

controllerNamespace Данное свойство указывает пространство имен, в котором по умолчанию должны находиться названия классов контроллеров. По-умолчанию значение равно `app\controllers`. Если ID контроллера `post`, то согласно соглашению, соответствующий класс контроллера (без пространства имен) будет равен `PostController`, а полное название класса будет равно `app\controllers\PostController`.

Класс контроллера может также находиться в поддиректории директории, соответствующей этому пространству имен. Например, ID контроллера `admin/post`, будет соответствовать полное имя класса контроллера `app\controllers\admin\PostController`.

Очень важно, чтобы полное имя класса контроллера могло быть использовано автозагрузкой и соответствующее пространство имен вашего контроллера соответствовало данному свойству. Иначе, Вы получите ошибку “Страница не найдена”, при доступе к приложению.

В случае, если вы хотите переопределить соответствия как описано выше, вы можете настроить свойство `controllerMap`.

language Данное свойство указывает язык приложения, на котором содержимое страницы должно быть отображено конечному пользователю. По-умолчанию значение данного свойства равно `en`, что означает “Английский”. Если ваше приложение должно поддерживать несколько языков, вы должны настроить данное свойство.

Значение данного свойства определяется различными аспектами интернационализации, в том числе переводом сообщений, форматированием дат, форматированием чисел, и т. д. Например, виджет `yii\yii\DatePicker` использует данное свойство для определения по умолчанию языка, на котором должен быть отображен календарь и формат данных для календаря.

Рекомендуется что вы будете указывать язык в рамках стандарта IETF³. Например, для английского языка используется `en`, в то время как для английского в США - `en-US`.

Более детальная информация приведена в разделе Интернационализация.

modules Данное свойство указывает модули, которые содержаться в приложении.

Значениями свойства могут быть массивы имен классов модулей или конфигураций, а ключами - ID модулей. Например,

```
[
    'modules' => [
        // a "booking" module specified with the module class
        'booking' => 'app\modules\booking\BookingModule',

        // a "comment" module specified with a configuration array
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

Более детальная информация приведена в разделе Модули.

name Свойство указывает название приложения, которое может быть показано конечным пользователям. В отличие от свойства `id`, которое должно быть уникальным, значение данного свойства нужно в основном для отображения и не обязательно должно быть уникальным.

Если ваш код не использует данное свойство, то вы можете не настраивать его.

³http://en.wikipedia.org/wiki/IETF_language_tag

params Данное свойство указывает массив глобально доступных параметров приложения. Вместо того, чтобы использовать жестко фиксированные числа и строки в вашем коде, лучше объявить их параметрами приложения в едином месте и использовать в нужных вам местах кода. Например, вы можете определить размер превью для изображений следующим образом:

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

Затем, когда вам нужно использовать данные значения в вашем коде, вы делаете это как представлено ниже:

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

Если позже вам понадобится изменить размер превью изображений, вам нужно только изменить это значение в настройке приложения, не касаясь зависимого кода.

sourceLanguage Данное свойство указывает язык на котором написан код приложения. По-умолчанию значение равно `'en-US'`, что означает “Английский” (США). Вы должны настроить данное свойство соответствующим образом, если содержимое в вашем коде является не английским языком.

Аналогично свойству `language`, вы должны указать данное свойство в рамках стандарта IETF⁴. Например, для английского языка используется `en`, в то время как для английского в США - `en-US`.

Более детальная информация приведена в разделе Интернационализация.

timeZone Данное свойство предоставляет альтернативный способ установки временной зоны в процессе работы приложения. Путем указания данного свойства, вы по существу вызываете РНР функцию `date_default_timezone_set()`⁵. Например,

```
[
    // Europe/Moscow для России прим(. пер.)
    'timeZone' => 'America/Los_Angeles',
]
```

⁴http://en.wikipedia.org/wiki/IETF_language_tag

⁵<http://www.php.net/manual/ru/function.date-default-timezone-set.php>

version Данное свойство указывает версию приложения. По-умолчанию значение равно `'1.0'`. Вы можете не настраивать это свойство, если ваш код не использует его.

Полезные свойства

Свойства, указанные в данном подразделе, не являются часто конфигурируемыми, т. к. их значения по умолчанию соответствуют общепринятым соглашениям. Однако, вы можете их настроить, если вам нужно использовать другие соглашения.

charset Свойство указывает кодировку, которую использует приложение. По-умолчанию значение равно `'UTF-8'`, которое должно быть оставлено как есть для большинства приложения, только если вы не работаете с устаревшим кодом, который использует большее количество данных не юникода.

defaultRoute Свойство указывает маршрут, который должно использовать приложение, когда он не указан во входящем запросе. Маршрут может состоять из ID модуля, ID контроллера и/или ID действия. Например, `help`, `post/create`, `admin/post/create`. Если действие не указано, то будет использовано значение по умолчанию указанное в `yii\base\Controller::$defaultAction`.

Для веб приложений, значение по умолчанию для данного свойства равно `'site'`, что означает контроллер `SiteController` и его действие по умолчанию должно быть использовано. Таким образом, если вы попытаетесь получить доступ к приложению не указав маршрут, оно покажет вам результат действия `app\controllers\SiteController::actionIndex()`.

Для консольных приложений, значение по умолчанию равно `'help'`, означающее, что встроенная команда `yii\console\controllers\HelpController::actionIndex()` должна быть использована. Таким образом, если вы выполните команду `yii` без аргументов, вам будет отображена справочная информация.

extensions Данное свойство указывает список расширений, которые установлены и используются приложением. По-умолчанию, значением данного свойства будет массив, полученный из файла `@vendor/yiisoft/extensions.php`. Файл `extensions.php` генерируется и обрабатывается автоматически, когда вы используете Composer⁶ для установки расширений. Таким образом, в большинстве случаев вам не нужно настраивать данное свойство.

⁶<http://getcomposer.org>

В особых случаях, когда вы хотите обрабатывать расширения в ручную, вы можете указать данное свойство следующим образом:

```
[
  'extensions' => [
    [
      'name' => 'extension name',
      'version' => 'version number',
      'bootstrap' => 'BootstrapClassName', // опционально, может
      быть также массив конфигурации
      'alias' => [ // опционально
        '@alias1' => 'to/path1',
        '@alias2' => 'to/path2',
      ],
    ],
    // ... аналогично для остальных расширений ...
  ],
]
```

Свойство является массивом спецификаций расширений. Каждое расширение указано массивом, состоящим из элементов `name` и `version`. Если расширение должно быть выполнено в процессе начальной загрузки, то следует указать `bootstrap` элемент, который может являться именем класса или конфигурацией. Расширение также может определять несколько псевдонимов.

layout Данное свойство указывает имя шаблона по умолчанию, который должен быть использован при формировании представлений. Значение по умолчанию равно `'main'`, означающее, что должен быть использован шаблон `main.php` в папке шаблонов. Если оба свойства папка шаблонов и папка представлений имеют значение по умолчанию, то файл шаблона по умолчанию может быть представлен псевдонимом пути как `@app/views/layouts/main.php`.

Для отключения использования шаблона, вы можете указать данное свойство как `false`, хотя это используется очень редко.

layoutPath Свойство указывает путь, по которому следует искать шаблоны. Значение по умолчанию равно `layouts`, означающее подпапку в папке представлений. Если значение папки представлений является значением по умолчанию, то папка шаблонов по умолчанию может быть представлена псевдонимом пути как `@app/views/layouts`.

Вы можете настроить данное свойство как папку так и как псевдоним.

runtimePath Свойство указывает путь, по которому хранятся временные файлы, такие как: лог файлы, кэш файлы. По-умолчанию значение

равно папке, которая представлена псевдонимом пути `@app/runtime`.

Вы можете настроить данное свойство как папку или как псевдоним пути. Обратите внимание, что данная папка должна быть доступна для записи, процессом, который запускает приложение. Также папка должна быть защищена от доступа конечными пользователями, хранимые в ней временные файлы могут содержать важную информацию.

Для упрощения работы с данной папкой, Yii предоставляет предопределенный псевдоним пути `@runtime`.

viewPath Данное свойство указывает базовую папку, где содержаться все файлы представлений. Значение по умолчанию представляет собой псевдоним `@app/views`. Вы можете настроить данное свойство как папку так и как псевдоним.

vendorPath Свойство указывает папку сторонних библиотек, которые используются и управляются Composer⁷. Она содержит все сторонние библиотеки используемые приложением, включая Yii фреймворк. Значение по умолчанию представляет собой псевдоним `@app/vendor`.

Вы можете настроить данное свойство как папку так и как псевдоним. При изменении данного свойства, убедитесь что вы также изменили соответствующим образом настройки Composer.

Для упрощения работы с данной папкой, Yii предоставляет предопределенный псевдоним пути `@vendor`.

enableCoreCommands Данное свойство поддерживается только **консольными приложениями**. Оно указывает нужно ли использовать встроенные в Yii консольные команды. Значение по умолчанию равно `true`.

3.3.3 События приложения

В течение жизненного цикла приложения, возникает несколько событий. Вы можете назначать обработчики событий в конфигурации приложения следующим образом:

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

Использование синтаксиса `on eventName` детально описано в разделе Конфигурации.

Также вы можете назначить обработчики событий в процессе начальной загрузки приложения, сразу после того как приложение будет создано. Например,

⁷<http://getcomposer.org>

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

EVENT_BEFORE_REQUEST

Данное событие возникает *до* того как приложение начинает обрабатывать входящий запрос. Настоящее имя события - `beforeRequest`.

На момент возникновения данного события, объект приложения уже создан и проинициализирован. Таким образом, это является хорошим местом для вставки вашего кода с помощью событий, для перехвата управления обработкой запроса. Например, обработчик события, может динамически подставлять язык приложения `yii\base\Application::$language` в зависимости от некоторых параметров.

EVENT_AFTER_REQUEST

Данное событие возникает *после* того как приложение заканчивает обработку запроса, но *до* того как произойдет отправка ответа. Настоящее имя события - `afterRequest`.

На момент возникновения данного события, обработка запроса завершена и вы можете воспользоваться этим для произведения постобработки запроса, с целью настройки ответа.

Обратите внимание, что в компоненте **response** также возникают события в процессе отправки данных конечному пользователю. Эти события возникают *после* текущего события.

EVENT_BEFORE_ACTION

Событие возникает *до* того как будет выполнено действие контроллера. Настоящее имя события - `beforeAction`.

Событие является объектом `yii\base\ActionEvent`. Обработчик события может устанавливать свойство `yii\base\ActionEvent::$isValid` равным `false` для предотвращения выполнения действия.

Например,

```
[  
    'on beforeAction' => function ($event) {  
        if некоторое(условие) {  
            $event->isValid = false;  
        } else {  
        }  
    },  
]
```

Обратите внимание что то же самое событие `beforeAction` возникает в модулях и контроллерах. Объекты приложения являются первыми, кто

возбуждает данные события, следуя за модулями (если таковые имеются) и в конце контроллерами. Если обработчик события устанавливает свойство `yii\base\ActionEvent::$isValid` равным `false`, все последующие события не возникнут.

EVENT_AFTER_ACTION

Событие возникает *после* выполнения действия контроллера. Настоящее имя события - `afterAction`.

Событие является объектом `yii\base\ActionEvent`. Через свойство `yii\base\ActionEvent::$result` обработчик события может получить доступ и изменить значение выполнения действия контроллера.

Например,

```
[
    'on afterAction' => function ($event) {
        if некоторое( условие) {
            // modify $event->result
        } else {
        }
    },
]
```

Обратите внимание, что то же самое событие `afterAction` возникает в модулях и контроллерах. Эти объекты возбуждают событие в обратном порядке, если сравнивать с `beforeAction`. Таким образом, контроллеры являются первыми, где возникает данное событие, затем в модулях (если таковые имеются), и наконец в приложениях.

3.3.4 Жизненный цикл приложения

Когда входной скрипт выполняется для обработки запроса, приложение будет развиваться согласно следующему жизненному циклу:

1. Входной скрипт загружает конфигурацию приложения в качества массива;
2. Входной скрипт создаёт новый объект приложения:
 - Вызывается метод `preInit()`, который настраивает некоторые жизненно важные свойства приложения, такие как `basePath`;
 - Регистрируется обработчик ошибок;
 - Настраиваются свойства приложения;
 - Вызывается метод `init()`, который затем вызывает метод `bootstrap()` для начальной загрузки компонентов.
3. Входной скрипт вызывает метод `yii\base\Application::run()` для запуска приложения:

- Возникает событие `EVENT_BEFORE_REQUEST`;
 - Обработка запроса: разбор информации запроса в маршрут с соответствующими параметрами; создание объектов модуля, контроллера и действия согласно указанному маршруту; запуск действия;
 - Возникает событие `EVENT_AFTER_REQUEST`;
 - Ответ отсылается конечному пользователю.
4. Входной скрипт получает значение статуса выхода от приложения и заканчивает обработку запроса.

3.4 Компоненты приложения

Приложения являются сервис локаторами. Они хранят множество так называемых *компонентов приложения*, которые предоставляют различные средства для обработки запросов. Например, компоненты `urlManager` ответственен за маршрутизацию веб запросов к нужному контроллеру; компонент `db` предоставляет средства для работы с базой данных; и т. д.

Каждый компонент приложения имеет свой уникальный ID, который позволяет идентифицировать его среди других различных компонентов в одном и том же приложении. Вы можете получить доступ к компоненту следующим образом:

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

Например, вы можете использовать `\Yii::$app->db` для получения соединения с БД, и `\Yii::$app->cache` для получения доступа к основному компоненту кэша, зарегистрированному в приложении.

Компонентами приложения могут быть любые объекты. Вы можете зарегистрировать их с помощью свойства `yii\base\Application::$components` в конфигурации приложения. Например,

```
[
    'components' => [
        // регистрация "cache" компонента с помощью имени класса
        'cache' => 'yii\caching\ApcCache',

        // регистрация "db" компонента с помощью массива конфигурации
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // регистрация "search" компонента с помощью анонимной функции
        'search' => function () {
```

```
        return new app\components\SolrService;  
    },  
],  
]
```

Информация: Хотя вы можете зарегистрировать столько компонентов в приложении сколько вам нужно, все таки стоит это делать разумно. Компоненты приложения похожи на глобальные переменные. Использование слишком большого количества компонентов приложения может потенциально сделать ваш код сложным для разработки и тестирования. В большинстве случаев вы можете просто создать локальный компонент и использовать его при необходимости.

3.4.1 Встроенные компоненты приложения

В Yii есть несколько *встроенных* компонентов приложения, с фиксированными ID и конфигурациями по умолчанию. Например, компонент **request** используется для сбора информации о запросе пользователя и разбора его в определенный маршрут; компонент **db** представляет собой соединение с базой данных, через которое вы можете выполнять запросы. Именно с помощью этих встроенных компонентов Yii приложения могут обработать запрос пользователя.

Ниже представлен список встроенных компонентов приложения. Вы можете конфигурировать их также как и другие компоненты приложения. Когда вы конфигурируете встроенный компонент приложения и не указываете класс этого компонента, то значение по умолчанию будет использовано.

- **assetManager**: используется для управления и опубликования ресурсов приложения. Более детальная информация представлена в разделе Ресурсы;
- **db**: представляет собой соединение с базой данных, через которое вы можете выполнять запросы. Обратите внимание, что когда вы конфигурируете данный компонент, вы должны указать класс компонента также как и остальные необходимые параметры, такие как `yii\db\Connection::$dsn`. Более детальная информация представлена в разделе Объекты доступа к данным (DAO);
- **errorHandler**: осуществляет обработку PHP ошибок и исключений. Более детальная информация представлена в разделе Обработка ошибок;
- **yii\base\Formatter**: форматирует данные для отображения их конечному пользователю. Например, число может быть отображено с различными разделителями, дата может быть отображена в формате `long`. Более детальная информация представлена в разделе Форматирование данных;

- **i18n**: используется для перевода сообщений и форматирования. Более детальная информация представлена в разделе Интернационализация;
- **log**: обработка и маршрутизация логов. Более детальная информация представлена в разделе Логирование;
- **mail**: предоставляет возможности для составления и рассылки писем. Более детальная информация представлена в разделе Отправка почты;
- **response**: представляет собой данные от сервера, которые будут направлены пользователю. Более детальная информация представлена в разделе Ответы;
- **request**: представляет собой запрос, полученный от конечных пользователей. Более детальная информация представлена в разделе Запросы;
- **session**: информация о сессии. Данный компонент доступен только в **веб приложениях**. Более детальная информация представлена в разделе Сессии и куки;
- **urlManager**: используется для разбора и создания URL. Более детальная информация представлена в разделе Разбор и генерация URL;
- **user**: представляет собой информацию аутентифицированного пользователя. Данный компонент доступен только в **веб приложениях**. Более детальная информация представлена в разделе Аутентификация;
- **view**: используется для отображения представлений. Более детальная информация представлена в разделе Представления.

3.5 Контроллеры

Контроллеры являются частью MVC⁸ архитектуры. Это объекты классов, унаследованных от `yii\base\Controller` и отвечающие за обработку запроса и генерирование ответа. В сущности, после обработки запроса приложениями, контроллеры проанализируют входные данные, передадут их в модели, вставят результаты модели в представления, и в конечном итоге сгенерируют исходящие ответы.

3.5.1 Действия

Контроллеры состоят из *действий*, которые являются основными блоками, к которым может обращаться конечный пользователь и запрашивать исполнение того или иного функционала. В контроллере может быть одно или несколько действий.

⁸<https://ru.wikipedia.org/wiki/Model-View-Controller>

Следующий пример показывает `post` контроллер с двумя действиями: `view` и `create`:

```
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()
    {
        $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,
            ]);
        }
    }
}
```

В действии `view` (определенном методом `actionView()`), код сначала загружает модель согласно запрошенному ID модели; Если модель успешно загружена, то код отобразит ее с помощью представления под названием `view`. В противном случае будет брошено исключение.

В действии `create` (определенном методом `actionCreate()`), код аналогичен. Он сначала пытается загрузить модель с помощью данных из запроса и сохранить модель. Если все прошло успешно, то код перенаправляет браузер на действие `view` с ID только что созданной модели. В противном случае он отобразит представление `create`, через которое пользователь может заполнить нужные данные.

3.5.2 Маршруты

Конечные пользователи обращаются к действиям через так называемые *маршруты*. Маршрут это строка, состоящая из следующих частей:

- ID модуля: он существует, только если контроллер принадлежит не приложению, а модулю;
- ID контроллера: строка, которая уникально идентифицирует контроллер среди всех других контроллеров одного и того же приложения (или одного и того же модуля, если контроллер принадлежит модулю);
- ID действия: строка, которая уникально идентифицирует действие среди всех других действия одного и того же контроллера.

Маршруты могут иметь следующий формат:

```
ControllerID/ActionID
```

или следующий формат, если контроллер принадлежит модулю:

```
ModuleID/ControllerID/ActionID
```

Таким образом, если пользователь запрашивает URL `http://hostname/index.php?r=site/index`, то `index` действие в `site` контроллере будет вызвано. Секция Маршрутизация содержит более подробную информацию о том как маршруты сопоставляются с действиями.

3.5.3 Создание контроллеров

В Веб приложениях, контроллеры должны быть унаследованы от `yii\web\Controller` или его потомков. Аналогично для консольных приложений, контроллеры должны быть унаследованы от `yii\console\Controller` или его потомков. Следующий код определяет `site` контроллер:

```
namespace app\controllers;  
  
use yii\web\Controller;  
  
class SiteController extends Controller  
{  
}
```

ID контроллеров

Обычно контроллер сделан таким образом, что он должен обрабатывать запросы, связанные с определенным ресурсом. Именно по этим причинам, ID контроллеров обычно являются существительные, ссылающиеся на ресурс, который они обрабатывают. Например, вы можете использовать `article` в качестве ID контроллера, которые отвечает за обработку данных статей.

По-умолчанию, ID контроллеров должны содержать только следующие символы: Английские буквы в нижнем регистре, цифры, подчеркивания, тире и слэш. Например, оба `article` и `post-comment` являются допустимыми ID контроллеров, в то время как `article?`, `PostComment`, `admin\post` не являются таковыми.

ID контроллеров также могут содержать префикс подпапки. Например, в `admin/article` часть `article` является контроллером в подпапке `admin` в **пространстве имен контроллеров**. Допустимыми символами для префиксов подпапок являются: Английские буквы в нижнем и верхнем регистре, символы подчеркивания и слэш, где слэш используется в качестве разграничителя для многовложенных подпапок (например `panels/admin`).

Правила наименования классов контроллеров

Названия классов контроллеров могут быть получены из ID контроллеров следующими способами:

- Привести в верхний регистр первый символ в каждом слове, разделенном дефисами. Обратите внимание что, если ID контроллера содержит слэш, то данное правило распространяется только на часть после последнего слэша в ID контроллера;
- Убрать дефисы и заменить любой прямой слэш на обратный;
- Добавить суффикс `Controller`;
- Добавить в начало **пространство имен контроллеров**.

Ниже приведены несколько примеров, с учетом того, что **пространство имен контроллеров** имеет значение по умолчанию равное `app\controllers`:

- `article` соответствует `app\controllers\ArticleController`;
- `post-comment` соответствует `app\controllers\PostCommentController`;
- `admin/post-comment` соответствует `app\controllers\admin\PostCommentController`;
- `adminPanels/post-comment` соответствует `app\controllers\adminPanels\PostCommentController`.

Классы контроллеров должны быть автозагружаемыми. Именно по этой причине, в вышеприведенном примере, контроллер `article` должен быть сохранен в файл, псевдоним которого `@app/controllers/ArticleController.php`; в то время как контроллер `admin/post2-comment` должен находиться в файле `@app/controllers/admin/Post2CommentController.php`.

Информация: Последний пример `admin/post2-comment` показывает каким образом вы можете расположить контроллер в подпапке **пространства имен контроллеров**. Это очень удобно, когда вы хотите организовать свои контроллеры в несколько категорий и не хотите использовать модули.

Карта контроллеров

Вы можете сконфигурировать **карту контроллеров** для того, чтобы преодолеть описанные выше ограничения именования ID контроллеров и названий классов. В основном это очень удобно, когда вы используете сторонние контроллеры, именование которых вы не можете контролировать.

Вы можете сконфигурировать **карту контроллеров** в настройках приложения следующим образом:

```
[
    'controllerMap' => [
        [
            // объявляет "account" контроллер, используя название класса
            'account' => 'app\controllers\UserController',

            // объявляет "article" контроллер, используя массив конфигурации
            'article' => [
                'class' => 'app\controllers\PostController',
                'enableCsrfValidation' => false,
            ],
        ],
    ],
]
```

Контроллер по умолчанию

Каждое приложение имеет контроллер по умолчанию, указанный через свойство `yii\base\Application::$defaultRoute`. Когда в запросе не указан маршрут, тогда будет использован маршрут указанный в данном свойстве. Для Веб приложений, это значение `'site'`, в то время как для консольных приложений, это `'help'`. Таким образом, если задан URL `http://hostname/index.php`, это означает, что контроллер `site` выполнит обработку запроса.

Вы можете изменить контроллер по умолчанию следующим образом в настройках приложения:

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

3.5.4 Создание действий

Создание действий не представляет сложностей также как и объявление так называемых *методов действий* в классе контроллера. Метод действия это *public* метод, имя которого начинается со слова **action**. Возвращаемое значение метода действия представляет собой ответные данные, которые будут высланы конечному пользователю. Приведенный ниже код определяет два действия `index` и `hello-world`:

```
namespace app\controllers;

use yii\web\Controller;

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

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

ID действий

В основном действие разрабатывается для какой-либо конкретной обработки ресурса. По этой причине, ID действий в основном являются глаголами, такими как `view`, `update`, и т. д.

По-умолчанию, ID действия должен содержать только следующие символы: Английские буквы в нижнем регистре, цифры, подчеркивания и дефисы. Дефисы в ID действий используются для разделения слов. Например, `view`, `update2`, `comment-post` являются допустимыми ID действий, в то время как `view?`, `Update` не являются таковыми.

Вы можете создавать действия двумя способами: встроенные действия и отдельные действия. Встроенное действие является методом, определенным в классе контроллера, в то время как отдельное действие является экземпляром класса, унаследованного от `yii\base\Action` или его потомков. Встроенные действия требуют меньше усилий для создания и в основном используются если у вас нет надобности в повторном использовании действий. Отдельные действия, с другой стороны, в основном создаются для использования в различных контроллерах или при использовании в расширениях.

Встроенные действия

Встроенные действия это те действия, которые определены в рамках методов контроллера, как мы это уже обсудили.

Названия методов действий могут быть получены из ID действий следующим образом:

- Привести первый символ каждого слова в ID действия в верхний регистр;
- Убрать дефисы;
- Добавить префикс `action`.

Например, `index` соответствует `actionIndex`, а `hello-world` соответствует `actionHelloWorld`.

Примечание: Названия имен действий являются *регистро-зависимыми*. Если у вас есть метод `ActionIndex`, он не будет учтен как метод действия, таким образом, запрос к действию `index` приведет к выбросу исключению. Также следует учесть, что методы действий должны иметь область видимости `public`. Методы имеющие область видимости `private` или `protected` НЕ определяют методы встроенных действий.

Встроенные действия в основном используются, потому что для их создания не нужно много усилий. Тем не менее, если вы планируете повторно использовать некоторые действия в различных местах, или если вы хотите перераспределить действия, вы должны определить его как *отдельной действие*.

Отдельные действия

Отдельные действия определяются в качестве классов, унаследованных от `yii\base\Action` или его потомков. Например, в Yii релизах, присутствуют `yii\web\ViewAction` и `yii\web>ErrorAction`, оба из которых являются отдельными действиями.

Для использования отдельного действия, вы должны указать его в *карте действий*, с помощью переопределения метода `yii\base\Controller::actions()` в вашем классе контроллера, следующим образом:

```
public function actions()
{
    return [
        // объявляет "error" действие с помощью названия класса
        'error' => 'yii\web>ErrorAction',

        // объявляет "view" действие с помощью конфигурационного массива
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

Как вы можете видеть, метод `actions()` должен вернуть массив, ключами которого являются ID действий, а значениями - соответствующие названия класса действия или конфигурация. В отличие от встроенных действий, ID отдельных действий могут содержать произвольные символы, до тех пор пока они определены в методе `actions()`.

Для создания отдельного действия, вы должны наследоваться от класса `yii\base\Action` или его потомков, и реализовать метод `run()`

с областью видимости `public`. Роль метода `run()` аналогична другим методам действий. Например,

```
<?php
namespace app\components;

use yii\base\Action;

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

Результаты действий

Возвращаемое значение методов действий или метода `run()` отдельного действия очень важно. Оно является результатом выполнения соответствующего действия.

Возвращаемое значение может быть объектом `response`, который будет отослан конечному пользователю в качестве ответа.

- Для **Веб приложений**, возвращаемое значение также может быть произвольными данными, которые будут присвоены `yii\web\Response::$data`, а затем сконвертированы в строку, представляющую тело ответа.
- Для **Консольных приложений**, возвращаемое значение также может быть числом, представляющим **статус выхода** исполнения команды.

В вышеприведенных примерах, все результаты действий являются строками, которые будут использованы в качестве тела ответа, высланного пользователю. Следующий пример, показывает действие может перенаправить браузер пользователя на новый URL, с помощью возврата `response` объекта (т. к. `redirect()` метод возвращает `response` объект):

```
public function actionForward()
{
    // перенаправляем браузер пользователя на http://example.com
    return $this->redirect('http://example.com');
}
```

Параметры действий

Методы действий для встроенных действий и методы `run()` для отдельных действий могут принимать параметры, называемые *параметры действий*. Их значения берутся из запросов. Для **Веб приложений**, значение каждого из параметров действия берется из `$_GET`, используя название

параметра в качестве ключа; для консольных приложений, они соответствуют аргументам командной строки.

В приведенном ниже примере, действие `view` (встроенное действие) определяет два параметра: `$id` и `$version`.

```
namespace app\controllers;

use yii\web\Controller;

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

Для разных запросов параметры действий будут определены следующим образом:

- `http://hostname/index.php?r=post/view&id=123`: параметр `$id` будет присвоено значение `'123'`, в то время как `$version` будет иметь значение `null`, т. к. строка запроса не содержит параметра `version`;
- `http://hostname/index.php?r=post/view&id=123&version=2`: параметрам `$id` и `$version` будут присвоены значения `'123'` и `'2'` соответственно;
- `http://hostname/index.php?r=post/view`: будет брошено исключение `yii\web\BadRequestHttpException`, т. к. обязательный параметр `$id` не был указан в запросе;
- `http://hostname/index.php?r=post/view&id[]=123`: будет брошено исключение `yii\web\BadRequestHttpException`, т. к. параметр `$id` получил неверное значение `['123']`.

Если вы хотите, чтобы параметр действия принимал массив значений, вы должны использовать type-hint значение `array`, как показано ниже:

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

Теперь, если запрос будет содержать URL `http://hostname/index.php?r=post/view&id[]=123`, то параметр `$id` примет значение `['123']`. Если запрос будет содержать URL `http://hostname/index.php?r=post/view&id=123`, то параметр `$id` все равно будет содержать массив, т. к. скалярное значение `'123'` будет автоматически сконвертировано в массив.

Вышеприведенные примеры в основном показывают как параметры действий работают для Веб приложений. Больше информации о параметрах консольных приложений представлено в секции Консольные команды.

Действие по умолчанию

Каждый контроллер имеет действие, указанное через свойство `yii\base\Controller::$defaultAction`. Когда маршрут содержит только ID контроллера, то подразумевается, что действие контроллера по умолчанию было запрошено.

По-умолчанию, это действие имеет значение `index`. Если вы хотите изменить это значение, просто переопределите данное свойство в классе контроллера следующим образом:

```
namespace app\controllers;

use yii\web\Controller;

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

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

3.5.5 Жизненный цикл контроллера

При обработке запроса, приложение создаст контроллер, основываясь на запрошенном маршруте. Для выполнения запроса, контроллер пройдет через следующие этапы жизненного цикла:

1. Метод `yii\base\Controller::init()` будет вызван после того как контроллер будет создан и сконфигурирован;
2. Контроллер создает объект действия, основываясь на запрошенном ID действия:
 - Если ID действия не указан, то будет использовано ID действия по умолчанию;
 - Если ID действия найдено в карте действий, то отдельное действие будет создано;
 - Если ID действия соответствует методу действия, то встроенное действие будет создано;
 - В противном случае, будет выброшено исключение `yii\base\InvalidRouteException`.
3. Контроллер последовательно вызывает метод `beforeAction()` приложения, модуля (если контроллер принадлежит модулю) и самого контроллера.

- Если один из методов вернул `false`, то остальные, невызванные методы `beforeAction` будут пропущены, а выполнение действия будет отменено;
 - По-умолчанию, каждый вызов метода `beforeAction()` вызовет событие `beforeAction`, на которое вы можете назначить обработчики.
4. Контроллер запускает действие:
- Параметры действия будут проанализированы и заполнены из данных запроса.
5. Контроллер последовательно вызывает методы `afterAction` контроллера, модуля (если контроллер принадлежит модулю) и приложения.
- По-умолчанию, каждый вызов метода `afterAction()` вызовет событие `afterAction`, на которое вы можете назначить обработчики.
6. Приложение, получив результат выполнения действия, присвоит его объекту `response`.

3.5.6 Лучшие практики

В хорошо-организованных приложениях, контроллеры обычно очень тонкие, и содержат лишь несколько строк кода. Если ваш контроллер слишком сложный, это обычно означает, что вам надо провести рефакторинг его и перенести какой-либо код в другие места.

В целом, контроллеры

- могут иметь доступ к данным [запроса|runtime-requests.md];
- могут вызывать методы моделей и других компонентов системы с данными запроса;
- могут использовать представления для формирования ответа;
- не должны заниматься обработкой данных, это должно происходить в слое моделей;
- должны избегать использования HTML или другой разметки, лучше это делать в представлениях.

3.6 Виды

Виды - это часть MVC⁹ архитектуры, это код, который отвечает за представление данных конечным пользователям. В веб приложениях виды создаются обычно в виде *видов - шаблонов*, которые суть PHP скрипты,

⁹<https://ru.wikipedia.org/wiki/Model-View-Controller>

в основном содержащие HTML код и код PHP, отвечающий за представление и внешний вид. Виды управляются компонентом приложения `view`, который содержит часто используемые методы для упорядочивания видов и их рендеринга. Для упрощения, мы будем называть виды - шаблоны просто видами.

3.6.1 Создание видов

Как мы упоминали ранее, вид - это просто PHP скрипт, состоящий из PHP и HTML кода. В примере ниже - вид, который представляет форму авторизации. Как видите, PHP код здесь генерирует динамический контент, как, например, заголовок страницы и саму форму, тогда как HTML организует полученные данные в готовую html страницу.

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

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

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

<pПожалуйста>, заполните следующие поля для входа на сайт:</p>

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

Внутри вида, вы можете использовать `$this`, которое представляет собой компонент вид, управляющий этим шаблоном и обеспечивающий его рендеринг.

Кроме `$this`, в виде могут быть доступны другие переменные, такие как `$form` и `$model` из примера выше. Эти переменные представляют собой данные, которые передаются в вид контроллерами или другими объектами, которые вызывают рендеринг вида.

Совет: Переданные переменные могут быть перечислены в блоке комментария в начале скрипта, чтобы их смогли распознать IDE. К тому же, это хороший способ добавления документации в вид.

Безопасность

При создании видов, которые генерируют HTML страницы, важно кодировать и/или фильтровать данные, которые приходят от пользователей перед тем как их показывать. В противном случае ваше приложение может стать жертвой атаки типа межсайтовый скриптинг¹⁰

Чтобы показать обычный текст, сначала кодируйте его с помощью `yii\helpers\Html::encode()`. В примере ниже имя пользователя кодируется перед выводом:

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

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

Чтобы показать HTML содержимое, используйте `yii\helpers\HtmlPurifier` для того, чтобы отфильтровать потенциально опасное содержимое. В примере ниже содержимое поста фильтруется перед показом:

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

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

Совет: Несмотря на то, что `HtmlPurifier` отлично справляется с тем, чтобы сделать вывод безопасным, работает он довольно медленно. Если от приложения требуется высокая производительность, рассмотрите возможность кэширования отфильтрованного результата

Организация видов

Как и для контроллеров, и моделей, для видов тоже есть определенные соглашения в их организации.

- Виды, которые рендерятся из контроллера, по умолчанию должны располагаться в папке `@app/views/ControllerID`, где `ControllerID` это ID контроллера. Например, если класс контроллера - `PostController`, то папка будет `@app/views/post`; если контроллер - `PostCommentController`, то папка будет `@app/views/post-comment`. В случае, если контроллер

¹⁰https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D1%81%D0%B0%D0%B9%D1%82%D0%BE%D0%B2%D1%8B%D0%B9_%D1%81%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%B8%D0%BD%D0%B3

принадлежит модулю, папка будет `views/ControllerID` в подпапке модуля.

- Виды, которые рендерятся из виджетов, должны располагаться в `ПутьВиджета/views`, где `ПутьВиджета` - это папка, которая содержит класс виджета.
- С видами, которые рендерятся из других объектов рекомендуется поступать по той же схеме, что и с видами виджетов.

В контроллерах и виджетах вы можете изменить папки видов по умолчанию, переопределив метод `yii\base\ViewContextInterface::getViewPath()`.

3.6.2 Рендеринг видов

Вы можете рендерить виды в контроллерах, widgets, или из любого другого места, вызывая методы рендеринга видов. Методы вызываются приблизительно так, как это показано в примере ниже,

```
/**
 * @param string $view название вида или путь файла, в зависимости от того,
 *    какой метод рендеринга используется
 * @param array $params данные, которые передаются виду
 * @return string результат рендеринга
 */
methodName($view, $params = [])
```

Рендеринг в контроллерах

Внутри контроллеров можно вызывать следующие методы рендеринга видов:

- **render()**: рендерит именованный вид и применяет шаблон к результату рендеринга.
- **renderPartial()**: рендерит именованный вид без шаблона.
- **renderAjax()**: рендерит именованный вид без шаблона, и добавляет все зарегистрированные JS/CSS скрипты и стили. Обычно этот метод применяется для рендеринга результата AJAX запроса.
- **renderFile()**: рендерит вид, заданный как путь к файлу или алиас.

Например,

```
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 NotFoundException;
}

// рендерит вид с названием 'view' и применяет к нему шаблон
return $this->render('view', [
    'model' => $model,
]);
}
```

Рендеринг в виджетах

Внутри виджетов, вы можете вызывать следующие методы для рендеринга видов.

- **render()**: рендерит именованный вид.
- **renderFile()**: рендерит вид, заданный как путь файла или алиас.

Например,

```
namespace app\components;

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

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

    public function run()
    {
        // рендерит вид с названием 'list'
        return $this->render('list', [
            'items' => $this->items,
        ]);
    }
}
```

Рендеринг в видах

Вы можете рендерить вид внутри другого вида используя методы, которые предоставляет **компонент вида**:

- **render()**: рендерит именованный вид.
- **renderAjax()**: рендерит именованный вид и добавляет зарегистрированные JS/CSS скрипты и стили. Обычно используется для рендеринга результата AJAX запроса.
- **renderFile()**: рендерит вид, заданный как путь к файлу или алиас.

Например, следующий код рендерит `_overview.php` файл вида, который находится в той же папке что и вид, который рендерится в текущий момент. Помните, что `$this` в виде - это **компонент вида** (а не контроллер, как это было в Yii1):

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

Рендеринг в других местах

Вы можете получить доступ к **виду** как компоненту приложения вот так: `Yii::$app->view`, а затем вызвать вышеупомянутые методы, чтобы отрендерить вид. Например,

```
// показывает файл "@app/views/site/license.php"  
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

Именованные виды

При рендеринге вида, вы можете указать нужный вид, используя как имя вида, так и путь к файлу/алиас. В большинстве случаев вы будете использовать первый вариант, т.к. он более нагляден и гибок. Мы называем виды, которые были вызваны с помощью сокращенного имени *именованные виды*.

Имя вида преобразуется в соответствующий ему путь файла в соответствии со следующими правилами:

- Имя вида можно указывать без расширения. В таком случае в качестве расширения будет использоваться `.php`. К примеру, имя вида `about` соответствует файлу `about.php`.
- Если имя вида начинается с двойного слеша `//`, соответствующий ему путь будет `@app/views/ViewName`. Т.е. вид будет искаться в **папке видов приложения по умолчанию**. Например, `//site/about` будет преобразован в `@app/views/site/about.php`.
- Если имя вида начинается с одинарного слеша `/`, то вид будет искаться в **папке видов по умолчанию** текущего модуля. Если активного модуля на данный момент нет, будет использована папка видов приложения по умолчанию, т.е. вид будет искаться в `@app/views`, как в одном из примеров выше.
- Если вид рендерится с помощью **контекста** и контекст реализует интерфейс `yii\base\ViewContextInterface`, путь к виду образуется путем присоединения **пути видов** контекста к имени вида. В основном это применимо к видам, которые рендерятся из контроллеров и виджетов. Например, `about` будет преобразован в `@app/views/site/about.php` если контекстом является контроллер `SiteController`.
- Если вид рендерится из другого вида, папка, в которой находится текущий вид будет добавлена к пути вложенного вида. Например,

`item` будет преобразован в `@app/views/post/item` если он рендерится из вида `@app/views/post/index.php`.

В соответствии с вышесказанным, вызов `$this->render('view')` в контроллере `app\controllers\PostController` будет рендерить файл `@app/views/post/view.php`, а вызов `$this->render('_overview')` в этом виде будет рендерить файл `@app/views/post/_overview.php`.

Доступ к данным из видов

Данные можно передавать в вид явно или подгружать их динамически, обращаясь к контексту из вида.

Передавая данные через второй параметр методов рендеринга вида, вы явно передаете данные в вид. Данные должны быть представлены как обычный массив: ключ-значение. При рендеринге вида, `php` вызывает встроенную функцию PHP `extract()` на переданном массиве, чтобы переменные из массива “распаковались” в переменные вида. Например, следующий код в контроллере передаст две переменные виду `report` : `$foo = 1` и `$bar = 2`.

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

Другой подход, подход контекстного доступа, извлекает данные из **компонента вида** или других объектов, доступных в виде (например через глобальный контейнер `Yii::$app`). Внутри вида вы можете вызывать объект контроллера таким образом: `$this->context` (см пример снизу), и, таким образом, получить доступ к его свойствам и методам, например, как указано в примере, вы можете получить ID контроллера:

```
ID контроллера: <?= $this->context->id ?>
```

Явная передача данных в вид обычно более предпочтительна, т.к. она делает виды независимыми от контекста. Однако, у нее есть недостаток - необходимость каждый раз вручную строить массив данных, что может быть довольно утомительно и привести к ошибкам, если вид рендерится в разных местах.

Передача данных между видами

Компонент вида имеет свойство `params`, которое вы можете использовать для обмена данными между видами.

Например, в виде `about` вы можете указать текущий сегмент хлебных крошек с помощью следующего кода.

```
$this->params['breadcrumbs'][] = '0 нас';
```

Затем, в шаблоне, который также является видом, вы можете отобразить хлебные крошки используя данные, переданные через `params`.

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

3.6.3 Шаблоны

Шаблоны - особый тип видов, которые представляют собой общие части разных видов. Например, у большинства страниц веб приложений одинаковые верх и низ (хедер и футер). Можно, конечно, указать их и в каждом виде, однако лучше сделать это один раз, в шаблоне, и затем, при рендеринге, включать уже отрендеренный вид в заданное место шаблона.

Создание шаблонов

Поскольку шаблоны это виды, их можно создавать точно так же, как и обычные виды. По умолчанию шаблоны хранятся в папке `@app/views/layouts`. Шаблоны, которые используются в конкретном модуле, хранятся в подпапке `views/layouts` папки модуля. Вы можете изменить папку шаблонов по умолчанию, используя свойство `yii\base\Module::$layoutPath` приложения или модулей.

Пример ниже показывает как выглядит шаблон. Для лучшего понимания мы сильно упростили код шаблона. На практике, однако, в нем часто содержится больше кода, например, тэги `<head>`, главное меню и т.д.

```
<?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() ?>
    <headerМоя компания</header>
    <?= $content ?>
    <footerМоя компания &copy; 2014</footer>
```

```
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

Как видите, шаблон генерирует HTML тэги, которые присутствуют на всех страницах. Внутри секции `<body>`, шаблон выводит переменную `$content`, которая содержит результат рендеринга видов контента, который передается в шаблон, при работе метода `yii\base\Controller::render()`.

Большинство шаблонов вызывают методы, аналогично тому, как это сделано в примере выше, чтобы скрипты и тэги, зарегистрированные в других местах приложения могли быть правильно отображены в местах вызова (например, в шаблоне).

- **beginPage()**: Этот метод нужно вызывать в самом начале шаблона. Он вызывает событие `EVENT_BEGIN_PAGE`, которое происходит при начале обработки страницы.
- **endPage()**: Этот метод нужно вызывать в конце страницы. Он вызывает событие `EVENT_END_PAGE`. Оно указывает на обработку конца страницы.
- **head()**: Этот метод нужно вызывать в `<head>` секции страницы html. Он генерирует метку, которая будет заменена зарегистрированным ранее кодом HTML (тэги [link](#), мета тэги), когда рендеринг страницы будет завершен.
- **beginBody()**: Этот метод нужно вызывать в начале секции `<body>`. Он вызывает событие `EVENT_BEGIN_BODY` и генерирует метку, которая будет заменена зарегистрированным HTML кодом (например, Javascript'ом), который нужно разместить в начале `<body>` страницы.
- **endBody()**: Этот метод нужно вызывать в конце секции `<body>`. Он вызывает событие `EVENT_END_BODY` и генерирует метку, которая будет заменена зарегистрированным HTML кодом (например, Javascript'ом), который нужно разместить в конце `<body>` страницы.

Доступ к данным в шаблонах

Внутри шаблона, у вас есть доступ к двум предопределенным переменным: `$this` и `$content`. Первая представляет собой **вид** компонент, как и в обычных видах, тогда как последняя содержит результат рендеринга вида, который рендерится при вызове метода **render()** в контроллерах.

Если вы хотите получить доступ к другим данным из шаблона, используйте метод явной передачи (он описан в секции **Доступ к данным в видах** настоящего документа). Если вы хотите передать данные из вида шаблону, вы можете использовать метод, описанный в передаче данных между видами.

Использование шаблонов

Как было описано в секции Рендеринг в контроллерах, когда вы рендерите вид, вызывая метод `render()` из контроллера, к результату рендеринга будет применен шаблон. По умолчанию будет использован шаблон `@app/views/layouts/main.php`.

Вы можете использовать разные шаблоны, конфигурируя `yii\base\Application::$layout` или `yii\base\Controller::$layout`. Первый переопределяет шаблон, который используется по умолчанию всеми контроллерами, а второй переопределяет шаблон в отдельном контроллере. Например, код внизу показывает, как можно сделать так, чтобы контроллер использовал шаблон `@app/views/layouts/post.php` при рендеринге вида. Другие контроллеры, если их свойство `layout` не переопределено, все еще будут использовать `@app/views/layouts/main.php` как шаблон.

```
namespace app\controllers;

use yii\web\Controller;

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

    // ...
}
```

Для контроллеров, принадлежащих модулю, вы также можете переопределять свойство модуля `layout`, чтобы использовать особый шаблон для этих контроллеров.

Поскольку свойство `layout` может быть сконфигурировано на разных уровнях приложения (контроллеры, модули, само приложение), Yii определяет какой шаблон использовать для контроллера в два этапа.

На первом этапе определяется значение шаблона и контекстный модуль.

- Если `yii\base\Controller::$layout` свойство контроллера отличается от `null`, используется оно, и модуль контроллера как контекстный модуль.
- Если `layout` равно `null` (не задано), происходит поиск среди родительских модулей контроллера, включая само приложение (которое по умолчанию является родительским модулем для контроллеров, не принадлежащих модулям) и находится первый модуль, свойство `layout` которого не равно `null`. Тогда используется найденное значение `layout` этого модуля и сам модуль в качестве контекста. Если такой модуль не найден, значит шаблон применен не будет.

На втором этапе определяется сам файл шаблона для рендеринга на основании значения `layout` и контекстного модуля. Значением `layout` может

быть:

- Алиас пути (например, `@app/views/layouts/main`).
- Абсолютный путь (например `/main`): значение `layout` начинается со слеша. Будет искаться шаблон из папки шаблонов приложения, по умолчанию это `@app/views/layouts`.
- Относительный путь (например `main`): Будет искаться шаблон из папки шаблонов контекстного модуля, по умолчанию это `views/layouts` в папке модуля.
- Булево значение `false`: шаблон не будет применен.

Если у значения `layout` нет расширения, будет использовано расширение по умолчанию - `.php`.

Вложенные шаблоны

Иногда нужно вложить один шаблон в другой. Например, в разных разделах сайта используются разные шаблоны, но у всех этих шаблонов есть основная разметка, которая определяет HTML5 структуру страницы. Вы можете использовать вложенные шаблоны, вызывая `beginContent()` и `endContent()` в дочерних шаблонах таким образом:

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>код
... дочернего шаблона...
<?php $this->endContent(); ?>
```

В коде выше дочерний шаблон заключается в `beginContent()` и `endContent()`. Параметр, передаваемый в метод `beginContent()` определяет родительский шаблон. Это может быть как путь к файлу, так и алиас.

Используя подход выше, вы можете вкладывать шаблоны друг в друга в несколько уровней.

Использование блоков

Блоки позволяют “записывать” контент в одном месте, а показывать в другом. Они часто используются совместно с шаблонами. Например, вы определяете (записываете) блок в виде и отображаете его в шаблоне.

Для определения блока вызываются методы `beginBlock()` и `endBlock()`. После определения, блок доступен через `$view->blocks[$blockID]`, где `$blockID` - это уникальный ID, который вы присваиваете блоку в начале определения.

В примере ниже показано, как можно использовать блоки, определенные в виде, чтобы динамически изменять фрагменты шаблона.

Сначала, в виде, вы записываете один или несколько блоков:

```
...
<?php $this->beginBlock('block1'); ?>содержимое
```

```

... блока 1...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('block3'); ?>содержимое

... блока 3...

<?php $this->endBlock(); ?>

```

Затем, в шаблоне, рендерите блоки если они есть, или показываете контент по умолчанию, если блок не определен.

```

...
<?php if (isset($this->blocks['block1'])): ?>
    <?= $this->blocks['block1'] ?>
<?php else: ?>
    ... контент по умолчанию для блока 1 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>
    <?= $this->blocks['block2'] ?>
<?php else: ?>
    ... контент по умолчанию для блока 2 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block3'])): ?>
    <?= $this->blocks['block3'] ?>
<?php else: ?>
    ... контент по умолчанию для блока 3 ...
<?php endif; ?>

...

```

3.6.4 Использование компонентов вида

Компоненты вида дают много возможностей. Несмотря на то, что существует возможность создавать индивидуальные экземпляры `yii\base\View` или дочерних классов, в большинстве случаев используется сам компонент `view` приложения. Вы можете сконфигурировать компонент в конфигурации приложения таким образом:

```

[
    // ...
    'components' => [
        'view' => [
            'class' => 'app\components\View',

```

```
    ],  
    // ...  
  ],  
]
```

Компоненты вида предоставляют широкие возможности по работе с видами, они описаны в отдельных секциях документации:

- темы: позволяет менять темы оформления для сайта.
- кэширование фрагментов: позволяет кэшировать фрагменты веб-страниц.
- работа с клиентскими скриптами: Поддерживает регистрацию и рендеринг CSS и Javascript.
- управление связками: позволяет регистрацию и управление связками клиентского кода.
- альтернативные движки шаблонов: позволяет использовать другие шаблонные движки, такие как Twig¹¹, Smarty¹².

Также удобно пользоваться мелкими, но удобными фишками при разработке веб страниц, которые приведены ниже.

Установка заголовков страниц

У каждой страницы должен быть заголовок. Обычно заголовок выводится в шаблоне. Однако на практике заголовок часто определяется в видах, а не в шаблонах. Чтобы передать заголовок из вида в шаблон, используется свойство `title`.

В виде можно задать заголовок таким образом:

```
<?php  
$this->title = 'Мой заголовок страницы';  
?>
```

В шаблоне заголовок выводится следующим образом, (убедитесь, что в `<head>` у вас соответствующий код):

```
<title><?= Html::encode($this->title) ?></title>
```

Регистрация мета-тэгов

На веб страницах обычно есть мета-тэги, которые часто используются различными сервисами. Как и заголовки страниц, мета-тэги выводятся в `<head>` и обычно генерируются в шаблонах.

Если вы хотите указать, какие мета-тэги генерировать в видах, вы можете вызвать метод `yii\web\View::registerMetaTag()` в виде так, как в примере ниже:

¹¹<http://twig.sensiolabs.org/>

¹²<http://www.smarty.net/>

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework,
    php']);
?>
```

Этот код зарегистрирует мета тэг “keywords” в виде. Зарегистрированные мета тэги рендерятся после того, как закончен рендеринг шаблона. Они вставляются в то место, где в шаблоне вызван метод `yii\web\View::head()`. Результатом рендеринга мета тэгов является следующий код:

```
<meta name="keywords" content="yii, framework, php">
```

Обратите внимание, что при вызове метода `yii\web\View::registerMetaTag()` несколько раз мета тэги будут регистрироваться каждый раз без проверки на уникальность.

Чтобы убедиться, что зарегистрирован только один экземпляр одного типа мета тэгов, вы можете указать ключ мета тэга в качестве второго параметра при вызове метода. К примеру, следующий код регистрирует два мета тэга “description”, однако отрендерен будет только второй.

```
$this->registerMetaTag(['name' => 'description', 'content' => Мой сайт
    сделан с помощью Yii!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => Это сайт о
    забавных енотах.'], 'description');
```

Регистрация тэгов link

Как и мета тэги, link тэги полезны во многих случаях, как, например, задание уникальной favicon, указание на RSS фид или указание OpenID сервера для авторизации. С link тэгами можно работать аналогично работе с мета тэгами, вызывая метод `yii\web\View::registerLinkTag()`. Например, вы можете зарегистрировать link тэг в виде таким образом:

```
$this->registerLinkTag([
    'title' => 'Сводка новостей по Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'http://www.yiiframework.com/rss.xml/',
]);
```

Этот код выведет

```
<link title="Сводка новостей по Yii" rel="alternate" type="application/rss+
    xml" href="http://www.yiiframework.com/rss.xml/">
```

Как и в случае с `registerMetaTags()`, вы можете указать ключ вторым параметром при вызове `registerLinkTag()` чтобы избежать дублирования link тэгов одного типа.

3.6.5 События в видах

Компонент **вида** вызывает несколько событий во время рендеринга. Вы можете задавать обработчики для этих событий чтобы добавлять контент в вид или делать пост-обработку результатов рендеринга до того, как они будут отправлены конечным пользователям.

- **EVENT_BEFORE_RENDER**: вызывается в начале рендеринга файла в контроллере. Обработчики этого события могут придать атрибуту `yii\base\ViewEvent::$isValid` значение `false`, чтобы отменить процесс рендеринга.
- **EVENT_AFTER_RENDER**: событие инициируется после рендеринга файла вызовом `yii\base\View::afterRender()`. Обработчики события могут получать результат рендеринга через `yii\base\ViewEvent::$output` и могут изменять это свойство для изменения результата рендеринга.
- **EVENT_BEGIN_PAGE**: инициируется вызовом `yii\base\View::beginPage()` в шаблонах.
- **EVENT_END_PAGE**: инициируется вызовом `yii\base\View::endPage()` в шаблонах.
- **EVENT_BEGIN_BODY**: инициируется вызовом `yii\web\View::beginBody()` в шаблонах.
- **EVENT_END_BODY**: инициируется вызовом `yii\web\View::endBody()` в шаблонах.

Например, следующий код вставляет дату в конец `body` страницы:

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {  
    echo date('Y-m-d');  
});
```

3.6.6 Рендеринг статических страниц

Статическими страницами мы считаем страницы, которые содержат в основном статические данные и для формирования которых не нужно строить динамические данные в контроллерах.

Вы можете выводить статические страницы, сохраняя их в видах, а затем используя подобный код в контроллере:

```
public function actionAbout()  
{  
    return $this->render('about');  
}
```

Если сайт содержит много статических страниц, описанный выше подход не вполне подходит - его использование приведет к многократному повторению похожего кода. Вместо этого вы можете использовать отдельное действие `yii\web\ViewAction` в контроллере. Например,

```
namespace app\controllers;
```

```
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

Теперь, если вы создадите вид `about` в папке `@app/views/site/pages`, он будет отображаться по такому адресу:

```
http://localhost/index.php?r=site/page&view=about
```

GET параметр `view` сообщает `yii\web\ViewAction` какой вид затребован. Действие будет искать этот вид в папке `@app/views/site/pages`. Вы можете сконфигурировать параметр `yii\web\ViewAction::$viewPrefix` чтобы изменить папку в которой ищется вид.

3.6.7 Полезные советы

Виды отвечают за представление данных моделей в формате, понятным конечным пользователям. В целом, виды

- должны в основном содержать код, отвечающий за представление, такой как HTML и простой PHP для обхода, форматирования и рендеринга данных.
- не должны содержать кода, который производит запросы к БД. Такими запросами должны заниматься модели.
- должны избегать прямого обращения к данным запроса, таким как `$_GET`, `$_POST`. Разбором запроса должны заниматься контроллеры. Если данные запросов нужны для построения вида, они должны явно передаваться в вид контроллерами.
- могут читать свойства моделей, но не должны их изменять.

Чтобы сделать виды более управляемыми, избегайте создания видов, которые содержат слишком сложную логику или большое количество кода. Используйте следующие подходы для их упрощения:

- используйте шаблоны для отображения основных секций разметки сайта (верхняя часть (хедер), нижняя часть (футер) и т.п.)
- разбивайте сложный вид на несколько видов попроще. Меньшие виды можно рендерить и объединять в больший используя методы рендеринга, описанный в настоящем документе.
- создавайте и используйте виджеты как строительный материал для видов.

- создавайте и используйте классы-хелперы для изменения и форматирования данных в видах.

Error: not existing file: structure-models.md

3.7 Фильтры

Фильтры — это объекты, которые могут запускаться как перед так и после действий контроллера. Например, фильтр управления доступом может запускаться перед действиями удостовериться, что запросившему их пользователю разрешен доступ; фильтр сжатия содержимого может запускаться после действий для сжатия содержимого ответа перед отправкой его конечному пользователю.

Фильтр может состоять из *пре-фильтра* (фильтрующая логика применяется *перед* действиями) и/или *пост-фильтра* (логика, применяемая *после* действий).

3.7.1 Использование фильтров

Фильтры являются особым видом поведений. Их использование ничем не отличается от использования поведений. Вы можете объявлять фильтры в классе контроллера путём перекрытия метода `behaviors()`:

```
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');
            },
        ],
    ];
}
```

По умолчанию фильтры, объявленные в классе контроллера, будут применяться ко *всем* его действиям. Тем не менее, вы можете явно указать и конкретные действия задав свойство `only`. В примере выше фильтр `HttpCache` применяется только к действиям `index` и `view`. Вы можете настроить свойство `except` чтобы указать действия, к которым фильтр применяться не должен.

Кроме контроллеров, можно объявлять фильтры в модуле или в приложении. В этом случае они применяются ко *всем* действиям контроллеров, находящихся в этом модуле или приложении если не заданы свойства `only` и `except` как было описано выше.

Примечание: При объявлении фильтров в модулях или приложениях, следует использовать маршруты вместо идентификаторов действий в свойствах `only` и `except` так как сами по себе, идентификаторы действий не могут полностью идентифицировать действие в контексте модуля или приложения.

Когда несколько фильтров указываются для одного действия, они применяются согласно следующим правилам:

- Пре-фильтрация
 - Применяются фильтры, объявленные в приложении в том порядке, в котором они перечислены в `behaviors()`.
 - Применяются фильтры, объявленные в модуле в том порядке, в котором они перечислены в `behaviors()`.
 - Применяются фильтры, объявленные в контроллере в том порядке, в котором они перечислены в `behaviors()`.
 - Если, какой-либо из фильтров отменяет выполнение действия, оставшиеся фильтры (как пре-фильтры, так и пост-фильтры) не будут применены.
- Выполняется действие, если оно прошло пре-фильтрацию.
- Пост-фильтрация
 - Применяются фильтры объявленные в контроллере, в порядке обратном, перечисленному в `behaviors()`.
 - Применяются фильтры объявленные в модуле, в порядке обратном, перечисленному в `behaviors()`.
 - Применяются фильтры объявленные в приложении, в порядке обратном, перечисленному в `behaviors()`.

3.7.2 Создание фильтров

При создании нового фильтра действия, необходимо наследоваться от `yii\base\ActionFilter` и переопределить методы `beforeAction()` и/или `afterAction()`. Первый из них будет вызван перед выполнением действия, а второй после. Возвращаемое `beforeAction()` значение определяет, будет ли действие выполняться или нет. Если вернётся `false`, то оставшиеся фильтры не будут применены и действие выполнено не будет.

Пример ниже показывает фильтр, который выводит время выполнения действия:

```
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) - $this->_startTime;
    Yii::trace("Action '{$action->uniqueId}' spent $time second.");
    return parent::afterAction($action, $result);
}
```

3.7.3 Стандартные фильтры

Yii предоставляет набор часто используемых фильтров, которые находятся, в основном, в пространстве имен `yii\filters`. Далее вы будете кратко ознакомлены с ними.

AccessControl

Фильтр `AccessControl` обеспечивает простое управление доступом, основанное на наборе правил `rules`. В частности, перед тем как действие начинает выполнение, фильтр `AccessControl` проверяет список указанных правил, пока не найдёт соответствующее текущему контексту переменных (таких как IP адрес пользователя, статус аутентификации и так далее). Найденное правило указывает, разрешить или запретить выполнение запрошенного действия. Если ни одно из правил не подходит, то доступ будет запрещён.

В следующем примере авторизованным пользователям разрешен доступ к действиям `create` и `update`, в то время как всем другим пользователям доступ запрещён.

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['create', 'update'],
            'rules' => [
                // разрешаем аутентифицированным пользователям
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // всё остальное по умолчанию запрещено
            ],
        ],
    ];
}
```

Более подробно об управлении доступом вы можете прочитать в разделе Авторизация.

Фильтр метода аутентификации

Фильтр метода аутентификации используется для аутентификации пользователя различными способами, такими как HTTP Basic Auth¹³, OAuth 2¹⁴. Классы данных фильтров находятся в пространстве имён `yii\filters\auth`.

Следующий пример показывает, как использовать `yii\filters\auth\HttpBasicAuth` для аутентификации пользователя с помощью токена доступа, основанного на методе `basic` HTTP `auth`. Обратите внимание, что для того чтобы это работало, ваш класс `user identity class` должен реализовывать метод `findIdentityByAccessToken()`.

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::className(),
        ],
    ];
}
```

Фильтры метода аутентификации часто используются при реализации RESTful API. Более подробную информацию о технологии RESTful, смотрите в разделе Authentication.

ContentNegotiator

ContentNegotiator поддерживает согласование формата ответа и языка приложения. Он пытается определить формат ответа и/или язык, путём проверки `GET` параметров и HTTP заголовка `Accept`.

В примере ниже, ContentNegotiator сконфигурирован чтобы поддерживать форматы ответа JSON и XML, а также Английский (США) и Немецкий языки.

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

public function behaviors()
{
    return [
        [
            'class' => ContentNegotiator::className(),
        ],
    ];
}
```

¹³http://en.wikipedia.org/wiki/Basic_access_authentication

¹⁴<http://oauth.net/2/>


```
        'formats' => [
            'application/json' => Response::FORMAT_JSON,
            'application/xml' => Response::FORMAT_XML,
        ],
        'languages' => [
            'en-US',
            'de',
        ],
    ],
];
}
```

Часто требуется, чтобы форматы ответа и языки приложения были определены как можно раньше в его жизненном цикле. По этой причине, ContentNegotiator разработан так, что помимо фильтра может использоваться как компонент предварительной загрузки. Например, вы можете настроить его в конфигурации приложения:

```
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',
            ],
        ],
    ],
];
```

Информация: В случае, если предпочтительный тип содержимого и язык не могут быть определены из запроса, будут использованы первый формат и язык, описанные в **formats** и **languages**.

HttpCache

Фильтр HttpCache реализовывает кэширование на стороне клиента, используя HTTP заголовки Last-Modified и Etag:

```
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');
            },
        ],
    ];
}

```

Подробнее об использовании `HttpCache` можно прочитать в разделе HTTP Кэширование.

PageCache

Фильтр `PageCache` реализует кэширование целых страниц на стороне сервера. В следующем примере `PageCache` применяется только в действии `index` для кэширования всей страницы в течение не более чем 60 секунд или пока количество записей в таблице `post` не изменится. Он также хранит различные версии страницы в зависимости от выбранного языка приложения.

```

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,
            ]
        ],
    ];
}

```

Подробнее об использовании `PageCache` читайте в разделе Кэширование страниц.

RateLimiter

Ограничитель количества запросов в единицу времени (*RateLimiter*) реализует алгоритм ограничения запросов, основанный на алгоритме leaky

bucket¹⁵. В основном, он используется при создании RESTful API. Подробнее об использовании данного фильтра можно прочитать в разделе Ограничение запросов.

VerbFilter

Фильтр по типу запроса (*VerbFilter*) проверяет, разрешено ли запросам HTTP выполнять затребованные ими действия. Если нет, то будет выброшено исключение HTTP с кодом 405. В следующем примере в фильтре по типу запроса указан обычный набор разрешённых методов запроса при выполнении CRUD операций.

```
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

Совместное использование разными источниками CORS¹⁶ - это механизм, который позволяет использовать различные ресурсы (шрифты, скрипты, и т.д.) с отличных от основного сайта доменов. В частности, AJAX вызовы JavaScript могут использовать механизм XMLHttpRequest. В противном случае, такие “междоменные” запросы были бы запрещены из-за политики безопасности same origin. CORS задаёт способ взаимодействия сервера и браузера, определяющий возможность делать междоменные запросы.

Фильтр **Cors filter** следует определять перед фильтрами Аутентификации / Авторизации, для того чтобы быть уверенными, что заголовки CORS будут всегда посланы.

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;
```

¹⁵http://en.wikipedia.org/wiki/Leaky_bucket

¹⁶https://developer.mozilla.org/fr/docs/HTTP/Access_control_CORS

```
public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
        ],
    ], parent::behaviors());
}
```

Фильтрация Cors может быть настроена с помощью свойства `cors`.

- `cors['Origin']`: массив, используемый для определения источников. Может принимать значение `['*']` (все) или `['http://www.myserver.net', 'http://www.myotherserver.com']`. По умолчанию значение равно `['*']`.
- `cors['Access-Control-Request-Method']`: массив разрешенных типов запроса, таких как `['GET', 'OPTIONS', 'HEAD']`. Значение по умолчанию `['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']`.
- `cors['Access-Control-Request-Headers']`: массив разрешенных заголовков. Может быть `['*']` то есть все заголовки или один из указанных `['X-Request-With']`. Значение по умолчанию `['*']`.
- `cors['Access-Control-Allow-Credentials']`: определяет, может ли текущий запрос быть сделан с использованием авторизации. Может принимать значения `true`, `false` или `null` (не установлено). Значение по умолчанию `null`.
- `cors['Access-Control-Max-Age']`: определяет *срок жизни запроса, перед его началом*. По умолчанию 86400.

Например, разрешим CORS для источника `:http://www.myserver.net` с методами GET, HEAD и OPTIONS :

```
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());
}
```

Вы можете настроить заголовки CORS переопределения параметров по умолчанию *для каждого из действий*.

Например, добавление `Access-Control-Allow-Credentials` для действия `login` может быть сделано так :

```

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()
    ]);
}

```

3.8 Виджеты

Виджеты представляют собой многоразовые строительные блоки, используемые в представлениях для создания сложных и настраиваемых элементов пользовательского интерфейса в рамках объектно-ориентированного подхода. Например, виджет выбора даты (date picker) позволяет генерировать интерактивный интерфейс для выбора дат, предоставляя пользователям приложения удобный способ для ввода данных такого типа. Все, что нужно для подключения виджета - это добавить следующий код в представление:

```

<?php
use yii\bootstrap\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>

```

В комплект Yii входит большое количество виджетов, например: **active form**, **menu**, виджеты jQuery UI, виджеты Twitter Bootstrap. Далее будут представлены базовые сведения о виджетах. Для получения сведений относительно использования конкретного виджета, следует обратиться к документации соответствующего класса.

3.8.1 Использование Виджетов

Главным образом, виджеты применяют в представлениях. Для того, чтобы использовать виджет в представлении, достаточно вызвать метод `yii`

`\base\Widget::widget()`. Метод принимает массив настроек для инициализации виджета и возвращает результат его рендеринга. Например, следующий код добавляет виджет для выбора даты, сконфигурированный для использования русского в качестве языка интерфейса виджета и хранения вводимых данных в атрибуте `from_date` модели `$model`.

```
<?php
use yii\bootstrap\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ru',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]) ?>
```

Некоторые виджеты могут иметь внутреннее содержимое, которое следует располагать между вызовами методов `yii\base\Widget::begin()` и `yii\base\Widget::end()`. Например, для генерации формы входа, в следующем фрагменте кода используется виджет `yii\widgets\ActiveForm`. Этот виджет сгенерирует открывающий и закрывающий теги `<form>` в местах вызова методов `begin()` и `end()` соответственно. При этом, содержимое, расположенное между вызовами указанных методов будет выведено без каких-либо изменений.

```
<?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('Login') ?>
    </div>

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

Обратите внимание на то, что в отличие от метода `yii\base\Widget::widget()`, который возвращает результат рендеринга, метод `yii\base\Widget::begin()` возвращает экземпляр виджета, который может быть использован в дальнейшем для формирования его внутреннего содержимого.

3.8.2 Создание Виджетов

Для того, чтобы создать виджет, следует унаследовать класс `yii\base\Widget` и переопределить методы `yii\base\Widget::init()` и/или `yii\base\Widget::run()`. Как правило, метод `init()` должен содержать код, выполняющий нормализацию свойств виджета, а метод `run()` - код, возвращающий результат рендеринга виджета. Результат рендеринга может быть выведен непосредственно с помощью конструкции “echo” или же возвращен в строке методом `run()`.

В следующем примере, виджет `HelloWidget` HTML-кодирует и отображает содержимое, присвоенное свойству `message`. В случае, если указанное свойство не установлено, виджет, в качестве значения по умолчанию отобразит строку “Hello World”.

```
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);
    }
}
```

Для того, чтобы использовать этот виджет, достаточно добавить в представление следующий код:

```
<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'Good morning']) ?>
```

Ниже представлен вариант виджета `HelloWidget`, который принимает содержимое, обрамленное вызовами методов `begin()` и `end()`, HTML-кодирует его и выводит.

```
namespace app\components;

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

```
class HelloWorld extends Widget
{
    public function init()
    {
        parent::init();
        ob_start();
    }

    public function run()
    {
        $content = ob_get_clean();
        return Html::encode($content);
    }
}
```

Как Вы можете видеть, в методе `init()` происходит включение буферизации вывода PHP таким образом, что весь вывод между вызовами `init()` и `run()` может быть перехвачен, обработан и возвращен в `run()`.

Информация: При вызове метода `yii\base\Widget::begin()` будет создан новый экземпляр виджета, при этом вызов метода `init()` произойдет сразу после выполнения остального кода в конструкторе виджета. При вызове метода `yii\base\Widget::end()`, будет вызван метод `run()`, а возвращенное им значение будет выведено методом `end()`.

Следующий фрагмент кода содержит пример использования модифицированного варианта `HelloWidget`:

```
<?php
use app\components\HelloWidget;
?>
<?php HelloWorld::begin(); ?>

    content that may contain <tag>'s

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

В некоторых случаях, виджету может потребоваться вывести крупный блок содержимого. И хотя это содержимое может быть встроено непосредственно в метод `run()`, целесообразней поместить его в представление и вызвать метод `yii\base\Widget::render()` для его рендеринга. Например,

```
public function run()
{
    return $this->render('hello');
}
```

По умолчанию, файлы представлений виджетов должны находиться в директории `WidgetPath/views`, где `WidgetPath` - директория, содержащая

файл класса виджета. Таким образом, в приведенном выше примере, для виджета будет использован файл представления `@app/components/views/hello.php`, при этом файл с классом виджета расположен в `@app/components`. Для того, чтобы изменить директорию, в которой содержатся файлы-представления для виджета, следует переопределить метод `yii\base\Widget::getViewPath()`.

3.8.3 Лучшие Практики

Виджеты представляют собой объектно-ориентированный подход к повторному использованию кода пользовательского интерфейса.

При создании виджетов, следует придерживаться основных принципов концепции MVC. В общем случае, основную логику следует располагать в классе виджета, разделяя при этом код, отвечающий за разметку в представлении.

Разрабатываемые виджеты должны быть самодостаточными. Это означает, что для их использования должно быть достаточно всего лишь добавить виджет в представление. Добиться этого бывает затруднительно в том случае, когда для его функционирования требуются внешние ресурсы, такие как CSS, JavaScript, изображения и т.д. К счастью, Yii предоставляет поддержку механизма для работы с ресурсами `asset bundles`, который может быть успешно использован для решения данной проблемы.

В случае, когда виджет не содержит логики, а содержит только код, отвечающий за вывод разметки, он мало отличается от представления. В действительности, единственное его отличие состоит в том, что виджет представляет собой отдельный и удобный для распространения класс, в то время как представление - это обычный PHP скрипт, подходящий для использования только лишь в конкретном приложении.

3.9 Модули

Модули - это законченные программные блоки, состоящие из моделей, представлений, контроллеров и других вспомогательных компонентов. При установке модулей в приложение, конечный пользователь получает доступ к их контроллерам. По этой причине модули часто рассматриваются как миниатюрные приложения. В отличие от приложений, модули нельзя развертывать отдельно. Модули должны находиться внутри приложений.

3.9.1 Создание модулей

Модуль помещается в директорию, которая называется **базовым путем** модуля. Так же как и в директории приложения, в этой директории

существуют поддиректории `controllers`, `models`, `views` и другие, в которых размещаются контроллеры, модели, представления и другие элементы. В следующем примере показано примерное содержимое модуля:

<code>forum/</code>	
<code>Module.php</code>	файл класса модуля
<code>controllers/</code>	содержит файлы классов контроллеров
<code>DefaultController.php</code>	файл класса контроллера по умолчанию
<code>models/</code>	содержит файлы классов моделей
<code>views/</code>	содержит файлы представлений контроллеров
и шаблонов	
<code>layouts/</code>	содержит файлы представлений шаблонов
<code>default/</code>	содержит файлы представления контроллера
<code>DefaultController</code>	
<code>index.php</code>	файл основного представления

Классы модулей

Каждый модуль объявляется с помощью уникального класса, который наследуется от `yii\base\Module`. Этот класс должен быть помещен в корне базового пути модуля и поддерживать автозагрузку. Во время доступа к модулю будет создан один экземпляр соответствующего класса модуля. Как и экземпляры приложения, экземпляры модулей нужны, чтобы код модулей мог получить общий доступ к данным и компонентам.

Приведем пример того, как может выглядеть класс модуля:

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';
        // ... остальной инициализирующий код ...
    }
}
```

Если метод `init()` стал слишком громоздким из-за кода, который задает свойства модуля, эти свойства можно сохранить в виде конфигурации, а затем загрузить в методе `init()` следующим образом:

```
public function init()
{
    parent::init();
    // инициализация модуля с помощью конфигурации, загруженной из config.
    // php
    \Yii::configure($this, require(__DIR__ . '/config.php'));
}
```

При этом в конфигурационном файле `config.php` может быть код следующего вида, аналогичный конфигурации приложения:

```
<?php
return [
    'components' => [
        // список конфигураций компонентов
    ],
    'params' => [
        // список параметров
    ],
];
```

Контроллеры в модулях

При создании контроллеров модуля принято помещать классы контроллеров в подпространство `controllers` пространства имен класса модуля. Это также подразумевает, что файлы классов контроллеров должны располагаться в директории `controllers` базового пути модуля. Например, чтобы описать контроллер `post` в модуле `forum` из предыдущего примера, класс контроллера объявляется следующим образом:

```
namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}
```

Изменить пространство имен классов контроллеров можно задав свойство `yii\base\Module::$controllerNamespace`. Если какие-либо контроллеры выпадают из этого пространства имен, доступ к ним можно осуществить, настроив свойство `yii\base\Module::$controllerMap`, аналогично тому, как это делается в приложении.

Представления в модулях

Представления модуля также следует поместить в в поддиректорию `views` базового пути модуля. Виды, которые рендерит контроллер модуля, должны располагаться в директории `views/ControllerID`, где `ControllerID` соответствует идентификатору контроллера. Например, если контроллер реализуется классом `PostController`, представления следует разместить в поддиректории `views/post` базового пути модуля.

В модуле можно задать шаблон, который будет использоваться для рендеринга всех представлений контроллерами модуля. По умолчанию шаблон помещается в директорию `views/layouts`, а свойство `yii\base\Module::$layout` должно указывать на имя этого шаблона. Если не

задать свойство `layout`, модуль будет использовать шаблон, заданный в приложении.

3.9.2 Использование модулей

Чтобы задействовать модуль в приложении, достаточно включить его в свойство `modules` в конфигурации приложения. Следующий код в конфигурации приложения задействует модуль `forum`:

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... другие настройки модуля ...
        ],
    ],
]
```

Свойству `modules` присваивается массив, содержащий конфигурацию модуля. Каждый ключ массива представляет собой *идентификатор модуля*, который однозначно определяет модуль среди других модулей приложения, а соответствующий массив - это конфигурация для создания модуля.

Маршруты

Как маршруты приложения используются для обращения к контроллерам приложения, маршруты модуля используются, чтобы обращаться к контроллерам этого модуля. Маршрут контроллера в модуле должен начинаться с идентификатора модуля, за которым следуют идентификатор контроллера и идентификатор действия. Например, если в приложении задействован модуль `forum`, то маршрут `forum/post/index` соответствует действию `index` контроллера `post` этого модуля. Если маршрут состоит только из идентификатора модуля, то контроллер и действие определяются исходя из свойства `yii\base\Module::$defaultRoute`, которое по умолчанию равно `default`. Таким образом, маршрут `forum` соответствует контроллеру `default` модуля `forum`.

Получение доступа к модулям

Зачастую внутри модуля может потребоваться доступ к экземпляру класса модуля, через который получают идентификатор модуля, его параметры, компоненты, и т. п. Это можно сделать с помощью следующей конструкции:

```
$module = MyModuleClass::getInstance();
```

где `MyModuleClass` соответствует имени класса модуля, доступ к которому нужно получить. Метод `getInstance()` возвращает запрошенный в данный момент экземпляр класса модуля. Если модуль не запрошен, метод вернет `null`. Учтите, что обычно экземпляры класса модуля вручную не создаются, так как созданный вручную экземпляр будет отличаться от экземпляра, созданного Yii в качестве ответа на запрос.

Информация: При разработке модуля нельзя исходить из предположения, что модулю будет назначен конкретный идентификатор. Это связано с тем, что идентификатор, назначаемый модулю при использовании в приложении или в другом модуле, может быть выбран совершенно произвольно. Чтобы получить идентификатор модуля, нужно вначале выбрать экземпляр модуля, как это описано выше, а затем получить доступ к идентификатору через свойство `$module->id`.

Доступ к экземпляру модуля можно получить следующими способами:

```
// получение дочернего модуля с идентификатором "forum"
$module = \Yii::$app->getModule('forum');

// получение модуля, к которому принадлежит запрошенный в настоящее время
// контроллер
$module = \Yii::$app->controller->module;
```

Первый подход годится только если известен идентификатор модуля, а второй подход наиболее полезен, если известно, какой контроллер запрошен.

Имея экземпляр модуля можно получить доступ к параметрам и компонентам, зарегистрированным в модуле. Например,

```
$maxPostCount = $module->params['maxPostCount'];
```

Предзагрузка модулей

Может потребоваться запускать некоторые модули при каждом запросе. Модуль `debug` - один из таких модулей. Для этого список идентификаторов таких модулей необходимо указать в свойстве `bootstrap` приложения.

Например, следующая конфигурация приложения обеспечивает загрузку модуля `debug` при каждом запросе:

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

```
],
]
```

3.9.3 Вложенные модули

Модули могут вкладываться друг в друга без ограничений по глубине. Иными словами, в модуле содержится модуль, в который входит еще один модуль, и т. д. Первый модуль называется *родительским*, остальные - *дочерними*. дочерние модули объявляются в свойстве `modules` родительских модулей. Например,

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->modules = [
            'admin' => [
                // здесь имеет смысл использовать более лаконичное
                пространство имен
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

Маршрут к контроллеру вложенного модуля должен содержать идентификаторы всех его предков. Например, маршрут `forum/admin/dashboard/index` соответствует действию `index` контроллера `dashboard` модуля `admin`, который в свою очередь является дочерним модулем модуля `forum`.

Информация: Метод `getModule()` возвращает только те дочерние модули, которые принадлежат родительскому модулю непосредственно. В свойстве `yii\base\Application::$loadedModules` содержится список загруженных модулей, в том числе прямых и косвенных потомков, с индексированием по имени класса.

3.9.4 Лучшие практики

Модули лучше всего подходят для крупных приложений, функционал которых можно разделить на несколько групп, в каждой из которых функции тесно связаны между собой. Каждая группа функций может разрабатываться в виде модуля, над которым работает один разработчик или одна команда.

Модули - это хороший способ повторно использовать код на уровне групп функций. В виде модулей можно реализовать такой функционал как управление пользователями или управление комментариями, а затем использовать эти модули в будущих разработках.

3.10 Ресурсы

Ресурс в Yii это файл который может быть задан в Web странице. Это может быть CSS файл, JavaScript файл, изображение или видео файл и т.д. Ресурсы располагаются в Web доступных директориях и обслуживаются непосредственно Web серверами.

Желательно, управлять ресурсами программно. Например, при использовании виджета `yii\jui\DatePicker` в странице, автоматически включаются необходимые CSS и JavaScript файлы, вместо того чтобы просить Вас в ручную найти эти файлы и включить их. И когда Вы обновляете виджет до новой версии, будут автоматически использованы новые версии файлов-ресурсов. В этом руководстве будет описана мощная возможность управления ресурсами представленная в Yii.

3.10.1 Комплекты ресурсов

Yii управляет ресурсами как единицей *комплекта ресурсов*. Комплект ресурсов - это простой набор ресурсов расположенных в директории. Когда Вы регистрируете комплект ресурсов в представлении, в отображаемой Web странице включается набор CSS и JavaScript файлов.

3.10.2 Задание Комплекта Ресурсов

Комплект ресурсов определяется как PHP класс расширяющийся от `yii\web\AssetBundle`. Имя комплекта соответствует полному имени PHP класса (без ведущей обратной косой черты - backslash “\”). Класс комплекта ресурсов должен быть в состоянии возможности автозагрузки. При задании комплекта ресурсов обычно указывается где ресурсы находятся, какие CSS и JavaScript файлы содержит комплект, и как комплект зависит от других комплектов.

Следующий код задаёт основной комплект ресурсов используемый в шаблоне базового приложения:

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
```

```
public $basePath = '@webroot';  
public $baseUrl = '@web';  
public $css = [  
    'css/site.css',  
];  
public $js = [  
];  
public $depends = [  
    'yii\web\YiiAsset',  
    'yii\bootstrap\BootstrapAsset',  
];  
}
```

В коде выше класс `AppAsset` указывает, что файлы ресурса находятся в директории `@webroot`, которой соответствует URL `@web`; комплект содержит единственный CSS файл `css/site.css` и не содержит JavaScript файлов; комплект зависит от двух других комплектов: `yii\web\YiiAsset` и `yii\bootstrap\BootstrapAsset`. Более детальное объяснение о свойствах `yii\web\AssetBundle` может быть найдено ниже:

- **sourcePath**: задаёт корневую директорию содержащую файлы ресурса в этом комплекте. Это свойство должно быть установлено если корневая директория не доступна из Web. В противном случае, Вы должны установить **basePath** свойство и **baseUrl** свойство вместо текущего. Здесь могут быть использованы псевдонимы путей.
- **basePath**: задаёт Web доступную директорию, которая содержит файлы ресурсов текущего комплекта. Когда Вы задаёте свойство **sourcePath** Менеджер ресурсов опубликует ресурсы текущего комплекта в Web доступную директорию и перезапишет соответственно данное свойство. Вы должны задать данное свойство если Ваши файлы ресурсов уже в Web доступной директории и не нужно публиковать ресурсы. Здесь могут быть использованы псевдонимы путей.
- **baseUrl**: задаёт URL соответствующий директории **basePath**. Также как и для **basePath**, если Вы задаёте свойство **sourcePath** Менеджер ресурсов опубликует ресурсы и перезапишет это свойство соответственно. Здесь могут быть использованы псевдонимы путей.
- **js**: массив, перечисляющий JavaScript файлы, содержащиеся в данном комплекте. Заметьте, что только прямая косая черта (forward slash - “/”) может быть использована, как разделитель директорий. Каждый JavaScript файл может быть задан в одном из следующих форматов:
 - относительный путь, представленный локальным JavaScript файлом (например `js/main.js`). Актуальный путь файла может быть определён путём добавления `yii\web\AssetManager::`

`$basePath` к относительному пути, и актуальный URL файла может быть определён путём добавления `yii\web\AssetManager::basePath` к относительному пути.

- абсолютный URL, представленный внешним JavaScript файлом. Например, `http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js` или `//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js`.
- **css**: массив, перечисляющий CSS файлы, содержащиеся в данном комплекте. Формат этого массива такой же, как и у **js**.
- **depends**: массив, перечисляющий имена комплектов ресурсов, от которых зависит данный комплект.
- **jsOptions**: задаёт параметры, которые будут относиться к методу `yii\web\View::registerJsFile()`, когда он вызывается для регистрации *каждого* JavaScript файла данного комплекта.
- **cssOptions**: задаёт параметры, которые будут приняты методом `yii\web\View::registerCssFile()`, когда он вызывается для регистрации *каждого* CSS файла данного комплекта.
- **publishOptions**: задаёт параметры, которые будут приняты методом `yii\web\AssetManager::publish()`, когда метод будет вызван, опубликуются исходные файлы ресурсов в Web директории. Этот параметр используется только в том случае, если задаётся свойство `sourcePath`.

Расположение ресурсов

Ресурсы, в зависимости от их расположения, могут быть классифицированы как:

- **исходные ресурсы**: файлы ресурсов, расположенные вместе с исходным кодом PHP, которые не могут быть непосредственно доступны через Web. Для того, чтобы использовать исходные ресурсы на странице, они должны быть скопированы в Web директорию и превратиться в так называемые опубликованные ресурсы. Этот процесс называется *публикацией ресурсов*, который более подробно будет описан в ближайшее время.
- **опубликованные ресурсы**: файлы ресурсов, расположенные в Web директории и, таким образом, могут быть напрямую доступны через Web.
- **внешние ресурсы**: файлы ресурсов, расположенные на другом Web сервере, отличного от веб-хостинга вашего приложения.

При определении класса комплекта ресурсов, если Вы задаёте свойство `sourcePath`, это означает, что любые перечисленные ресурсы, используя относительные пути, будут рассматриваться как исходные ресурсы. Если Вы не задаёте данное свойство, это означает, что эти ресурсы - это опубликованные ресурсы (в этом случае Вам следует указать `basePath`

и `baseUrl`, чтобы дать знать Yii где ресурсы располагаются).

Рекомендуется размещать ресурсы, принадлежащие приложению, в Web директорию, для того, чтобы избежать не нужного процесса публикации ресурсов. Вот почему `AppAsset` в предыдущем примере задаёт `basePath` вместо `sourcePath`.

Для расширений, в связи с тем, что их ресурсы располагаются вместе с их исходным кодом в директориях, которые не являются веб-доступными, необходимо указать свойство `sourcePath` при задании класса комплекта ресурсов для них.

Примечание: Не используйте `@webroot/assets` как `source path`. Эта директория по умолчанию используется менеджером ресурсов `asset manager` для сохранения файлов ресурсов, опубликованных из их исходного месторасположения. Любое содержимое этой директории расценивается как временное и может быть удалено.

Зависимости ресурсов

Когда Вы включаете несколько CSS или JavaScript файлов в Web страницу, они должны следовать в определенном порядке, `` чтобы избежать переопределения при выдаче ``. Например, если Вы используете виджет jQuery UI в Web странице, вы должны убедиться, что jQuery JavaScript файл был включен до jQuery UI JavaScript файла. Мы называем такой порядок зависимостью между ресурсами.

Зависимости ресурсов в основном указываются через свойство `yii\web\AssetBundle::$depends`. Например в `AppAsset`, комплект ресурсов зависит от двух других комплектов ресурсов: `yii\web\YiiAsset` и `yii\bootstrap\BootstrapAsset`, что обозначает, что CSS и JavaScript файлы `AppAsset` будут включены *после* файлов этих двух комплектов зависимостей.

Зависимости ресурсов являются также зависимыми. Это значит, что если комплект А зависит от В, который зависит от С, то А тоже зависит от С.

Параметры ресурсов

Вы можете задать свойства `cssOptions` и `jsOptions`, чтобы настроить путь для включения CSS и JavaScript файлов в страницу. Значения этих свойств будут приняты методами `yii\web\View::registerCssFile()` и `yii\web\View::registerJsFile()` соответственно, когда они (методы) вызываются представлением происходит включение CSS и JavaScript файлов.

Примечание: Параметры, заданные в комплекте класса применяются для *каждого* CSS/JavaScript-файла в комплекте. Если Вы хотите использовать различные параметры для разных файлов, Вы должны создать отдельные комплекты ресурсов, и использовать одну установку параметров для каждого комплекта.

Например, условно включим CSS файл для браузера IE9 или ниже. Для этого Вы можете использовать следующий параметр:

```
public $cssOptions = ['condition' => 'lte IE9'];
```

Это вызовет CSS файл из комплекта, который будет включен в страницу, используя следующие HTML теги:

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

Для того чтобы обернуть созданную CSS ссылку в тег `<noscript>`, Вы можете настроить `cssOptions` следующим образом:

```
public $cssOptions = ['noscript' => true];
```

Для включения JavaScript файла в head раздел страницы (по умолчанию, JavaScript файлы включаются в конец раздела body) используйте следующий параметр:

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

По умолчанию, когда комплект ресурсов публикуется, всё содержимое в заданной директории `yii\web\AssetBundle::$sourcePath` будет опубликовано. Вы можете настроить это поведение, сконфигурировав свойство `publishOptions`. Например, опубликовать одну или несколько поддиректорий `yii\web\AssetBundle::$sourcePath` в классе комплекта ресурсов Вы можете в следующем образом:

```
<?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 function init()
    {
        parent::init();
        $this->publishOptions['beforeCopy'] = function ($from, $to) {
```

```
        $dirname = basename(dirname($from));  
        return $dirname === 'fonts' || $dirname === 'css';  
    };  
}  
}
```

В выше указанном примере определён комплект ресурсов для пакета “fontawesome”¹⁷. Задан параметр публикации `beforeCopy`, здесь только `fonts` и `css` поддиректории будут опубликованы.

Bower и NPM Ресурсы

Большинство JavaScript/CSS пакетов управляются Bower¹⁸ и/или NPM¹⁹. Если Вашим приложением или расширением используется такой пакет, то рекомендуется следовать следующим этапам для управления ресурсами библиотеки:

1. Исправить файл `composer.json` Вашего приложения или расширения и включить пакет в список в раздел `require`. Следует использовать `bower-asset/PackageName` (для Bower пакетов) или `npm-asset/PackageName` (для NPM пакетов) для обращения к соответствующей библиотеке.
2. Создать класс комплекта ресурсов и перечислить JavaScript/CSS файлы, которые Вы планируете использовать в Вашем приложении или расширении. Вы должны задать свойство `sourcePath` как `@bower/PackageName` или `@npm/PackageName`.

Это происходит потому, что Composer устанавливает Bower или NPM пакет в директорию, соответствующую этим псевдонимам.

Примечание: В некоторых пакетах файлы дистрибутива могут находиться в поддиректории. В этом случае, Вы должны задать поддиректорию как значение `sourcePath`. Например, `yii\web\jQueryAsset` использует `@bower/jquery/dist` вместо `@bower/jquery`.

3.10.3 Использование Комплекта Ресурсов

Для использования комплекта ресурсов, зарегистрируйте его в представлении вызвав метод `yii\web\AssetBundle::register()`. Например, комплект ресурсов в представлении может быть зарегистрирован следующим образом:

¹⁷<http://fontawesome.io/>

¹⁸<http://bower.io/>

¹⁹<https://www.npmjs.org/>

```
use app\assets\AppAsset;  
AppAsset::register($this); // $this - представляет собой объект  
представления
```

Для справки: Метод `yii\web\AssetBundle::register()` возвращает объект комплекта ресурсов, содержащий информацию о публикуемых ресурсах, таких как `basePath` или `baseUrl`.

Если Вы регистрируете комплект ресурсов в других местах (т.е. не в представлении), Вы должны обеспечить необходимый объект представления. Например, при регистрации комплекта ресурсов в классе `widget`, Вы можете взять за объект представления `$this->view`.

Когда комплект ресурсов регистрируется в представлении, Yii регистрирует все зависимые от него комплекты ресурсов. И, если комплект ресурсов расположен в директории не доступной из Web, то он будет опубликован в Web директории. Затем, когда представление отображает страницу, сгенерируются теги `<link>` и `<script>` для CSS и JavaScript файлов, перечисленных в регистрируемых комплектах. Порядок этих тегов определён зависимостью среди регистрируемых комплектов, и последовательность ресурсов перечислена в `yii\web\AssetBundle::$css` и `yii\web\AssetBundle::$js` свойствах.

Настройка Комплектов Ресурсов

Yii управляет комплектами ресурсов через компонент приложения называемый `assetManager`, который реализован в `yii\web\AssetManager`. Путём настройки свойства `yii\web\AssetManager::$bundles`, возможно настроить поведение комплекта ресурсов. Например, комплект ресурсов `yii\web\JqueryAsset` по умолчанию использует `jquery.js` файл из установленного jquery Bower пакета. Для повышения доступности и производительности, можно использовать версию jquery на Google хостинге. Это может быть достигнуто, настроив `assetManager` в конфигурации приложения следующим образом:

```
return [  
    // ...  
    'components' => [  
        'assetManager' => [  
            'bundles' => [  
                'yii\web\JqueryAsset' => [  
                    'sourcePath' => null, // не опубликовывать комплект  
                    'js' => [  
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery  
                    .min.js',  
                ],  
            ],  
        ],  
    ],  
],
```

```
    ],
];
```

Можно сконфигурировать несколько комплектов ресурсов аналогично через `yii\web\AssetManager::$bundles`. Ключи массива должны быть именами класса (без впереди стоящей обратной косой черты) комплектов ресурсов, а значения массивов должны соответствовать конфигурации массивов.

Совет: Можно условно выбрать, какой из ресурсов будет использован в комплекте ресурсов. Следующий пример показывает, как можно использовать в разработке окружения `jquery.js` или `jquery.min.js` в противном случае:

```
'yii\web\JqueryAsset' => [
    'js' => [
        YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
    ]
],
```

Можно запретить один или несколько комплектов ресурсов, связав `false` с именами комплектов ресурсов, которые Вы хотите сделать недоступными. Когда Вы регистрируете недоступный комплект ресурсов в представлении, обратите внимание, что зависимость комплектов будет зарегистрирована, и представление также не включит ни один из ресурсов комплекта в отображаемую страницу. Например, для запрета `yii\web\JqueryAsset` можно использовать следующую конфигурацию:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => false,
            ],
        ],
    ],
];
```

Можно также запретить *все* комплекты ресурсов, установив `yii\web\AssetManager::$bundles` как `false`.

Привязка ресурсов

Иногда необходимо исправить пути до файлов ресурсов, в нескольких комплектах ресурсов. Например, комплект А использует `jquery.min.js` версии 1.11.1, а комплект В использует `jquery.js` версии 2.1.1. Раньше Вы могли решить данную проблему, настраивая каждый комплект ресурсов

по отдельности, но более простой способ - использовать *asset map* возможность, чтобы найти неверные ресурсы и исправить их. Сделать это можно, сконфигурировав свойство `yii\web\AssetManager::$assetMap` следующим образом:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' => '//ajax.googleapis.com/ajax/libs/jquery
                /2.1.1/jquery.min.js',
            ],
        ],
    ],
];
```

Ключи `assetMap` - это имена ресурсов, которые Вы хотите исправить, а значения - это требуемые пути для ресурсов. Когда регистрируется комплект ресурсов в представлении, каждый соответствующий файл ресурса в `css` или `js` массивах будет рассмотрен в соответствии с этой привязкой. И, если какой-либо из ключей найден, как последняя часть пути до файла ресурса (путь на который начинается с `yii\web\AssetBundle::$sourcePath` по возможности), то соответствующее значение заменит ресурс и будет зарегистрировано в представлении. Например, путь до файла ресурса `my/path/to/jquery.js` - это соответствует ключу `jquery.js`.

Примечание: Ресурсы заданные только с использованием относительного пути могут использоваться в привязке ресурсов. Пути ресурсов должны быть абсолютные URLs или путь относительно `yii\web\AssetManager::$basePath`.

Публикация Ресурсов

Как уже было сказано выше, если комплект ресурсов располагается в директории которая не доступна из Web, эти ресурсы будут скопированы в Web директорию, когда комплект будет зарегистрирован в представлении. Этот процесс называется *публикацией ресурсов*, его автоматически выполняет `asset manager`.

По умолчанию, ресурсы публикуются в директорию `@webroot/assets` которая соответствует URL `@web/assets`. Можно настроить это местоположение сконфигурировав свойства `basePath` и `baseUrl`.

Вместо публикации ресурсов путём копирования файлов, можно рассмотреть использование символических ссылок, если Ваша операционная система или Web сервер это разрешают. Эта функция может быть включена путем установки `linkAssets` в `true`.

```
return [
```

```
// ...  
'components' => [  
    'assetManager' => [  
        'linkAssets' => true,  
    ],  
],  
];
```

С конфигурацией, установленной выше, менеджер ресурсов будет создавать символические ссылки на исходные пути комплекта ресурсов когда он будет публиковаться. Это быстрее, чем копирование файлов, а также может гарантировать, что опубликованные ресурсы всегда up-to-date(обновлённые/свежие).

Перебор Кэша

Для Web приложения запущенного в режиме продакшена, считается нормальной практикой разрешить HTTP кэширование для ресурсов и других статичных источников. Недостаток такой практики в том, что всякий раз, когда изменяется ресурс и разворачивается продакшен, пользователь может по-прежнему использовать старую версию ресурса вследствие HTTP кэширования. Чтобы избежать этого, можно использовать возможность перебора кэша, которая была добавлена в версии 2.0.3, для этого можно настроить `yii\web\AssetManager` следующим образом:

```
return [  
    // ...  
    'components' => [  
        'assetManager' => [  
            'appendTimestamp' => true,  
        ],  
    ],  
];
```

Делая таким образом, к URL каждого опубликованного ресурса будет добавляться временная метка его последней модификации. Например, URL для `yii.js` может выглядеть как `/assets/5515a87c/yii.js?v=1423448645`, где параметр `v` представляет собой временную метку последней модификации файла `yii.js`. Теперь если изменить ресурс, его URL тоже будет изменен, это означает что клиент получит последнюю версию ресурса.

3.10.4 Обычное Использование Комплекта Ресурсов

Код ядра Yii содержит большое количество комплектов ресурсов. Среди них, следующие комплекты широко используются и могут упоминаться в Вашем приложении или коде расширения:

- `yii\web\YiiAsset`: Включает основной `yii.js` файл который реализует механизм организации JavaScript кода в модулях. Также

обеспечивает специальную поддержку для `data-method` и `data-confirm` атрибутов и содержит другие полезные функции.

- `yii\web\JqueryAsset`: Включает `jquery.js` файл из jQuery Bower пакета.
- `yii\bootstrap\BootstrapAsset`: Включает CSS файл из Twitter Bootstrap фреймворка.
- `yii\bootstrap\BootstrapPluginAsset`: Включает JavaScript файл из Twitter Bootstrap фреймворка для поддержки Bootstrap JavaScript плагинов.
- `yii\jui\JuiAsset`: Включает CSS и JavaScript файлы из jQuery UI библиотеки.

Если Ваш код зависит от jQuery, jQuery UI или Bootstrap, Вам необходимо использовать эти предопределенные комплекты ресурсов, а не создавать свои собственные варианты. Если параметры по умолчанию этих комплектов не удовлетворяют Вашим нуждам, Вы можете настроить их как описано в подразделе Настройка Комплектов Ресурсов.

3.10.5 Преобразование Ресурсов

Вместо того, чтобы напрямую писать CSS и/или JavaScript код, разработчики часто пишут его в некотором `расширенном синтаксисе` и используют специальные инструменты конвертации в CSS/JavaScript. Например, для CSS кода можно использовать LESS²⁰ или SCSS²¹; а для JavaScript можно использовать TypeScript²².

Можно перечислить файлы ресурсов в `расширенном синтаксисе` в `css` и `js` свойствах из комплекта ресурсов. Например,

```
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',
    ];
}
```

Когда Вы регистрируете такой комплект ресурсов в представлении, `asset manager` автоматически запустит нужные инструменты препроцессора и

²⁰<http://lesscss.org/>

²¹<http://sass-lang.com/>

²²<http://www.typescriptlang.org/>

конвертирует ресурсы в CSS/JavaScript, если их расширенный синтаксис распознан. Когда представление окончательно отобразит страницу, в неё будут включены файлы CSS/JavaScript, вместо оригинальных ресурсов в расширенном синтаксисе.

Yii использует имена расширений файлов для идентификации расширенного синтаксиса внутри ресурса. По умолчанию признаны следующие синтаксисы и имена расширений файлов:

- LESS²³: `.less`
- SCSS²⁴: `.scss`
- Stylus²⁵: `.styl`
- CoffeeScript²⁶: `.coffee`
- TypeScript²⁷: `.ts`

Yii ориентируется на установленные инструменты конвертации ресурсов препроцессора. Например, используя LESS²⁸, Вы должны установить команду `lessc` препроцессора.

Вы можете настроить команды препроцессора и поддерживать расширенный синтаксис сконфигурировав `yii\web\AssetManager::$converter` следующим образом:

```
return [
    'components' => [
        'assetManager' => [
            'converter' => [
                'class' => 'yii\web\AssetConverter',
                'commands' => [
                    'less' => ['css', 'lessc {from} {to} --no-color'],
                    'ts' => ['js', 'tsc --out {to} {from}'],
                ],
            ],
        ],
    ],
];
```

В примере выше, Вы задали поддержку расширенного синтаксиса через `yii\web\AssetConverter::$commands` свойство. Ключи массива - это имена расширений файлов (без ведущей точки), а значения массива - это образующийся файл ресурса имён расширений и команд для выполнения конвертации ресурса. Маркеры `{from}` и `{to}` в командах будут заменены соответственно исходным путём файла ресурсов и путём назначения файла ресурсов.

Примечание: Существуют другие способы работы с ресурсами расширенного синтаксиса, кроме того, который указан вы-

²³<http://lesscss.org/>

²⁴<http://sass-lang.com/>

²⁵<http://learnboost.github.io/stylus/>

²⁶<http://coffeescript.org/>

²⁷<http://www.typescriptlang.org/>

²⁸<http://lesscss.org/>

ше. Например, Вы можете использовать инструменты построения, такие как `grunt`²⁹ для отслеживания и автоматической конвертации ресурсов расширенного синтаксиса. В этом случае, Вы должны перечислить конечные CSS/JavaScript файлы в комплекте ресурсов вместо исходных файлов.

3.10.6 Объединение и Сжатие Ресурсов

Web страница может включать много CSS и/или JavaScript файлов. Чтобы сократить количество HTTP запросов и общий размер загрузки этих файлов, общепринятой практикой является объединение и сжатие нескольких CSS/JavaScript файлов в один или в более меньшее количество, а затем включение этих сжатых файлов вместо исходных в Web страницы.

Примечание: Комбинирование и сжатие ресурсов обычно необходимо, когда приложение находится в режиме продакшена. В режиме разработки, использование исходных CSS/JavaScript файлов часто более удобно для отладочных целей.

Далее, мы представим подход комбинирования и сжатия файлов ресурсов без необходимости изменения Вашего существующего кода приложения.

1. Найдите все комплекты ресурсов в Вашем приложении, которые Вы планируете скомбинировать и сжать.
2. Распределите эти комплекты в одну или несколько групп. Обратите внимание, что каждый комплект может принадлежать только одной группе.
3. Скомбинируйте/сожмите CSS файлы каждой группы в один файл. Сделайте то же самое для JavaScript файлов.
4. Определите новый комплект ресурсов для каждой группы:
 - Или установите `css` и `js` свойства. Соответствующие CSS и JavaScript файлы будут объединены.
 - Или настройте комплекты ресурсов каждой группы, установив их `css` и `js` свойства как пустые, и установите их `depends` свойство как новый комплект ресурсов, созданный для группы.

Используя этот подход, при регистрации комплекта ресурсов в представлении, автоматически регистрируется новый комплект ресурсов для

²⁹<http://gruntjs.com/>

группы, к которому исходный комплект принадлежит. В результате скомбинированные/сжатые файлы ресурсов включаются в страницу вместо исходных.

Пример

Давайте рассмотрим пример, чтобы объяснить вышеуказанный подход.

Предположим, ваше приложение имеет две страницы, X и Y. Страница X использует комплект ресурсов A, B и C, в то время, как страница Y использует комплект ресурсов, B, C и D.

У Вас есть два пути, чтобы разделить эти комплекты ресурсов. Первый - использовать одну группу, включающую в себя все комплекты ресурсов. Другой путь - положить комплект A в группу X, D в группу Y, а (B, C) в группу S. Какой из этих вариантов лучше? Это зависит. Первый способ имеет преимущество в том, что в обоих страницах одинаково скомбинированы файлы CSS и JavaScript, что делает HTTP кэширование более эффективным. С другой стороны, поскольку одна группа содержит все комплекты, размер скомбинированных CSS и JavaScript файлов будет больше, и таким образом увеличится время отдачи файла (загрузки страницы). Для простоты в этом примере, мы будем использовать первый способ, то есть использовать единую группу, содержащую все пакеты.

Примечание: Разделение комплекта ресурсов на группы это не тривиальная задача. Это, как правило, требует анализа реальных данных о трафике различных ресурсов на разных страницах. В начале вы можете начать с одной группы, для простоты.

Используйте существующие инструменты (например Closure Compiler³⁰, YUI Compressor³¹) для объединения и сжатия CSS и JavaScript файлов во всех комплектах. Обратите внимание, что файлы должны быть объединены в том порядке, который удовлетворяет зависимости между комплектами. Например, если комплект A зависит от B, который зависит от C и D, то Вы должны перечислить файлы ресурсов начиная с C и D, затем B, и только после этого A.

После объединения и сжатия, Вы получите один CSS файл и один JavaScript файл. Предположим, они названы как `all-xyz.css` и `all-xyz.js`, где `xyz` это временная метка или хэш, который используется, чтобы создать уникальное имя файла, чтобы избежать проблем с HTTP кэшированием.

Сейчас мы находимся на последнем шаге. Настройте `asset manager` в конфигурации вашего приложения, как показано ниже:

³⁰<https://developers.google.com/closure/compiler/>

³¹<https://github.com/yui/yuicompressor/>

```

return [
    'components' => [
        'assetManager' => [
            'bundles' => [
                'all' => [
                    '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']],
            ],
        ],
    ],
];

```

Как объяснено в подразделе Настройка Комплектов Ресурсов, приведенная выше конфигурация изменяет поведение по умолчанию каждого комплекта. В частности, комплекты A, B, C и D не имеют больше никаких файлов ресурсов. Теперь они все зависят от `all` комплекта, который содержит скомбинированные `all-xyz.css` и `all-xyz.js` файлы. Следовательно, для страницы X, вместо включения исходных файлов ресурсов из комплектов A, B и C, только два этих объединённых файла будут включены, то же самое произойдёт и со страницей Y.

Есть еще один трюк, чтобы сделать работу вышеуказанного подхода более отлаженной. Вместо изменения конфигурационного файла приложения напрямую, можно поставить комплект массива настроек в отдельный файл, и условно включить этот файл в конфигурацию приложения. Например,

```

return [
    'components' => [
        'assetManager' => [
            'bundles' => require(__DIR__ . '/' . (YII_ENV_PROD ? 'assets-
prod.php' : 'assets-dev.php')),
        ],
    ],
];

```

То есть, массив конфигурации комплекта ресурсов сохраняется в `assets-prod.php` для режима продакшена, и в `assets-dev.php` для режима не продакшена (разработки).

Использование команды `asset`

Yii предоставляет консольную команду с именем `asset` для автоматизации подхода, который мы только что описали.

Чтобы использовать эту команду, Вы должны сначала создать файл конфигурации для описания того, как комплекты ресурсов должны быть скомбинированы, и как они должны быть сгруппированы. Затем Вы можете использовать подкоманду `asset/template`, чтобы сгенерировать первый шаблон и затем отредактировать его под свои нужды.

```
yii asset/template assets.php
```

Данная команда сгенерирует файл с именем `assets.php` в текущей директории. Содержание этого файла можно увидеть ниже:

```
<?php
/**
 * Файл конфигурации команды консоли "yii asset".
 * Обратите внимание, что в консольной среде, некоторые псевдонимы путей,
 * такие как "@webroot" и "@web",
 * не могут быть использованы.
 * Пожалуйста, определите отсутствующие псевдонимы путей.
 */
return [
    // Настроить командуобратный/ вызов для сжатия файлов JavaScript:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file {to}',
    // Настроить командуобратный/ вызов для сжатия файлов CSS:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o {to}',
    // Список комплектов ресурсов для сжатия:
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\jQueryAsset',
    ],
    // Комплект ресурса после сжатия:
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Настройка менеджера ресурсов:
    'assetManager' => [
    ],
];
```

Вы должны изменить этот файл и указать в `bundles` параметре, какие комплекты Вы планируете объединить. В параметре `targets` вы должны указать, как комплекты должны быть поделены в группы. Вы можете указать одну или несколько групп, как уже было сказано выше.

Примечание: Так как псевдонимы путей `@webroot` и `@web` не могут быть использованы в консольном приложении, Вы должны явно задать их в файле конфигурации.

JavaScript файлы объединены, сжаты и записаны в `js/all-{hash}.js`, где `{hash}` перенесён из хэша результирующего файла.

Параметры `jsCompressor` и `cssCompressor` указывают на консольные команды или обратный вызов PHP, выполняющие JavaScript и CSS объединение/сжатие. По умолчанию Yii использует Closure Compiler³² для объединения JavaScript файлов и YUI Compressor³³ для объединения CSS файлов. Вы должны установить эти инструменты вручную или настроить данные параметры, чтобы использовать ваши любимые инструменты.

Вы можете запустить команду `asset` с файлом конфигурации для объединения и сжатия файлов ресурсов, а затем создать новый файл конфигурации комплекта ресурса `assets-prod.php`:

```
yii asset assets.php config/assets-prod.php
```

Сгенерированный файл конфигурации может быть включен в конфигурацию приложения, как описано в последнем подразделе.

Для справки: Команда `asset` является не единственной опцией для автоматического процесса объединения и сжатия ресурсов. Вы можете также использовать такой замечательный инструмент запуска приложений как `grunt`³⁴ для достижения той же цели.

Группировка Комплектов Ресурсов

В последнем подразделе, мы пояснили, как объединять все комплекты ресурсов в единый в целях минимизации HTTP запросов для файлов ресурсов, упоминавшихся в приложении. Это не всегда желательно на практике. Например, представьте себе, что Ваше приложение содержит “front end”, а также и “back end”, каждый из которых использует свой набор JavaScript и CSS файлов. В этом случае, объединение всех комплектов ресурсов с обеих сторон в один не имеет смысла потому, что комплекты ресурсов для “front end” не используются в “back end”, и это будет бесполезной тратой трафика - отправлять “back end” ресурсы, когда страница из “front end” будет запрошена.

Для решения вышеуказанной проблемы, вы можете разделить комплекты по группам и объединить комплекты ресурсов для каждой группы. Следующая конфигурация показывает, как Вы можете объединять комплекты ресурсов:

```
return [  
    ...  
    // Укажите выходной комплект для групп:
```

³²<https://developers.google.com/closure/compiler/>

³³<https://github.com/yui/yuicompressor/>

³⁴<http://gruntjs.com/>

```

'targets' => [
    'allShared' => [
        'js' => 'js/all-shared-{hash}.js',
        'css' => 'css/all-shared-{hash}.css',
        'depends' => [
            // Включаем все ресурсы поделённые между 'backend' и '
frontend'
            'yii\web\YiiAsset',
            'app\assets\SharedAsset',
        ],
    ],
    'allBackEnd' => [
        'js' => 'js/all-{hash}.js',
        'css' => 'css/all-{hash}.css',
        'depends' => [
            // Включаем только 'backend' ресурсы:
            'app\assets\AdminAsset'
        ],
    ],
    'allFrontEnd' => [
        'js' => 'js/all-{hash}.js',
        'css' => 'css/all-{hash}.css',
        'depends' => [], // Включаем все оставшиеся ресурсы
    ],
    ...
];

```

Как вы можете видеть, комплекты ресурсов поделены на три группы: `allShared`, `allBackEnd` и `allFrontEnd`. Каждая из которых зависит от соответствующего набора комплектов ресурсов. Например, `allBackEnd` зависит от `app\assets\AdminAsset`. При запуске команды `asset` с данной конфигурацией будут объединены комплекты ресурсов согласно приведенной выше спецификации.

Для справки: Вы можете оставить `depends` конфигурацию пустой для одного из намеченных комплектов. Поступая таким образом, данный комплект ресурсов будет зависеть от всех остальных комплектов ресурсов, от которых другие целевые комплекты не зависят.

3.11 Расширения

Расширения - это распространяемые программные пакеты, специально разработанные для использования в приложениях Yii и содержащие готовые функции. Например, расширение `yii2-debug` добавляет удобную отладочную панель в нижнюю часть каждой страницы вашего приложения, чтобы помочь вам разобраться в том, как генерируются

страницы. Вы можете использовать расширения для ускорения процесса разработки. Вы также можете оформить ваш код как расширение, чтобы поделиться с другими людьми результатами вашей работы.

Информация: Мы используем термин “расширение” для специфичных для Yii программных пакетов. Программные пакеты общего назначения, которые могут быть использованы без Yii, мы будем называть “пакет” или “библиотека”.

3.11.1 Использование расширений

Чтобы использовать расширение, вам необходимо установить его. Большинство расширений распространяются как пакеты Composer³⁵, которые могут быть установлены посредством следующих двух шагов:

1. Отредактируйте файл вашего приложения `composer.json`, указав, какие расширения (пакеты Composer) вы хотите установить.
2. Выполните команду `php composer.phar install`, чтобы установить указанные расширения.

Обратите внимание, что вам может потребоваться установить Composer³⁶, если у вас его нет.

По умолчанию, Composer устанавливает пакеты, зарегистрированные на Packagist³⁷ - крупнейшем репозитории для пакетов Composer с открытым исходным кодом. Вы также можете создать свой репозиторий³⁸ и настроить Composer для его использования. Это полезно, если вы разрабатываете закрытые расширения и хотите использовать их в нескольких своих проектах.

Расширения, установленные Composer’ом, хранятся в директории `BasePath/vendor`, где `BasePath` - базовая директория приложения. Composer - это менеджер зависимостей, и поэтому после установки пакета он также установит все зависимые пакеты.

Например, для установки расширения `yiisoft/yii2-image` нужно отредактировать ваш `composer.json` как показано далее:

```
{
    // ...

    "require": {
        // ... другие зависимости

        "yiisoft/yii2-image": "*"
    }
}
```

³⁵<https://getcomposer.org/>

³⁶<https://getcomposer.org/>

³⁷<https://packagist.org/>

³⁸<https://getcomposer.org/doc/05-repositories.md#repository>

```
}  
}
```

После установки вы можете увидеть директорию `yiiisoft/yii2-image`, находящуюся по пути `BasePath/vendor`. Также вы можете увидеть директорию `image/image`, которая содержит зависимый пакет.

Информация: `yiiisoft/yii2-image` является базовым расширением, которое разрабатывает и поддерживает команда разработчиков Yii. Все базовые расширения размещены на Packagist³⁹ и называются `yiiisoft/yii2-xyz`, где `xyz` является названием расширения.

Теперь вы можете использовать установленное расширение как часть вашего приложения. Следующий пример показывает, как вы можете использовать класс `yii\image\Image`, который содержится в расширении `yiiisoft/yii2-image`.

```
use Yii;  
use yii\image\Image;  
  
// генерация миниатюры изображения  
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)  
->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>  
50]);
```

Информация: Классы расширений автоматически загружаются автозагрузчиком классов Yii.

Ручная установка расширений

В некоторых редких случаях вы можете захотеть установить некоторые расширения вручную, а не полагаться на Composer. Чтобы сделать это, вы должны

1. загрузить архив с файлами расширения и распаковать его в директорию `vendor`.
2. установить автозагрузчики классов, предоставляемые расширениями, если таковые имеются.
3. загрузить и установить все зависимые расширения в соответствии с инструкциями.

³⁹<https://packagist.org/>

Если расширение не имеет автозагрузчика классов, но следует стандарту PSR-4⁴⁰, то вы можете использовать автозагрузчик классов, предоставленный Yii для загрузки классов расширений. Всё, что вам нужно сделать, это объявить псевдоним для корневого каталога расширения. Например, если вы установили расширение в директорию `vendor/mycompany/myext` и классы расширения находятся в пространстве имён `myext`, то вы можете включить следующий код в конфигурацию вашего приложения:

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

3.11.2 Создание расширений

Вы можете захотеть создать расширение, когда чувствуете необходимость поделиться своим хорошим кодом с другими людьми. Расширение может содержать любой код, который вам нравится, например, класс-помощник, виджет, модуль и т.д.

Рекомендуется создавать расширение как пакет Composer⁴¹, для того, чтобы его можно было легко установить и использовать, как описано в предыдущей главе.

Ниже приведены основные шаги, которым нужно следовать, чтобы создать пакет Composer.

1. Создайте проект для вашего расширения и разместите его в VCS репозитории, таком как `github.com`⁴². Разработка и поддержка расширения должна выполняться в этом репозитории.
2. В корневой директории проекта создайте файл под названием `composer.json`, в соответствии с требованиями Composer. Вы можете обратиться к следующему разделу за более подробной информацией.
3. Зарегистрируйте ваше расширение в репозитории Composer, таком как `Packagist`⁴³, чтобы другие пользователи могли найти и установить ваше расширение, используя Composer.

`composer.json`

Каждый пакет Composer должен иметь файл `composer.json` в своей корневой директории. Этот файл содержит метаданные о пакете. Вы можете найти полную спецификацию по этому файлу в Руководстве Composer⁴⁴.

⁴⁰<http://www.php-fig.org/psr/psr-4/>

⁴¹<https://getcomposer.org/>

⁴²<https://github.com>

⁴³<https://packagist.org/>

⁴⁴<https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup>

Следующий пример демонстрирует файл `composer.json` для расширения `yiisoft/yii2-imagine`:

```
{
    // название пакета
    "name": "yiisoft/yii2-imagine",

    // тип пакета
    "type": "yii2-extension",

    "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%3Aimagine",
        "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"
        }
    ],

    // зависимости пакета
    "require": {
        "yiisoft/yii2": "*",
        "imagine/imagine": "v0.5.0"
    },

    // указание автозагрузчика классов
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Название пакета Каждый пакет Composer должен иметь название, которое однозначно идентифицирует пакет среди остальных. Название пакета имеет формат `имяРазработчиканазваниеПроекта/`. Например, в пакете `yiisoft/yii2-imagine`, `yiisoft` является именем разработчика, а `yii2-imagine` - названием пакета.

НЕ используйте `yiisoft` в качестве имени разработчика, так как оно зарезервировано для использования в коде ядра Yii.

Мы рекомендуем использовать префикс `yii2-` в названии проекта для пакетов, являющихся расширениями Yii 2, например, `moёИмя/yii2-mywidget`

. Это позволит пользователям легче определить, что пакет является расширением Yii2.

Тип пакета Важно указать тип пакета вашего расширения как `yii2-extension`, чтобы пакет можно было распознать как расширение Yii во время установки.

Когда пользователь запускает команду `php composer.phar install` для установки расширения, файл `vendor/yiisoft/extensions.php` будет автоматически обновлён, чтобы включить информацию о новом расширении. Из этого файла приложение Yii может узнать, какие расширения установлены (информацию можно получить с помощью `yii\base\Application::$extensions`).

Зависимости Ваше расширение зависит от Yii (естественно). Вы можете посмотреть список зависимостей в секции `require`, входящей в файл `composer.json`. Если ваше расширение зависит от других расширений или сторонних библиотек, то вы также должны их перечислить. Убедитесь, что в ограничениях вы указали соответствующую версию (например, `1.*`, `@stable`) для каждой зависимости. Используйте стабильные версии зависимостей, когда будет выпущена стабильная версия вашего расширения.

Автозагрузка классов Для того, чтобы ваши классы были загружены автозагрузчиком классов Yii или автозагрузчиком классов Composer, вы должны внести секцию `autoload` в файл `composer.json`, как показано ниже:

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\image\\": ""
        }
    }
}
```

Вы можете перечислить один или несколько корневых пространств имён и соответствующие им пути.

Когда расширение установлено в приложение, Yii для каждого указанного корневого пространства имён создаст псевдоним, который указывает на директорию, соответствующую пространству имён. Например, указанная в секции `autoload` запись будет соответствовать псевдониму `@yii/image`.

Рекомендованные практики

Поскольку расширения предназначены для использования другими людьми, вам придётся приложить дополнительные усилия в процессе разработки. Ниже приведены некоторые общие и рекомендованные практики для создания высококачественных расширений.

Пространства имён Во избежание конфликтов имён, а также для того, чтобы ваши классы были автозагружаемыми, вы должны следовать стандарту PSR-4⁴⁵ или стандарту PSR-0⁴⁶ в использовании пространств имён и названии классов вашего расширения.

Пространства имён в ваших классах должны начинаться с `имяРазработчиканазваниеРасширения`, где `названиеРасширения` совпадает с названием проекта в названии пакета, за исключением того, что оно не должно содержать префикса `yii2-`. Например, для расширения `yiiisoft/yii2-image` мы используем `yii\image` в качестве пространства имён.

Не используйте `yii`, `yii2` или `yiiisoft` в качестве имени разработчика. Эти имена являются зарезервированными для использования в коде ядра Yii.

Классы начальной загрузки Иногда вы можете захотеть выполнить некоторый код своего расширения в стадии начальной загрузки приложения. Например, ваше расширение может ответить на событие приложения `beginRequest`, чтобы установить некоторые настройки окружения. Вы можете в инструкции по установке вашего приложения написать, что необходимо назначить обработчик события `beginRequest`, но лучшим способом будет сделать это автоматически.

Для достижения этой цели вы можете создать так называемый *класс начальной загрузки*, реализовав интерфейс `yii\base\BootstrapInterface`. Например,

```
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 () {
            // остальной код
        });
    }
}
```

⁴⁵<http://www.php-fig.org/psr/psr-4/>

⁴⁶<http://www.php-fig.org/psr/psr-0/>

Затем нужно добавить этот класс в файл `composer.json` вашего расширения, как показано далее,

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

Когда расширение будет установлено в приложение, Yii автоматически иницирует экземпляр класса начальной загрузки и вызовет его метод `bootstrap()` в процессе начальной загрузки каждого запроса.

Работа с базами данных Ваше расширение может иметь доступ к базам данных. Не думайте, что приложения, которые используют ваше расширение, всегда используют `Yii::$db` в качестве соединения с БД. Вместо этого вам следует объявить свойство `db` в классах, которым необходим доступ в БД. Это свойство позволит пользователям вашего расширения настроить соединение с БД, которое они будут использовать в вашем расширении. В качестве примера вы можете обратиться к классу `yii\caching\DbCache` и посмотреть, как он объявляет и использует свойство `db`.

Если в вашем приложении необходимо создать определённые таблицы БД или сделать изменения в схеме БД, вы должны

- создать файлы миграций для изменения схемы БД вместо простых SQL-файлов;
- попытаться сделать миграции, применимые к различным СУБД;
- избегать использования Active Record в миграциях.

Использование ресурсов Если ваше расширение является виджетом или модулем, то есть вероятность, что оно потребует некоторых ресурсов для работы. Например, модуль может отображать некоторые страницы, которые содержат изображения, JavaScript и CSS. Так как все файлы расширения находятся в директории, недоступной из интернета, у вас есть два варианта сделать директорию ресурсов непосредственно доступной из интернета:

- попросить пользователей расширения вручную скопировать файлы ресурсов в определённую, доступную из интернета папку;
- объявить связку ресурсов и полагаться на механизм публикации ресурсов, который автоматически копирует файлы, описанные в связке ресурсов в папку, доступную из интернета.

Мы рекомендуем вам использовать второй подход, чтобы ваше расширение было более простым в использовании для других людей.

Интернационализация и локализация

Ваше расширение может быть использовано в приложениях, поддерживающих разные языки! Поэтому, если ваше расширение отображает поддерживаемое конечному пользователю, вы должны попробовать интернационализировать и локализовать его. В частности,

- Если расширение отображает сообщения, предназначенные для конечных пользователей, сообщения должны быть обёрнуты в метод `Yii::t()` так, чтобы они могли быть переведены. Сообщения, предназначенные для разработчиков (например, внутренние сообщения исключений), не нужно переводить.
- Если расширение отображает числа, даты и т.п., они должны быть отформатированы, используя `yii\base\Formatter` с соответствующими правилами форматирования.

Для более подробной информации вы можете обратиться к разделу Интернационализация

Тестирование Вы хотите, чтобы ваше расширение было стабильным и не приносило проблем другим людям. Для достижения этой цели вы должны протестировать ваше расширение перед его публикацией.

Рекомендуется создавать различные тесты для покрытия кода вашего расширения, а не вручную тестировать его. Каждый раз перед тем, как выпустить новую версию расширения, вы можете просто запустить эти тесты чтобы убедиться, что всё работает правильно. Yii имеет поддержку тестирования, которая может помочь вам легче писать модульные, приёмочные и функциональные тесты. Для более подробной информации вы можете обратиться в раздел Тестирование.

Версионирование Вы можете давать каждому выпуску вашего расширения номер версии (например, 1.0.1). Мы рекомендуем вам придерживаться практик семантического версионирования⁴⁷ при определении, какой номер версии должен использоваться.

Публикация Чтобы позволить другим людям узнать о вашем расширении, необходимо опубликовать его.

Если это первый выпуск вашего расширения, вы должны зарегистрировать его в репозитории Composer, таком, как Packagist⁴⁸. После этого вам остаётся только создать тег выпуска (например, v1.0.1) в VCS репозитории вашего расширения и уведомить репозиторий Composer о новом выпуске. Люди смогут найти новую версию и установить или обновить расширение через репозиторий Composer.

⁴⁷<http://semver.org>

⁴⁸<https://packagist.org/>

В выпусках вашего расширения помимо файлов с кодом вы также должны рассмотреть вопрос о включении следующих файлов, которые помогут людям изучить и использовать ваше расширение:

- Файл `readme` в корневой директории пакета: он описывает, что ваше расширение делает, а также как его установить и использовать. Мы рекомендуем вам написать его в формате Markdown⁴⁹ и дать ему название `readme.md`.
- Файл `changelog` в корневой директории пакета: он описывает, какие изменения произошли в каждом выпуске. Этот файл может быть написан в формате Markdown и назван `changelog.md`.
- Файл `upgrade` в корневой директории пакета: он даёт инструкции о том, как обновить старые версии расширения. Этот файл может быть написан в формате Markdown и назван `upgrade.md`.
- Руководства пользователя, демо-версии, скриншоты и т.д.: они необходимы, если ваше расширение предоставляет много возможностей, которые невозможно полностью описать в файле `readme`.
- Документация API: ваш код должен быть документирован, чтобы позволить другим людям легко читать и понимать его. Вы можете обратиться к файлу класса `Object`⁵⁰, чтобы узнать, как нужно документировать код.

Информация: Ваши комментарии к коду могут быть написаны в формате Markdown. Расширение `yiiisoft/yii2-apidoc` предоставляет инструмент для генерации документации API на основе ваших комментариев.

Информация: Пока это не обязательно, но мы всё-таки рекомендуем вам придерживаться определённого стиля кодирования. Вы можете обратиться к стилю кодирования фреймворка⁵¹.

3.11.3 Базовые расширения

Yii предоставляет следующие базовые расширения, которые разрабатывает и поддерживает команда разработчиков Yii. Они все зарегистрированы на Packagist⁵² и могут быть легко установлены, как описано в подразделе Использование расширений.

- `yiiisoft/yii2-apidoc`⁵³: предоставляет расширяемый и высокопроизводительный генератор документации API. Оно также используется для генерации документации API фреймворка.

⁴⁹<http://daringfireball.net/projects/markdown/>

⁵⁰<https://github.com/yiiisoft/yii2/blob/master/framework/base/Object.php>

⁵¹<https://github.com/yiiisoft/yii2/wiki/Core-framework-code-style>

⁵²<https://packagist.org/>

⁵³<https://github.com/yiiisoft/yii2-apidoc>

- `yii2-authclient`⁵⁴: предоставляет набор наиболее часто используемых клиентов авторизации, таких, как Facebook OAuth2 клиент и GitHub OAuth2 клиент.
- `yii2-bootstrap`⁵⁵: предоставляет набор виджетов, которые являются компонентами и плагинами Bootstrap⁵⁶.
- `yii2-codeception`⁵⁷: предоставляет поддержку тестирования, основанного на Codeception⁵⁸.
- `yii2-debug`⁵⁹: предоставляет поддержку отладки в приложениях Yii. Когда это расширение используется, отладочная панель появится в нижней части каждой страницы. Это расширение также предоставляет набор отдельных страниц для отображения более подробной отладочной информации.
- `yii2-elasticsearch`⁶⁰: предоставляет поддержку использования Elasticsearch⁶¹. Оно включает в себя поддержку основных поисковых запросов, а также реализует шаблон проектирования Active Record, который позволяет хранить записи Active Record в Elasticsearch.
- `yii2-faker`⁶²: предоставляет поддержку использования Faker⁶³ для генерации фиктивных данных.
- `yii2-gii`⁶⁴: предоставляет веб-интерфейс для генерации кода, который является весьма расширяемым и может быть использован для быстрой генерации моделей, форм, модулей, CRUD и т.д.
- `yii2-imagine`⁶⁵: предоставляет часто используемые функции для работы с изображениями, основанные на библиотеке Imagine⁶⁶.
- `yii2-jui`⁶⁷: предоставляет набор виджетов, основанный на взаимодействиях и виджетах JQuery UI⁶⁸.
- `yii2-mongodb`⁶⁹: предоставляет поддержку использования MongoDB⁷⁰. Оно включает такие возможности, как базовые запросы, Active Record, миграции, кэширование, генерация кода и т.д.

⁵⁴<https://github.com/yiisoft/yii2-authclient>

⁵⁵<https://github.com/yiisoft/yii2-bootstrap>

⁵⁶<http://getbootstrap.com/>

⁵⁷<https://github.com/yiisoft/yii2-codeception>

⁵⁸<http://codeception.com/>

⁵⁹<https://github.com/yiisoft/yii2-debug>

⁶⁰<https://github.com/yiisoft/yii2-elasticsearch>

⁶¹<http://www.elasticsearch.org/>

⁶²<https://github.com/yiisoft/yii2-faker>

⁶³<https://github.com/fzaninotto/Faker>

⁶⁴<https://github.com/yiisoft/yii2-gii>

⁶⁵<https://github.com/yiisoft/yii2-imagine>

⁶⁶<http://imagine.readthedocs.org/>

⁶⁷<https://github.com/yiisoft/yii2-jui>

⁶⁸<http://jqueryui.com/>

⁶⁹<https://github.com/yiisoft/yii2-mongodb>

⁷⁰<http://www.mongodb.org/>

- `yiiisoft/yii2-redis`⁷¹: предоставляет поддержку использования `redis`⁷². Оно включает такие возможности, как базовые запросы, Active Record, кэширование и т.д.
- `yiiisoft/yii2-smarty`⁷³: предоставляет шаблонизатор, основанный на `Smarty`⁷⁴.
- `yiiisoft/yii2-sphinx`⁷⁵: предоставляет поддержку использования `Sphinx`⁷⁶. Оно включает такие возможности, как базовые запросы, Active Record, генерация кода и т.д.
- `yiiisoft/yii2-swiftmailer`⁷⁷: предоставляет возможности отправки email, основанные на `swiftmailer`⁷⁸.
- `yiiisoft/yii2-twig`⁷⁹: предоставляет шаблонизатор, основанный на `Twig`⁸⁰.

⁷¹<https://github.com/yiiisoft/yii2-redis>

⁷²<http://redis.io/>

⁷³<https://github.com/yiiisoft/yii2-smarty>

⁷⁴<http://www.smarty.net/>

⁷⁵<https://github.com/yiiisoft/yii2-sphinx>

⁷⁶<http://sphinxsearch.com>

⁷⁷<https://github.com/yiiisoft/yii2-swiftmailer>

⁷⁸<http://swiftmailer.org/>

⁷⁹<https://github.com/yiiisoft/yii2-twig>

⁸⁰<http://twig.sensiolabs.org/>

Глава 4

Обработка запросов

4.1 Предзагрузка

Предзагрузка это процесс настройки рабочей среды до того, как будет запущено приложение и обработан входящий запрос. Предзагрузка осуществляется в двух местах: во входном скрипте и в приложении.

Во входном скрипте, регистрируются автозагрузчики классов различных библиотек. Этот процесс включает в себя автозагрузчик классов Composer через `autoload.php` файл и автозагрузчик классов Yii через его `yii` файл. Затем входной скрипт загружает конфигурацию приложения и создает объект приложения.

В конструкторе приложения происходит следующий процесс предзагрузки:

1. Вызывается метод `preInit()`, которые конфигурирует свойства приложения, имеющие наивысший приоритет, такие как `basePath`;
2. Регистрируется **обработчик ошибок**;
3. Происходит инициализация свойств приложения согласно заданной конфигурации;
4. Вызывается метод `init()`, который в свою очередь вызывает метод `bootstrap()` для запуска компонентов предзагрузки.
 - Подключается файл манифеста `vendor/yiisoft/extensions.php`;
 - Создаются и запускаются компоненты предзагрузки объявленные в расширениях;
 - Создаются и запускаются компоненты приложения и/или модули, объявленные в свойстве предзагрузка приложения.

Поскольку предзагрузка осуществляется прежде чем будет обработан *каждый* запрос, то очень важно, чтобы этот процесс был легким и максимально оптимизированным.

Старайтесь не регистрировать слишком много компонентов в предзагрузке. Компонент предзагрузки нужен только тогда, когда он должен участвовать в полном жизненном цикле процесса обработки запроса. Например, если модуль должен зарегистрировать дополнительные правила парсинга URL, то он должен быть указан в свойстве предзагрузка, чтобы новые правила URL были учтены при обработке запроса.

В производственном режиме включите байткод кэшеры, такие как РНР OPcache¹ или APC², для минимизации времени подключения и парсинг php файлов.

Некоторые большие приложения могут иметь сложную конфигурацию, которая разделена на несколько мелких файлов. Если это тот самый случай, возможно вам стоит кэшировать весь конфигурационный файл и загружать его прямо из кэша до создания объекта приложения во входном скрипте.

4.2 Работа с URL

Замечание: раздел находится в разработке.

Концепция работы с URL в Yii довольно проста. Предполагается, что в приложении используются внутренние маршруты и параметры вместо жестко заданных URL. Тогда фреймворк сам преобразует маршруты в URL и обратно, в соответствии с конфигурацией URL менеджера. Такой подход позволяет изменять вид URL на всем сайте, редактируя единственный конфигурационный файл не трогая код самого приложения.

4.2.1 Внутренние маршруты

При создании приложения с помощью Yii, вы будете работать с внутренними маршрутами, которые также часто называются маршрутами с параметрами. Каждому контроллеру и действию соответствует внутренний маршрут, например `site/index` или `user/create`. В первом примере, `site` - это ID контроллера а `create` это ID действия. Если контроллер находится в модуле, то перед его ID контроллера в маршруте ставится ID модуля, например так: `blog/post/index`. Здесь `blog` - это ID модуля, а `post` и `index` - ID контроллера и действия соответственно.

4.2.2 Создание URL

Самое важное при работе с URL = всегда создавать их через URL manager. Это встроенный компонент приложения, к которому можно обращаться по имени `urlManager` как в консольном, так и в веб-приложении,

¹<http://php.net/manual/ru/intro.opcache.php>

²<http://php.net/manual/ru/book.apc.php>

таким образом: `\Yii::$app->urlManager`. Через этот компонент доступны следующие методы создания URL: `createUrl($params)` - `createAbsoluteUrl($params, $schema = null)`

Метод `createUrl` создает относительный от корня приложения URL, например `index.php/site/index`. Метод `createAbsoluteUrl` создает абсолютный URL, добавляя к относительному протокол и имя хоста: `http://www.example.com/index.php/site/index`. `createUrl` больше подходит для URL внутри самого приложения, тогда как `createAbsoluteUrl` - для создания ссылок на внешние ресурсы и для них, для решения задач отправки почты, генерации RSS и т.п.

Примеры:

```
echo \Yii::$app->urlManager->createUrl(['site/page', 'id' => 'about']);
// /index.php/site/page/id/about/
echo \Yii::$app->urlManager->createUrl(['date-time/fast-forward', 'id' =>
105])
// /index.php?r=date-time/fast-forward&id=105
echo \Yii::$app->urlManager->createAbsoluteUrl('blog/post/index');
// http://www.example.com/index.php/blog/post/index/
```

Точный формат URL зависит от конфигурации URL manager. В другой конфигурации примеры выше могут выводить:

- `/site/page/id/about/`
- `/index.php?r=site/page&id=about`
- `/index.php?r=date-time/fast-forward&id=105`
- `/index.php/date-time/fast-forward?id=105`
- `http://www.example.com/blog/post/index/`
- `http://www.example.com/index.php?r=blog/post/index`

Чтобы упростить создание URL рекомендуется пользоваться встроенным хелпером `yii\helpers\Url`. Покажем, как работает хелпер на примерах. Предположим, что мы находимся на `/index.php?r=management/default/users&id=10`. Тогда:

```
use yii\helpers\Url;

// текущий активный маршрут
// /index.php?r=management/default/users
echo Url::to('');

// тот же контроллер, другое действие
// /index.php?r=management/default/page&id=contact
echo Url::toRoute(['page', 'id' => 'contact']);

// тот же модуль, другие контроллер и действие
// /index.php?r=management/post/index
echo Url::toRoute('post/index');

// абсолютный маршрут вне зависимости от того, в каком контроллере
происходит вызов
```

```
// /index.php?r=site/index
echo Url::toRoute('/site/index');

// url для регистрозависимого действия hiTech текущего контроллера
// /index.php?r=management/default/hi-tech
echo Url::toRoute('hi-tech');

// url для регистрозависимого контроллера, 'DateTimeController::
    actionFastForward'
// /index.php?r=date-time/fast-forward&id=105
echo Url::toRoute(['/date-time/fast-forward', 'id' => 105]);

// получение URL через alias
// http://google.com/
Yii::setAlias('@google', 'http://google.com/');
echo Url::to('@google');

// получение URL домой""
// /index.php?r=site/index
echo Url::home();

Url::remember(); // сохранить URL, чтобы использовать его позже
Url::previous(); // получить ранее сохраненный URL
```

> **Совет:** чтобы сгенерировать URL с хэштегом, например `/index.php?r=site/page&id=100#title`, укажите параметр `#` в хелпере таким образом: `Url::to(['post/read', 'id' => 100, '#' => 'title'])`.

Также существует метод `Url::canonical()`, который позволяет создать канонический URL³ (статья на англ., перевода пока нет) для текущего действия. Этот метод при создании игнорирует все параметры действия кроме тех, которые были переданы как аргументы. Пример:

```
namespace app\controllers;

use yii\web\Controller;
use yii\helpers\Url;

class CanonicalController extends Controller
{
    public function actionTest($page)
    {
        echo Url::canonical();
    }
}
```

При этом при текущем URL `/index.php?r=canonical/test&page=hello&number=42` канонический URL будет `/index.php?r=canonical/test&page=hello`.

³https://en.wikipedia.org/wiki/Canonical_link_element

4.2.3 Модификация URL

По умолчанию Yii использует url формата query string, например `/index.php?r=news/view&id=100`. Чтобы сделать более человеко-понятные URL⁴, например для улучшения их читабельности, необходимо сконфигурировать компонент `urlManager` в конфигурационном файле приложения. Придав параметру `enablePrettyUrl` значение `true`, мы тем самым получим URL такого вида: `/index.php/news/view?id=100`, а выставив параметр `showScriptName` => `false` мы исключим `index.php` из URL. Вот фрагмент конфигурационного файла:

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
        ],
    ],
];
```

Обратите внимание, что конфигурация с `'showScriptName' => false` будет работать только, если веб сервер был должным образом сконфигурирован. Смотрите раздел `installation` (ссылка не работает).

Именованные параметры

Правило может быть связано с несколькими GET параметрами. Эти GET параметры появляются в паттерне правила как специальные управляющие конструкции в таком формате:

```
ИмяПараметраПаттернПараметра
<:>
```

`ИмяПараметра` это имя GET параметра, а необязательный `ПаттернПараметра` - это регулярное выражение, которое используется для того, чтобы найти определенный GET параметр. В случае, если `ПаттернПараметра` не указан, это значит, что значение параметра это любая последовательность символов, кроме `/`. Правила используются и при парсинге URL, и при их создании; при создании URL параметры будут заменены на соответствующие значения, а при парсинге - GET параметры с соответствующими именами получат соответствующие значения из URL.

Приведем несколько примеров, чтобы пояснить, как работают правила. Предположим, что наш массив правил содержит три правила:

```
[
    'posts' => 'post/list',
```

⁴https://ru.wikipedia.org/wiki/%D0%A7%D0%9F%D0%A3_%28%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82%29

```

    'post/<id:\d+>'=>'post/read',
    'post/<year:\d{4}>/<title>'=>'post/read',
]

```

- Вызов `Url::toRoute('post/list')` генерирует `/index.php/posts`. Применяется первое правило.
- Вызов `Url::toRoute(['post/read', 'id' => 100])` генерирует `/index.php/post/100`. Применяется второе правило.
- Вызов `Url::toRoute(['post/read', 'year' => 2008, 'title' => 'a sample post'])` генерирует `/index.php/post/2008/a%20sample%20post`. Применяется третье правило.
- Вызов `Url::toRoute('post/read')` генерируется `/index.php/post/read`. Не применяется ни одного правила, вместо этого URL формируется по алгоритму по умолчанию.

Таким образом, при использовании `createUrl` для генерации URL, маршрут и переданные методу `GET` параметры используются для принятия решения о том, какое правило применить. Конкретное правило используется для генерации URL, если каждый параметр, указанный в правиле, может быть найден среди `GET` параметров, переданных `createUrl`, и сам маршрут, указанный в правиле, соответствует маршруту, переданному в качестве параметра.

Если параметров, переданных `Url::toRoute` больше, чем указано в правиле, то дополнительные параметры будут указаны в формате строки запроса. Например, при вызове `Url::toRoute(['post/read', 'id' => 100, 'year' => 2008])`, мы получим `/index.php/post/100?year=2008`.

Как было сказано ранее, другое назначение правил - парсинг URL. Это процесс, обратный их созданию. Например, когда пользователь запрашивает `/index.php/post/100`, будет применено второе правило из примера выше, которое извлечет маршрут `post/read` и `GET` параметр `['id' => 100]` (который можно получить так: `Yii::$app->request->get('id')`).

Параметры в маршрутах

Мы можем обращаться к именованным параметрам в маршрутной части правила. Это позволяет нескольким маршрутам подпадать под правило, в соответствии с переданными параметрами. Это также может помочь минимизировать количество правил URL в приложении, тем самым улучшив его общую производительность.

Вот пример, показывающий как использовать маршруты с именованными параметрами.

```

[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
    => '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/read',
    '<controller:(post|comment)>s' => '<controller>/list',
]

```

В этом примере, мы использовали два именованных параметра в маршрутной части правил: `controller` и `action`. Первый параметр подходит в случае, если `controller ID` - это `post` или `comment`, а последний - в случае, если `action ID` - `create`, `update` или `delete`. Имена параметров могут быть разными, но не должны совпадать по имени с GET-параметрами, которые могут появляться в URL.

Используя эти правила, URL `/index.php/post/123/create` после парсинга будет иметь маршрут `post/create` с GET параметром `id=123`. А с маршрутом `comment/list` и GET параметром `page=2` можно создать URL `/index.php/comments?page=2`.

Параметры в имени хоста

Существует возможность включать имена хостов в правила парсинга и создания URL. Например, может возникнуть необходимость получить часть имени хоста в качестве GET параметра, чтобы работать с поддоменами. Например, URL `http://admin.example.com/en/profile` может быть разобран на GET параметры `user=admin` и `lang=en`. Аналогично, правила с именами хостов могут использоваться для создания URL.

Чтобы использовать параметры в имени хоста, просто объявите правило с именем хоста, например,

```
[
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
]
```

В этом примере первый сегмент имени хоста обрабатывается как имя пользователя (`user`), тогда как первый сегмент пути обрабатывается как параметр языка (`lang`). Правило относится к маршруту `user/profile`.

Обратите внимание, что атрибут `yii\web\UrlManager::$showScriptName` не оказывает действия на правила, в которых используются параметры в имени хоста.

Также учтите, что любое правило с параметрами в имени хоста не должно содержать подпапку, если приложение размещено в подпапке `web` рута. Например, если приложение находится в `http://www.example.com/sandbox/blog`, тогда нам все равно в правиле нужно его прописать без `sandbox/blog`.

Добавка суффикса URL

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'suffix' => '.html',
        ],
    ],
]
```

```
],
];
```

Обработка REST запросов

TBD: - RESTful маршрутизация: `yii\filters\VerbFilter`, `yii\web\UrlManager::$rules` - Json API: - response: `yii\web\Response::$format` - request: `yii\web\Request::$parsers`, `yii\web\JsonParser`

4.2.4 Парсинг URL

Помимо создания URL Yii также умеет парсить URL, получая на выходе маршруты и параметры.

Строгий парсинг URL

По умолчанию если для URL не подошло ни одного заданного правила и URL соответствует стандартному формату, как, например `/site/page`, Yii делает попытку запустить указанный action соответствующего контроллера. Можно запретить обрабатывать URL стандартного формата, тогда, если URL не соответствует ни одному правилу, будет выброшена 404 ошибка.

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'enableStrictParsing' => true,
        ],
    ],
];
```

4.2.5 Создание классов для произвольных правил

`yii\web\UrlRule` класс используется для парсинга URL на параметры и создания URL по параметрам. Обычное решение для правил подойдет для большинства проектов, но бывают ситуации, когда лучшим выбором будет использования своего класса для задания правил. Например, на сайте автомобильного дилера могут быть URL типа `ПроизводительМодель//`, где и `Производитель` и `Модель` хранятся в таблице БД. Обычное правило не сработает, т.к. оно основывается на статически заданных регулярных выражений, и извлечение информации из БД в нем не предусмотрено.

Мы можем создать новый класс для правил URL, унаследовав его от `yii\web\UrlRule` и используя его в одном или нескольких URL правил. Ниже - реализация для вышеописанного примера с сайтом автомобильного дилера. Вот указание произвольного правила в конфигурации приложения:

```
// ...
'components' => [
    'urlManager' => [
        'rules' => [
            '<action:(login|logout|about)>' => 'site/<action>',

            // ...

            ['class' => 'app\components\CarUrlRule', 'connectionID' => 'db',
/* ... */],
        ],
    ],
],
```

В примере мы используем произвольное URL правило `CarUrlRule` для обработки URL формата `ПроизводительМодель//`.

Код самого класса может быть примерно таким:

```
namespace app\components;

use yii\web\UrlRule;

class CarUrlRule extends UrlRule
{
    public $connectionID = 'db';

    public function init()
    {
        if ($this->name === null) {
            $this->name = __CLASS__;
        }
    }

    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; // это правило не подходит
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\\w+)/(\\w+)?$%', $pathInfo, $matches)) {
            // check $matches[1] and $matches[3] to see
            // if they match a manufacturer and a model in the database
            // If so, set $params['manufacturer'] and/or $params['model']
            // and return ['car/index', $params]
        }
    }
}
```

```
        return false; // это правило не подходит
    }
}
```

Кроме использования произвольных классов URL в случаях, подобных примеру выше, их также можно использовать для других целей. Например, мы можем написать класс правила для логгирования парсинга URL и создания запросов. Это может быть полезно на этапе разработки. Мы также можем написать класс правила чтобы показывать особую 404 страницу в случае, если другие правила не подошли для обработки текущего запроса. В этом случае произвольное правило должно стоять последним.

4.3 Запросы

Запросы, сделанные к приложению, представлены в терминах `yii\web\Request` объектов, которые предоставляют информацию о параметрах запроса, HTTP заголовках, cookies и т.д. Для получения доступа к текущему запросу вы должны обратиться к объекту `request` application component, который по умолчанию является экземпляром `yii\web\Request`.

4.3.1 Параметры запроса

Чтобы получить параметры запроса, вы должны вызвать методы `get()` и `post()` компонента `request`. Они возвращают значения переменных `$_GET` и `$_POST` соответственно. Например,

```
$request = Yii::$app->request;

$get = $request->get();
// эквивалентно: $get = $_GET;

$id = $request->get('id');
// эквивалентно: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// эквивалентно: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// эквивалентно: $post = $_POST;

$name = $request->post('name');
// эквивалентно: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// эквивалентно: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

Информация: Вместо того, чтобы обращаться напрямую к переменным `$_GET` и `$_POST` для получения параметров запроса,

рекомендуется чтобы вы обращались к ним через компонент `request` как было показано выше. Это упростит написание тестов, поскольку вы можете создать mock компонент запроса с не настоящими данными запроса.

При реализации RESTful API, зачастую вам требуется получить параметры, которые были отправлены через PUT, PATCH или другие методы запроса. Вы можете получить эти параметры, вызвав метод `yii\web\Request::getBodyParam()`. Например,

```
$request = Yii::$app->request;  
  
// возвращает все параметры  
$params = $request->bodyParams;  
  
// возвращает параметр "id"  
$param = $request->getBodyParam('id');
```

Информация: В отличие от GET параметров, параметры, которые были переданы через POST, PUT, PATCH и д.р. отправляются в теле запроса. Компонент `request` будет обрабатывать эти параметры, когда вы попытаетесь к ним обратиться через методы, описанные выше. Вы можете настроить способ обработки этих параметров через настройку свойства `yii\web\Request::$parsers`.

4.3.2 Методы запроса

Вы можете получить названия HTTP метода, используемого в текущем запросе, обратившись к выражению `Yii::$app->request->method`. Также имеется целый набор логических свойств для проверки соответствует ли текущий метод определённому типу запроса. Например,

```
$request = Yii::$app->request;  
  
if ($request->isAjax) { // является ли текущий запрос AJAX запросом }  
if ($request->isGet) { // является ли текущий запрос GET запросом }  
if ($request->isPost) { // является ли текущий запрос POST запросом }  
if ($request->isPut) { // является ли текущий запрос PUT запросом }
```

4.3.3 URL запроса

Компонент `request` предоставляет множество способов изучения текущего запрашиваемого URL.

Если предположить, что URL запроса будет `http://example.com/admin/index.php/product?id=100`, то вы можете получить различные части этого адреса так как это показано ниже:

- **url**: вернёт адрес `/admin/index.php/product?id=100`, который содержит URL без информации об имени хоста.
- **absoluteUrl**: вернёт адрес `http://example.com/admin/index.php/product?id=100`, который содержит полный URL, включая имя хоста.
- **hostInfo**: вернёт адрес `http://example.com`, который содержит только имя хоста.
- **pathInfo**: вернёт адрес `/product`, который содержит часть между адресом начального скрипта и параметрами запроса, которые идут после знака вопроса.
- **queryString**: вернёт адрес `id=100`, который содержит часть URL после знака вопроса.
- **baseUrl**: вернёт адрес `/admin`, который является частью URL после информации о хосте и перед именем входного скрипта.
- **scriptUrl**: вернёт адрес `/admin/index.php`, который содержит URL без информации о хосте и параметрах запроса.
- **serverName**: вернёт адрес `example.com`, который содержит имя хоста в URL.
- **serverPort**: вернёт `80`, что является адресом порта, который использует веб-сервер.

4.3.4 HTTP заголовки

Вы можете получить информацию о HTTP заголовках через `header collection`, возвращаемыми свойством `yii\web\Request::$headers`. Например,

```
// переменная $headers является объектом yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// возвращает значения заголовка Accept
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { // есть ли в запросе заголовок User-Agent
}
```

Компонент `request` также предоставляет доступ к некоторым часто используемым заголовкам, включая

- **userAgent**: возвращает значение заголовка `User-Agent`.
- **contentType**: возвращает значение заголовка `Content-Type`, который указывает на MIME тип данных в теле запроса.
- **acceptableContentTypes**: возвращает список MIME типов данных, которые принимаются пользователем. Возвращаемый список типов будет отсортирован по показателю качества. Типы с более высокими показателями будут первыми в списке.
- **acceptableLanguages**: возвращает языки, которые поддерживает пользователь. Список языков будет отсортирован по уровню предпочтения. Наиболее предпочитаемый язык будет первым в списке.

Если ваше приложение поддерживает множество языков и вы хотите показать страницу на языке, который предпочитает пользователь, то вы можете воспользоваться языковым методом согласования (negotiation) `yii\web\Request::getPreferredLanguage()`. Этот метод принимает список поддерживаемых языков в вашем приложении, сравнивает их с `acceptableLanguages` и возвращает наиболее подходящий язык.

Подсказка: Вы также можете использовать фильтр `ContentNegotiator` для динамического определения какой тип содержимого и язык должен использоваться в ответе. Фильтр реализует согласование содержимого на основе свойств и методов, описанных выше.

4.3.5 Информация о клиенте

Вы можете получить имя хоста и IP адрес пользователя через свойства `userHost` и `userIP` соответственно. Например,

```
$userHost = Yii::$app->request->userHost;  
$userIP = Yii::$app->request->userIP;
```

4.4 Ответы

Когда приложение заканчивает обработку запроса, оно генерирует объект **ответа** и отправляет его пользователю. Объект ответа содержит такие данные, как HTTP-код состояния, HTTP-заголовки и тело ответа. Конечная цель разработки Web-приложения состоит в создании объектов ответа на различные запросы.

В большинстве случаев вам придется иметь дело с компонентом приложения `response`, который по умолчанию является экземпляром класса `yii\web\Response`. Однако Yii также позволяет вам создавать собственные объекты ответа и отправлять их пользователям. Это будет рассмотрено ниже.

В данном разделе мы опишем, как составлять ответы и отправлять их пользователям.

4.4.1 Код состояния

Первое, что вы делаете при построении ответа, — определяете, был ли успешно обработан запрос. Это реализуется заданием свойству `yii\web\Response::$statusCode` значения, которое может быть одним из валидных HTTP-кодов состояния⁵. Например, чтобы показать, что запрос был успешно обработан, вы можете установить значение кода состояния равным 200:

⁵<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

```
Yii::$app->response->statusCode = 200;
```

Однако в большинстве случаев явная установка не требуется так как значение `yii\web\Response::$statusCode` по умолчанию равно 200. Если же вам нужно показать, что запрос не удался, вы можете выбросить соответствующее HTTP-исключение:

```
throw new \yii\web\NotFoundHttpException;
```

Когда обработчик ошибок поймает исключение, он извлечёт код состояния из исключения и назначит его ответу. Исключение `yii\web\NotFoundHttpException` в коде выше представляет HTTP-код состояния 404. В Yii предопределены следующие HTTP-исключения:

- `yii\web\BadRequestHttpException`: код состояния 400.
- `yii\web\ConflictHttpException`: код состояния 409.
- `yii\web\ForbiddenHttpException`: код состояния 403.
- `yii\web\GoneHttpException`: код состояния 410.
- `yii\web\MethodNotAllowedHttpException`: код состояния 405.
- `yii\web\NotAcceptableHttpException`: код состояния 406.
- `yii\web\NotFoundHttpException`: код состояния 404.
- `yii\web\ServerErrorHttpException`: код состояния 500.
- `yii\web\TooManyRequestsHttpException`: код состояния 429.
- `yii\web\UnauthorizedHttpException`: код состояния 401.
- `yii\web\UnsupportedMediaTypeHttpException`: код состояния 415.

Если в приведённом выше списке нет исключения, которое вы хотите выбросить, вы можете создать его, расширив класс `yii\web\HttpException`, или выбросить его напрямую с кодом состояния, например:

```
throw new \yii\web\HttpException(402);
```

4.4.2 HTTP-заголовки

Вы можете отправлять HTTP-заголовки, работая с коллекцией заголовков компонента `response`:

```
$headers = Yii::$app->response->headers;  
  
// добавить заголовок Pragma. Уже имеющиеся Pragma-заголовки НЕ будут  
// перезаписаны.  
$headers->add('Pragma', 'no-cache');  
  
// установить заголовок Pragma. Любые уже имеющиеся Pragma-заголовки будут  
// сброшены.  
$headers->set('Pragma', 'no-cache');  
  
// удалить заголовок или( заголовки) Pragma и вернуть их значения массивом  
$values = $headers->remove('Pragma');
```

Информация: названия заголовков не чувствительны к регистру символов. Заново зарегистрированные заголовки не отсылаются пользователю до вызова `yii\web\Response::send()`.

4.4.3 Тело ответа

Большинство ответов должны иметь тело, содержащее то, что вы хотите показать пользователям.

Если у вас уже имеется отформатированная строка для тела, вы можете присвоить её свойству `yii\web\Response::$content` объекта запроса:

```
Yii::$app->response->content = 'hello world!';
```

Если ваши данные перед отправкой конечным пользователям нужно привести к определённому формату, вам следует установить значения двух свойств: `format` и `data`. Свойство `format` определяет, в каком формате следует возвращать данные из `data`. Например:

```
$response = Yii::$app->response;
$response->format = \yii\web\Response::FORMAT_JSON;
$response->data = ['message' => 'hello world'];
```

Yii из коробки имеет поддержку следующих форматов, каждый из которых реализован классом **форматтера**. Вы можете настроить эти форматтеры или добавить новые через свойство `yii\web\Response::$formatters`.

- HTML: реализуется классом `yii\web\HtmlResponseFormatter`.
- XML: реализуется классом `yii\web\XmlResponseFormatter`.
- JSON: реализуется классом `yii\web\JsonResponseFormatter`.
- JSONP: реализуется классом `yii\web\JsonResponseFormatter`.

Хотя тело запроса может быть явно установлено показанным выше способом, в большинстве случаев вы можете задавать его неявно через возвращаемое значение методов действий. Типичный пример использования:

```
public function actionIndex()
{
    return $this->render('index');
}
```

Действие `index` в коде выше возвращает результат рендеринга представления `index`. Возвращаемое значение будет взято компонентом `response`, отформатировано и затем отправлено пользователям.

Так как по умолчанию форматом ответа является HTML, в методе действия следует вернуть строку. Если вы хотите использовать другой формат ответа, необходимо настроить его перед отправкой данных:

```
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
```

```
return [  
    'message' => 'hello world',  
    'code' => 100,  
];  
}
```

Как уже было сказано, кроме использования стандартного компонента приложения `response` вы также можете создавать свои объекты ответа и отправлять их конечным пользователям. Вы можете сделать это, возвращая такой объект в методе действия:

```
public function actionInfo()  
{  
    return \Yii::createObject([  
        'class' => 'yii\web\Response',  
        'format' => \yii\web\Response::FORMAT_JSON,  
        'data' => [  
            'message' => 'hello world',  
            'code' => 100,  
        ],  
    ]);  
}
```

Примечание: создавая собственные объекты ответов, вы не сможете воспользоваться конфигурацией компонента `response`, настроенной вами в конфигурации приложения. Тем не менее, вы можете воспользоваться внедрением зависимости, чтобы применить общую конфигурацию к вашим новым объектам ответа.

4.4.4 Перенаправление браузера

Перенаправление браузера основано на отправке HTTP-заголовка `Location`. Так как данная возможность широко применяется, Yii имеет средства для её использования.

Вы можете перенаправить браузер пользователя на URL-адрес, вызвав метод `yii\web\Response::redirect()`. Этот метод использует указанный URL-адрес в качестве значения заголовка `Location` и возвращает сам объект ответа. В методе действия вы можете вызвать короткую версию этого метода — `yii\web\Controller::redirect()`. Например:

```
public function actionOld()  
{  
    return $this->redirect('http://example.com/new', 301);  
}
```

В приведённом выше коде метод действия возвращает результат `redirect()`. Как говорилось выше, объект ответа, возвращаемый методом действия, будет использоваться в качестве ответа конечным пользователям.

В коде, находящемся вне методов действий, следует использовать `yii\web\Response::redirect()` и непосредственно после него — метод `yii\web\Response::send()`. Так можно быть уверенным, что к ответу не будет добавлено нежелательное содержимое.

```
\Yii::$app->response->redirect('http://example.com/new', 301)->send();
```

Информация: По умолчанию метод `yii\web\Response::redirect()` устанавливает код состояния ответа равным 302, сообщая браузеру, что запрашиваемый ресурс *временно* находится по другому URI-адресу. Вы можете передать код состояния 301, чтобы сообщить браузеру, что ресурс перемещён *навсегда*.

Если текущий запрос является AJAX-запросом, отправка заголовка `Location` не заставит браузер автоматически осуществить перенаправление. Чтобы решить эту задачу, метод `yii\web\Response::redirect()` устанавливает значение заголовка `X-Redirect` равным URL для перенаправления. На стороне клиента вы можете написать JavaScript-код для чтения значения этого заголовка и перенаправления браузера соответственно.

Информация: Yii поставляется с JavaScript-файлом `yii.js`, который предоставляет набор часто используемых JavaScript-утилит, включая и перенаправление браузера на основе заголовка `X-Redirect`. Следовательно, если вы используете этот JavaScript-файл (зарегистрировав пакет ресурсов `yii\web\YiiAsset`), вам не нужно писать дополнительный код для поддержки AJAX-перенаправления.

4.4.5 Отправка файлов

Как и перенаправление браузера, отправка файлов является ещё одной возможностью, основанной на определённых HTTP-заголовках. Yii предоставляет набор методов для решения различных задач по отправке файлов. Все они поддерживают HTTP-заголовок `range`.

- `yii\web\Response::sendFile()`: отправляет клиенту существующий файл.
- `yii\web\Response::sendContentAsFile()`: отправляет клиенту строку как файл.
- `yii\web\Response::sendStreamAsFile()`: отправляет клиенту существующий файловый поток как файл.

Эти методы имеют одинаковую сигнатуру и возвращают объект ответа. Если отправляемый файл очень велик, следует использовать `yii\web\Response::sendStreamAsFile()`, так как он более эффективно использует оперативную память. Следующий пример показывает, как отправить файл в действии контроллера:

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

При вызове метода отправки файла вне методов действий чтобы быть уверенным, что к ответу не будет добавлено никакое нежелательное содержимое, следует вызвать сразу после него `yii\web\Response::send()`.

```
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

Некоторые Web-серверы поддерживают особый режим отправки файлов, который называется *X-Sendfile*. Идея в том, чтобы перенаправить запрос файла Web-серверу, который отдаст файл пользователю самостоятельно. В результате Web-приложение может завершиться раньше, пока Web-сервер ещё пересылает файл. Чтобы использовать эту возможность, воспользуйтесь методом `yii\web\Response::xSendFile()`. Далее приведены ссылки на то, как включить X-Sendfile для популярных Web-серверов:

- Apache: X-Sendfile⁶
- Lighttpd v1.4: X-LIGHTTPD-send-file⁷
- Lighttpd v1.5: X-Sendfile⁸
- Nginx: X-Accel-Redirect⁹
- Cherokee: X-Sendfile and X-Accel-Redirect¹⁰

4.4.6 Отправка ответа

Содержимое ответа не отправляется пользователю до вызова метода `yii\web\Response::send()`. По умолчанию он вызывается автоматически в конце метода `yii\base\Application::run()`. Однако, чтобы ответ был отправлен немедленно, вы можете вызвать этот метод явно.

Для отправки ответа метод `yii\web\Response::send()` выполняет следующие шаги:

1. Иницируется событие `yii\web\Response::EVENT_BEFORE_SEND`.
2. Для форматирования данных ответа в содержимое ответа вызывается метод `yii\web\Response::prepare()`.
3. Иницируется событие `yii\web\Response::EVENT_AFTER_PREPARE`.
4. Для отправки зарегистрированных HTTP-заголовков вызывается метод `yii\web\Response::sendHeaders()`.

⁶http://tn123.org/mod_xsendfile

⁷<http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file>

⁸<http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file>

⁹<http://wiki.nginx.org/XSendfile>

¹⁰http://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

5. Для отправки тела ответа вызывается метод `yii\web\Response::sendContent()`.
6. Иницируется событие `yii\web\Response::EVENT_AFTER_SEND`.

Повторный вызов `yii\web\Response::send()` игнорируется. Это означает, что если ответ уже отправлен, то к нему уже ничего не добавить.

Как видно, метод `yii\web\Response::send()` иницирует несколько полезных событий. Реагируя на эти события, можно настраивать или декорировать ответ.

Error: not existing file: runtime-sessions-cookies.md

4.5 Работа с URL

Замечание: раздел находится в разработке.

Концепция работы с URL в Yii довольно проста. Предполагается, что в приложении используются внутренние маршруты и параметры вместо жестко заданных URL. Тогда фреймворк сам преобразует маршруты в URL и обратно, в соответствии с конфигурацией URL менеджера. Такой подход позволяет изменять вид URL на всем сайте, редактируя единственный конфигурационный файл не трогая код самого приложения.

4.5.1 Внутренние маршруты

При создании приложения с помощью Yii, вы будете работать с внутренними маршрутами, которые также часто называются маршрутами с параметрами. Каждому контроллеру и действию соответствует внутренний маршрут, например `site/index` или `user/create`. В первом примере, `site` - это ID контроллера а `create` это ID действия. Если контроллер находится в модуле, то перед его ID контроллера в маршруте ставится ID модуля, например так: `blog/post/index`. Здесь `blog` - это ID модуля, а `post` и `index` - ID контроллера и действия соответственно.

4.5.2 Создание URL

Самое важное при работе с URL = всегда создавать их через URL manager. Это встроенный компонент приложения, к которому можно обращаться по имени `urlManager` как в консольном, так и в веб-приложении, таким образом: `\Yii::$app->urlManager`. Через этот компонент доступны следующие методы создания URL: - `createUrl($params)` - `createAbsoluteUrl($params, $schema = null)`

Метод `createUrl` создает относительный от корня приложения URL, например `index.php/site/index`. Метод `createAbsoluteUrl` создает абсолютный URL, добавляя к относительному протокол и имя хоста: `http://www.example.com/index.php/site/index`. `createUrl` больше подходит для URL внутри самого приложения, тогда как `createAbsoluteUrl` - для создания ссылок на внешние ресурсы и для них, для решения задач отправки почты, генерации RSS и т.п.

Примеры:

```
echo \Yii::$app->urlManager->createUrl(['site/page', 'id' => 'about']);
// /index.php/site/page/id/about/
echo \Yii::$app->urlManager->createUrl(['date-time/fast-forward', 'id' =>
105])
// /index.php?r=date-time/fast-forward&id=105
echo \Yii::$app->urlManager->createAbsoluteUrl('blog/post/index');
// http://www.example.com/index.php/blog/post/index/
```

Точный формат URL зависит от конфигурации URL manager. В другой конфигурации примеры выше могут выводить:

- /site/page/id/about/
- /index.php?r=site/page&id=about
- /index.php?r=date-time/fast-forward&id=105
- /index.php/date-time/fast-forward?id=105
- <http://www.example.com/blog/post/index/>
- <http://www.example.com/index.php?r=blog/post/index>

Чтобы упростить создание URL рекомендуется пользоваться встроенным хелпером `yii\helpers\Url`. Покажем, как работает хелпер на примерах. Предположим, что мы находимся на `/index.php?r=management/default/users&id=10`. Тогда:

```
use yii\helpers\Url;

// текущий активный маршрут
// /index.php?r=management/default/users
echo Url::to('');

// тот же контроллер, другое действие
// /index.php?r=management/default/page&id=contact
echo Url::toRoute(['page', 'id' => 'contact']);

// тот же модуль, другие контроллер и действие
// /index.php?r=management/post/index
echo Url::toRoute('post/index');

// абсолютный маршрут вне зависимости от того, в каком контроллере
// происходит вызов
// /index.php?r=site/index
echo Url::toRoute('/site/index');

// url для регистрозависимого действия hiTech текущего контроллера
// /index.php?r=management/default/hi-tech
echo Url::toRoute('hi-tech');

// url для регистрозависимого контроллера, 'DateTimeController::
//   actionFastForward'
// /index.php?r=date-time/fast-forward&id=105
echo Url::toRoute(['/date-time/fast-forward', 'id' => 105]);

// получение URL через alias
// http://google.com/
Yii::setAlias('@google', 'http://google.com/');
echo Url::to('@google');

// получение URL домой""
// /index.php?r=site/index
echo Url::home();

Url::remember(); // сохранить URL, чтобы использовать его позже
```

```
Url::previous(); // получить ранее сохраненный URL
```

> **Совет:** чтобы сгенерировать URL с хэштегом, например `/index.php?r=site/page&id=100#title`, укажите параметр `#` в хелпере таким образом: `Url::to(['post/read', 'id' => 100, '#' => 'title'])`.

Также существует метод `Url::canonical()`, который позволяет создать канонический URL¹¹ (статья на англ., перевода пока нет) для текущего действия. Этот метод при создании игнорирует все параметры действия кроме тех, которые были переданы как аргументы. Пример:

```
namespace app\controllers;

use yii\web\Controller;
use yii\helpers\Url;

class CanonicalController extends Controller
{
    public function actionTest($page)
    {
        echo Url::canonical();
    }
}
```

При этом при текущем URL `/index.php?r=canonical/test&page=hello&number=42` канонический URL будет `/index.php?r=canonical/test&page=hello`.

4.5.3 Модификация URL

По умолчанию Yii использует url формата query string, например `/index.php?r=news/view&id=100`. Чтобы сделать более человеко-понятные URL¹², например для улучшения их читабельности, необходимо сконфигурировать компонент `urlManager` в конфигурационном файле приложения. Придав параметру `enablePrettyUrl` значение `true`, мы тем самым получим URL такого вида: `/index.php/news/view?id=100`, а выставив параметр `showScriptName' => false` мы исключим `index.php` из URL. Вот фрагмент конфигурационного файла:

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
        ],
    ],
];
```

¹¹https://en.wikipedia.org/wiki/Canonical_link_element

¹²https://ru.wikipedia.org/wiki/%D0%A7%D0%9F%D0%A3_%28%D0%98%D0%BD%D1%82%D0%B5%D1%80%D0%BD%D0%B5%D1%82%29

Обратите внимание, что конфигурация с `'showScriptName' => false` будет работать только, если веб сервер был должным образом сконфигурирован. Смотрите раздел `installation` (ссылка не работает).

Именованные параметры

Правило может быть связано с несколькими `GET` параметрами. Эти `GET` параметры появляются в паттерне правила как специальные управляющие конструкции в таком формате:

```
ИмяПараметраПаттернПараметра
<:>
```

`ИмяПараметра` это имя `GET` параметра, а необязательный `ПаттернПараметра` - это регулярное выражение, которое используется для того, чтобы найти определенный `GET` параметр. В случае, если `ПаттернПараметра` не указан, это значит, что значение параметра это любая последовательность символов, кроме `/`. Правила используются и при парсинге URL, и при их создании; при создании URL параметры будут заменены на соответствующие значения, а при парсинге - `GET` параметры с соответствующими именами получают соответствующие значения из URL.

Приведем несколько примеров, чтобы пояснить, как работают правила. Предположим, что наш массив правил содержит три правила:

```
[
  'posts'=>'post/list',
  'post/<id:\d+>'=>'post/read',
  'post/<year:\d{4}>/<title>'=>'post/read',
]
```

- Вызов `Url::toRoute('post/list')` генерирует `/index.php/posts`. Применяется первое правило.
- Вызов `Url::toRoute(['post/read', 'id' => 100])` генерирует `/index.php/post/100`. Применяется второе правило.
- Вызов `Url::toRoute(['post/read', 'year' => 2008, 'title' => 'a sample post'])` генерирует `/index.php/post/2008/a%20sample%20post`. Применяется третье правило.
- Вызов `Url::toRoute('post/read')` генерируется `/index.php/post/read`. Не применяется ни одного правила, вместо этого URL формируется по алгоритму по умолчанию.

Таким образом, при использовании `createUrl` для генерации URL, маршруты и переданные методу `GET` параметры используются для принятия решения о том, какое правило применить. Конкретное правило используется для генерации URL, если каждый параметр, указанный в правиле, может быть найден среди `GET` параметров, переданных `createUrl`, и сам маршрут, указанный в правиле, соответствует маршруту, переданному в качестве параметра.

Если параметров, переданных `Url::toRoute` больше, чем указано в правиле, то дополнительные параметры будут указаны в формате строки запроса. Например, при вызове `Url::toRoute(['post/read', 'id' => 100, 'year' => 2008])`, мы получим `/index.php/post/100?year=2008`.

Как было сказано ранее, другое назначение правил - парсинг URL. Это процесс, обратный их созданию. Например, когда пользователь запрашивает `/index.php/post/100`, будет применено второе правило из примера выше, которое извлечет маршрут `post/read` и GET параметр `['id' => 100]` (который можно получить так: `Yii::$app->request->get('id')`).

Параметры в маршрутах

Мы можем обращаться к именованным параметрам в маршрутной части правила. Это позволяет нескольким маршрутам подпадать под правило, в соответствии с переданными параметрами. Это также может помочь минимизировать количество правил URL в приложении, тем самым улучшив его общую производительность.

Вот пример, показывающий как использовать маршруты с именованными параметрами.

```
[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
    => '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/read',
    '<controller:(post|comment)>s' => '<controller>/list',
]
```

В этом примере, мы использовали два именованных параметра в маршрутной части правил: `controller` и `action`. Первый параметр подходит в случае, если `controller ID` - это `post` или `comment`, а последний - в случае, если `action ID` - `create`, `update` или `delete`. Имена параметров могут быть разными, но не должны совпадать по имени с GET-параметрами, которые могут появляться в URL.

Используя эти правила, URL `/index.php/post/123/create` после парсинга будет иметь маршрут `post/create` с GET параметром `id=123`. А с маршрутом `comment/list` и GET параметром `page=2` можно создать URL `/index.php/comments?page=2`.

Параметры в имени хоста

Существует возможность включать имена хостов в правила парсинга и создания URL. Например, может возникнуть необходимость получить часть имени хоста в качестве GET параметра, чтобы работать с поддоменами. Например, URL `http://admin.example.com/en/profile` может быть разобран на GET параметры `user=admin` и `lang=en`. Аналогично, правила с именами хостов могут использоваться для создания URL.

Чтобы использовать параметры в имени хоста, просто объявите правило с именем хоста, например,

```
[  
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',  
]
```

В этом примере первый сегмент имени хоста обрабатывается как имя пользователя (user), тогда как первый сегмент пути обрабатывается как параметр языка (lang). Правило относится к маршруту `user/profile`.

Обратите внимание, что атрибут `yii\web\UrlManager::$showScriptName` не оказывает действия на правила, в которых используются параметры в имени хоста.

Также учтите, что любое правило с параметрами в имени хоста не должно содержать подпапку, если приложение размещено в подпапке web рута. Например, если приложение находится в `http://www.example.com/sandbox/blog`, тогда нам все равно в правиле нужно его прописать без `sandbox/blog`.

Добавка суффикса URL

```
<?php  
return [  
    // ...  
    'components' => [  
        'urlManager' => [  
            'suffix' => '.html',  
        ],  
    ],  
];
```

Обработка REST запросов

TBD: - RESTful маршрутизация: `yii\filters\VerbFilter`, `yii\web\UrlManager::$rules` - Json API: - response: `yii\web\Response::$format` - request: `yii\web\Request::$parsers`, `yii\web\JsonParser`

4.5.4 Парсинг URL

Помимо создания URL Yii также умеет парсить URL, получая на выходе маршруты и параметры.

Строгий парсинг URL

По умолчанию если для URL не подошло ни одного заданного правила и URL соответствует стандартному формату, как, например `/site/page`, Yii делает попытку запустить указанный action соответствующего контроллера. Можно запретить обрабатывать URL стандартного формата,

тогда, если URL не соответствует ни одному правилу, будет выброшена 404 ошибка.

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'enableStrictParsing' => true,
        ],
    ],
];
```

4.5.5 Создание классов для произвольных правил

`yii\web\UrlRule` класс используется для парсинга URL на параметры и создания URL по параметрам. Обычное решение для правил подойдет для большинства проектов, но бывают ситуации, когда лучшим выбором будет использования своего класса для задания правил. Например, на сайте автомобильного дилера могут быть URL типа `ПроизводительМодель//`, где и `Производитель` и `Модель` хранятся в таблице БД. Обычное правило не сработает, т.к. оно основывается на статически заданных регулярных выражений, и извлечение информации из БД в нем не предусмотрено.

Мы можем создать новый класс для правил URL, унаследовав его от `yii\web\UrlRule` и используя его в одном или нескольких URL правил. Ниже - реализация для вышеописанного примера с сайтом автомобильного дилера. Вот указание произвольного правила в конфигурации приложения:

```
// ...
'components' => [
    'urlManager' => [
        'rules' => [
            '<action:(login|logout|about)>' => 'site/<action>',

            // ...

            ['class' => 'app\components\CarUrlRule', 'connectionID' => 'db',
             /* ... */],
        ],
    ],
];
```

В примере мы используем произвольное URL правило `CarUrlRule` для обработки URL формата `ПроизводительМодель//`.

Код самого класса может быть примерно таким:

```
namespace app\components;

use yii\web\UrlRule;
```

```

class CarUrlRule extends UrlRule
{
    public $connectionID = 'db';

    public function init()
    {
        if ($this->name === null) {
            $this->name = __CLASS__;
        }
    }

    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; // это правило не подходит
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\\w+)/?(\\w+)?$%', $pathInfo, $matches)) {
            // check $matches[1] and $matches[3] to see
            // if they match a manufacturer and a model in the database
            // If so, set $params['manufacturer'] and/or $params['model']
            // and return ['car/index', $params]
        }
        return false; // это правило не подходит
    }
}

```

Кроме использования произвольных классов URL в случаях, подобных примеру выше, их также можно использовать для других целей. Например, мы можем написать класс правила для логгирования парсинга URL и создания запросов. Это может быть полезно на этапе разработки. Мы также можем написать класс правила чтобы показывать особую 404 страницу в случае, если другие правила не подошли для обработки текущего запроса. В этом случае произвольное правило должно стоять последним.

4.6 Обработка ошибок

В состав Yii входит встроенный **обработчик ошибок**, делающий работу с ошибками гораздо более приятным занятием. А именно:

- Все нефатальные ошибки PHP (то есть warning, notice) конверти-

- руются в исключения, которые можно перехватывать.
- Исключения и фатальные ошибки PHP отображаются в режиме отладки с детальным стеком вызовов и исходным кодом.
- Можно использовать для отображения ошибок действие контроллера.
- Поддерживаются различные форматы ответа.

По умолчанию **обработчик ошибок** включен. Вы можете выключить его объявив константу `YII_ENABLE_ERROR_HANDLER` со значением `false` во входном скрипте вашего приложения.

4.6.1 Использование обработчика ошибок

Обработчик ошибок регистрируется в качестве компонента приложения с именем `errorHandler`. Вы можете настраивать его следующим образом:

```
return [  
    'components' => [  
        'errorHandler' => [  
            'maxSourceLines' => 20,  
        ],  
    ],  
];
```

С приведённой выше конфигурацией на странице ошибки будет отображаться до 20 строк исходного кода.

Как уже было упомянуто, обработчик ошибок конвертирует все нефатальные ошибки PHP в перехватываемые исключения. Это означает что можно поступать с ошибками следующим образом:

```
use Yii;  
use yii\base\ErrorException;  
  
try {  
    10/0;  
} catch (ErrorException $e) {  
    Yii::warning("Деление на ноль.");  
}  
  
// можно продолжать выполнение
```

Если вам необходимо показать пользователю страницу с ошибкой, говорящей ему о том, что его запрос не верен или не должен был быть сделан, вы можете выкинуть **исключение HTTP**, такое как `yii\web\NotFoundHttpException`. Обработчик ошибок корректно выставит статус код HTTP для ответа и использует подходящий вид страницы ошибки.

```
use yii\web\NotFoundHttpException;  
  
throw new NotFoundHttpException();
```

4.6.2 Настройка отображения ошибок

Обработчик ошибок меняет отображение ошибок в зависимости от значения константы `YII_DEBUG`. При `YII_DEBUG` равной `true` (режим отладки), обработчик ошибок будет отображать для облегчения отладки детальный стек вызовов и исходный код. При `YII_DEBUG` равной `false` отображается только сообщение об ошибке, тем самым не позволяя получить информацию о внутренностях приложения.

Информация: Если исключение является наследником `yii\base\UserException`, стек вызовов не отображается вне зависимости от значения `YII_DEBUG` так как такие исключения считаются ошибками пользователя и исправлять что-либо разработчику не требуется.

По умолчанию обработчик ошибок показывает ошибки используя два представления:

- `@yii/views/errorHandler/error.php`: используется для отображения ошибок БЕЗ стека вызовов. При `YII_DEBUG` равной `false` используется только это представление.
- `@yii/views/errorHandler/exception.php`: используется для отображения ошибок СО стеком вызовов.

Вы можете настроить свойства `errorView` и `exceptionView` для того, чтобы использовать свои представления.

Использование действий для отображения ошибок

Лучшим способом изменения отображения ошибок является использование действий путём конфигурирования свойства `errorAction` компонента `errorHandler`:

```
// ...
'components' => [
    // ...
    'errorHandler' => [
        'errorAction' => 'site/error',
    ],
]
```

Свойство `errorAction` принимает маршрут действия. Конфигурация выше означает, что для отображения ошибки без стека вызовов будет использовано действие `site/error`.

Само действие можно реализовать следующим образом:

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
```

```
class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}

```

“Приведённый выше код задаёт действие ‘error’ используя класс `[[yii\web\ErrorAction]]`, который рендерит ошибку используя отображение ‘error’. Вместо использования `[[yii\web\ErrorAction]]` вы можете создать действие ‘error’ как обычный метод:

```
““php
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}

```

Вы должны создать файл представления `views/site/error.php`. В этом файле, если используется `yii\web\ErrorAction`, вам доступны следующие переменные:

- **name**: имя ошибки;
- **message**: текст ошибки;
- **exception**: объект исключения, из которого можно получить дополнительную информацию, такую как статус HTTP, код ошибки, стек вызовов и т.д.

Информация: Если вы используете шаблоны приложения `basic` или `advanced`, действие `error` и файл представления уже созданы за вас.

Изменение формата ответа

Обработчик ошибок отображает ошибки в соответствии с выбранным форматом ответа. Если **формат ответа** задан как `html`, будут использоваться представления для ошибок и исключений, как описывалось ранее. Для остальных форматов ответа обработчик ошибок присваивает массив данных, представляющий ошибку свойству `yii\web\Response::`

`$data`. Оно далее конвертируется в необходимый формат. Например, если используется формат ответа `json`, вы получите подобный ответ:

```
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
}
```

Изменить формат можно в обработчике события `beforeSend` компонента `response` в конфигурации приложения:

```
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;
                }
            },
        ],
    ],
];
```

Приведённый код изменит формат ответа на подобный:

```
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
    }
}
```

4.7 Логгирование

Раздел находится в разработке

В Yii встроен гибкий и расширяемый логгер, который способен обрабатывать сообщения в соответствии с их уровнем важности и типом. С его помощью также можно фильтровать сообщения по разным критериям и пересылать их в файлы, email, в дебаггер и т.п.

4.7.1 Основы логгирования

В самом простом виде логгирование - это просто вызов метода, как в примере ниже:

```
\Yii::info('Привет, я - тестовое сообщение лога');
```

Вы можете логгировать как данные строкового типа, так и более сложные структуры данных, такие как массивы и объекты. Если логируемые данные - не строка, обработчики логов по умолчанию сериализуют значение, используя `yii\helpers\VarDumper::export()`.

Категории сообщений

Вы можете указать категорию сообщения, чтобы разделить сообщения разного типа в дальнейшем, и по разному их обработать. Категория сообщения передается вторым аргументом методов логгирования. По умолчанию присваивается категория `application`.

Уровни важности

Доступно несколько уровней важности и соответствующим им методов логгирования:

- `Yii::trace()` в основном используется в разработке, чтобы логгировать прогресс выполнения кода. Заметьте, что он работает только в режиме разработки, когда константа `YII_DEBUG` имеет значение `true`.
- `Yii::error()` используется в случае невосстановимой ошибки.
- `Yii::warning()` используется, когда произошла ошибка, но исполнение может быть продолжено.
- `Yii::info()` используется, чтобы фиксировать информацию о важных событиях, таких как логин администратора.

4.7.2 Цели сообщений

Когда вызывается один из логирующих методов, сообщение передается компоненту `yii\log\Logger`, доступному через `Yii::getLogger()`. Логгер

хранит сообщения в памяти, и когда сообщений достаточно для отправки, или когда заканчивается текущий запрос, отправляет сообщения по целям назначения, таким как файл или email.

Вы можете конфигурировать цели сообщений таким образом:

```
[
    'bootstrap' => ['log'], // убеждаемся, что логгер загружается до
    запуска приложения
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                    'levels' => ['trace', 'info'],
                    'categories' => ['yii\*'],
                ],
                'email' => [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error', 'warning'],
                    'message' => [
                        'to' => ['admin@example.com', 'developer@example.com'],
                        'subject' => 'Новое сообщение логгера example.com',
                    ],
                ],
            ],
        ],
    ],
],
```

В конфигурации выше мы назначает две цели: `file` и `email`. В обоих случаях мы фильтруем сообщения по важности, а в случае с записью в файл еще и по категории. `yii*` значит все категории, начинающиеся с `yii\`.

Каждая цель может иметь имя, и к ней можно обращаться через `yii\log\Logger::targets` следующим образом:

```
Yii::$app->log->targets['file']->enabled = false;
```

Когда приложение заканчивает работу, или когда достигнут предел количества сообщений `flushInterval`, логгер вызовет метод `flush()` для отправки сообщений по маршрутам.

Обратите внимание, что в примере выше мы добавили компонент `log` в список автозагрузки компонентов, чтобы он инициализировался в самом начале жизненного цикла приложения, чтобы, в свою очередь, убедиться, что логгирование будет происходить с самого начала приложения.

4.7.3 Профилирование

Профилирование - особый тип сообщений, который может быть использован для замера времени, необходимого определенным блокам кода для отработки, чтобы выяснить, где можно улучшить производительность.

Чтобы пользоваться профилированием нужно понять, какие блоки кода нужно профилировать. Затем нужно отметить начало и конец каждого блока, вызывая нижеследующие методы:

```
\Yii::beginProfile('myBenchmark');блок  
... кода для профилирования...  
\Yii::endProfile('myBenchmark');
```

где `myBenchmark` - уникальный идентификатор блока кода.

Заметьте, что блоки кода должны быть правильно вложены друг в друга. Посмотрите пример ниже:

```
\Yii::beginProfile('block1');  
    // код блока 1  
    \Yii::beginProfile('block2');  
        // код блока два, который входит в блок один  
    \Yii::endProfile('block2');  
\Yii::endProfile('block1');
```

Результаты профилирования можно отображать в дебаггере.

Глава 5

Основные понятия

5.1 Компоненты

Компоненты — это главные строительные блоки приложений основанных на Yii. Компоненты наследуются от класса `yii\base\Component` или его наследников. Три главные возможности, которые компоненты предоставляют для других классов:

- Свойства.
- События.
- Поведения.

Как по отдельности, так и вместе, эти возможности делают классы Yii более простыми в настройке и использовании. Например, пользовательские компоненты, включающие в себя **виджет выбора даты**, могут быть использованы в представлении для генерации интерактивных элементов выбора даты:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ru',
    'name'     => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

Свойства виджета легко доступны для записи потому, что его класс унаследован от класса `yii\base\Component`.

Компоненты — очень мощный инструмент. Но в то же время они немного тяжелее обычных объектов, потому что на поддержку событий и поведений тратится дополнительная память и процессорное время. Если ваши компоненты не нуждаются в этих двух возможностях, вам стоит унаследовать их от `yii\base\Object`, а не от `yii\base\Component`. Поступив так, вы сделаете ваши компоненты такими же эффективными,

как и обычные PHP объекты, но с поддержкой свойств.

При наследовании ваших классов от `yii\base\Component` или `yii\base\Object`, рекомендуется следовать некоторым соглашениям:

- Если вы переопределяете конструктор, то добавьте *последним* аргументом параметр `$config` и затем передайте его в конструктор предка.
- Всегда вызывайте конструктор предка *в конце* вашего переопределенного конструктора.
- Если вы переопределяете метод `yii\base\Object::init()`, убедитесь, что вы вызываете родительскую реализацию этого метода *в начале* вашего метода `init()`.

Пример:

```
<?php

namespace yii\components\MyClass;

use yii\base\Object;

class MyClass extends Object
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... инициализация происходит перед тем, как будет применена
        конфигурация.

        parent::__construct($config);
    }

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

        // ... инициализация происходит после того, как была применена
        конфигурация.
    }
}
```

Следуя этому руководству вы позволите настраивать ваш компонент при создании. Например:

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// альтернативный способ
$component = \Yii::createObject([
    'class' => MyClass::className(),
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

Информация: Способ инициализации через вызов `Yii::createObject()` выглядит более сложным. Но в то же время он более мощный из-за того, что он реализован на самом верху контейнера внедрения зависимостей.

Жизненный цикл объектов класса `yii\base\Object` содержит следующие этапы:

1. Предварительная инициализация в конструкторе. Здесь вы можете установить значения свойств по умолчанию.
2. Конфигурация объекта с помощью `$config`. Во время конфигурации могут быть перезаписаны значения свойств по умолчанию, установленные в конструкторе.
3. Конфигурация после инициализации в методе `init()`. Вы можете переопределить этот метод, для проверки готовности объекта и нормализации свойств.
4. Вызов методов объекта.

Первые три шага всегда выполняются из конструктора объекта. Это значит, что если вы получите экземпляр объекта, он уже будет проинициализирован и готов к работе.

5.2 Свойства

В PHP, переменные-члены класса называются *свойства*. Эти переменные являются частью объявления класса и используются для хранения состояния объектов этого класса (т.е. именно этим отличается один экземпляр класса от другого). На практике вам часто придётся производить чтение и запись свойств особым образом. Например, вам может понадобиться обрезать строку при её записи в поле `label`. Для этого вы можете использовать следующий код:

```
$object->label = trim($label);
```

Недостатком приведённого выше кода является то, что вам придётся вызывать функцию `trim()` во всех местах, где вы присваиваете значение полю `label`. Если в будущем понадобится производить еще какие-либо действие, например преобразовать первую букву в верхний регистр, вам придётся изменить каждый участок кода, где производится присваивание значения полю `label`. Повторение кода приводит к ошибкам и его необходимо избегать всеми силами.

Что бы решить эту проблему, в Yii был добавлен базовый класс `yii\base\Object` который реализует работу со свойствами через *геттеры* и *сеттеры*. Если вашему классу нужна такая возможность, необходимо унаследовать его от `yii\base\Object` или его потомка.

Информация: Почти все внутренние классы Yii наследуются от `yii\base\Object` или его потомков. Это значит, что всякий раз, когда вы встречаете геттер или сеттер в классах фреймворка, вы можете обращаться к нему как к свойству.

Геттер — это метод, чье название начинается со слова `get`. Имя сеттера начинается со слова `set`. Часть названия после `get` или `set` определяет имя свойства. Например, геттер `getLabel()` и/или сеттер `setLabel()` определяют свойство `label`, как показано в коде ниже:

```
namespace app\components;

use yii\base\Object;

class Foo extend Object
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

    public function setLabel($value)
    {
        $this->_label = trim($value);
    }
}
```

В коде выше геттер и сеттер реализуют свойство `label`, значение которого хранится в `private` свойстве `_label`.

Свойства, определенные с помощью геттеров и сеттеров, можно использовать как обычные свойства класса. Главное отличие в том, что когда происходит чтение такого свойства, вызывается соответствующий геттер, при присвоении значения такому свойству запускается соответствующий сеттер. Например:

```
// Идентично вызову $label = $object->getLabel();
$label = $object->label;

// Идентично вызову $object->setLabel('abc');
$object->label = 'abc';
```

Свойство, для которого объявлен только геттер без сеттера, может использоваться *только для чтения*. Попытка присвоить ему значение вызовет `InvalidCallException`. Точно так же, свойство для которого объявлен только сеттер без геттера может использоваться *только для записи*. Попытка получить его значение так же вызовет исключение. Свойства, предназначенные только для чтения, встречаются не часто.

При определении свойств класса при помощи геттеров и сеттеров нужно помнить о некоторых правилах и ограничениях:

- Имена таких свойств *регистронезависимы*. Таким образом, `$object->label` и `$object->Label` — одно и то же. Это обусловлено тем, что имена методов в PHP регистронезависимы.
- Если имя такого свойства уже используется переменной-членом класса, то последнее будет иметь более высокий приоритет. Например, если в классе `Foo` объявлено свойство `label`, то при вызове `$object->label = 'abc'` будет напрямую изменено значение свойства `label`. А метод `setLabel()` не будет вызван.
- Свойства, объявленные таким образом, не поддерживают модификаторы видимости. Это значит, что объявление геттера или сеттера как `public`, `protected` или `private` никак не скажется на области видимости свойства.
- Свойства могут быть объявлены только с помощью *нестатичных* геттеров и/или сеттеров. Статические методы не будут обрабатываться подобным образом.

Возвращаясь к проблеме необходимости вызова функции `trim()` во всех местах, где присваивается значение свойству `label`, описанной в начале этого руководства, функцию `trim()` теперь необходимо вызывать только один раз — в методе `setLabel()`. При возникновении нового требования о возведение первой буквы в верхний регистр, можно быстро поправить метод `setLabel()` не затрагивая остальной код. Эта правка будет распространяться на все присвоения значения свойству `label`.

5.3 События

События - это механизм, внедряющий элементы собственного кода в существующий код в определенные моменты его исполнения. К событию можно присоединить собственный код, который будет выполняться автоматически при срабатывании события. Например, объект, отвечающий за почту, может инициировать событие `messageSent` при успешной отправке сообщения. При этом если нужно отслеживать успешно отправленные сообщения, достаточно присоединить соответствующий код к событию `messageSent`.

Для работы с событиями Yii использует базовый класс `yii\base\Component`. Если класс должен инициировать события, его нужно унаследовать от `yii\base\Component` или потомка этого класса.

5.3.1 Обработчики событий

Обработчик события - это callback-функция PHP¹, которая выполняется при срабатывании события, к которому она присоединена. Можно использовать следующие callback-функции:

¹<http://www.php.net/manual/ru/language.types.callable.php>

- глобальную функцию PHP, указав строку с именем функции (без скобок), например, `'trim'`;
- метод объекта, указав массив, содержащий строки с именами объекта и метода (без скобок), например, `[$object, 'methodName']`;
- статический метод класса, указав массив, содержащий строки с именами класса и метода (без скобок), например, `['ClassName', 'methodName']`;
- анонимную функцию, например, `function ($event) { ... }`.

Сигнатура обработчика события выглядит следующим образом:

```
function ($event) {
    // $event - это объект класса yii\base\Event или его потомка
}
```

Через параметр `$event` обработчик события может получить следующую информацию о возникшем событии:

- **event name**
- **event sender**: объект, метод `trigger()` которого был вызван
- **custom data**: данные, которые были предоставлены во время присоединения обработчика события (будет описано ниже)

5.3.2 Присоединение обработчиков событий

Обработчики события присоединяются с помощью метода `yii\base\Component::on()`. Например:

```
$foo = new Foo;

// обработчик - глобальная функция
$foo->on(Foo::EVENT_HELLO, 'function_name');

// обработчик - метод объекта
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// обработчик - статический метод класса
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// обработчик - анонимная функция
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // логика обработки события
});
```

Также обработчики событий можно присоединять с помощью конфигураций. Дополнительную информацию см. в разделе Конфигурации.

Присоединяя обработчик события, можно передать дополнительные данные с помощью третьего параметра метода `yii\base\Component::on()`. Эти данные будут доступны в обработчике, когда сработает событие и он будет вызван. Например:

```
// Следующий код выводит "abc" при срабатывании события
```

```
// так как в $event->data содержатся данные, которые переданы в качестве
// третьего аргумента метода "on"
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

5.3.3 Порядок обработки событий

К одному событию можно присоединить несколько обработчиков. При срабатывании события обработчики будут вызываться в том порядке, в котором они присоединялись к событию. Чтобы запретить в обработчике вызов всех следующих за ним обработчиков, необходимо установить свойство `yii\base\Event::$handled` параметра `$event` в `true`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

По умолчанию, новые обработчики присоединяются к концу очереди обработчиков, уже существующей у события. В результате при срабатывании события обработчик выполнится последним. Чтобы обработчик присоединился к началу очереди и запускался первым, при вызове `yii\base\Component::on()` в качестве четвертого параметра `$append` следует передать `false`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4 Инициирование событий

События иницируются при вызове метода `yii\base\Component::trigger()`. Методу нужно передать *имя события*, а при необходимости - объект события, в котором описываются параметры, передаваемые обработчикам событий. Например:

```
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);
    }
}
```

```
}
```

Показанный выше код инициирует событие `hello` при каждом вызове метода `bar()`.

Подсказка: Желательно для обозначения имен событий использовать константы класса. В предыдущем примере константа `EVENT_HELLO` обозначает событие `hello`. У такого подхода три преимущества. Во-первых, исключаются опечатки. Во-вторых, для событий работает автозавершение в различных средах разработки. В третьих, чтобы узнать, какие события поддерживаются классом, достаточно проверить константы, объявленные в нем.

Иногда при инициировании события может понадобиться передать его обработчику дополнительную информацию. Например, объекту, отвечающему за почту, может понадобиться передать обработчику события `messageSent` определенные данные, раскрывающие смысл отправленных почтовых сообщений. Для этого в качестве второго параметра методу `yii\base\Component::trigger()` передается объект события. Объект события должен быть экземпляром класса `yii\base\Event` или его потомка. Например:

```
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)
    {
        // отправка... $message...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}
```

При вызове метода `yii\base\Component::trigger()` будут вызваны все обработчики, присоединенные к указанному событию.

5.3.5 Отсоединение обработчиков событий

Для отсоединения обработчика от события используется метод `yii\base\Component::off()`. Например:

```
// обработчик - глобальная функция
$foo->off(Foo::EVENT_HELLO, 'function_name');

// обработчик - метод объекта
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// обработчик - статический метод класса
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// обработчик - анонимная функция
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

Учтите, что в общем случае отсоединять обработчики - анонимные функции можно только если они где-то сохраняются в момент присоединения к событию. В предыдущем примере предполагается, что анонимная функция сохранена в переменной `$anonymousFunction`.

Чтобы отсоединить ВСЕ обработчики от события, достаточно вызвать `yii\base\Component::off()` без второго параметра:

```
$foo->off(Foo::EVENT_HELLO);
```

5.3.6 Обработчики событий на уровне класса

Во всех предыдущих примерах мы рассматривали присоединение событий *на уровне экземпляров*. Есть случаи, когда необходимо обрабатывать события, которые инициируются *любым* экземпляром класса, а не только конкретным экземпляром. В таком случае присоединять обработчик события к каждому экземпляру класса не нужно. Достаточно присоединить обработчик *на уровне класса*, вызвав статический метод `yii\base\Event::on()`.

Например, объект `Active Record` инициирует событие `EVENT_AFTER_INSERT` после добавления в базу данных новой записи. Чтобы отслеживать записи, добавленные в базу данных *каждым* объектом `Active Record`, можно использовать следующий код:

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;

Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT,
    function ($event) {
        Yii::trace(get_class($event->sender) . ' добавлен');
    });
```

Обработчик будет вызван при срабатывании события `EVENT_AFTER_INSERT` в экземплярах класса `ActiveRecord` или его потомков. В обработчике

можно получить доступ к объекту, который инициировал событие, с помощью свойства `$event->sender`.

При срабатывании события будут в первую очередь вызваны обработчики на уровне экземпляра, а затем - обработчики на уровне класса.

Инициировать событие *на уровне класса* можно с помощью статического метода `yii\base\Event::trigger()`. Событие на уровне класса не связано ни с одним конкретным объектом. В таком случае будут вызваны только обработчики события на уровне класса. Например:

```
use yii\base\Event;

Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {
    echo $event->sender; // выводит "app\models\Foo"
});

Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

Обратите внимание, что в данном случае `$event->sender` ссылается на имя класса, который инициировал событие, а не на его экземпляр.

Примечание: Поскольку обработчики на уровне класса отвечают на события, инициируемые всеми экземплярами этого класса и всех его потомков, их следует использовать с осторожностью, особенно в случае базовых классов низкого уровня, таких как `yii\base\Object`.

Отсоединить обработчик события на уровне класса можно с помощью метода `yii\base\Event::off()`. Например:

```
// отсоединение $handler
Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);

// отсоединяются все обработчики Foo::EVENT_HELLO
Event::off(Foo::className(), Foo::EVENT_HELLO);
```

5.3.7 Глобальные события

Yii поддерживает так называемые *глобальные события*, которые на самом деле основаны на нестандартном использовании описанного выше механизма событий. Для глобальных событий нужен глобально доступный объект-синглетон, например, экземпляр приложения - `application`.

Чтобы создать глобальное событие, отправитель сообщения вызывает метод `trigger()` синглтона, а не свой собственный метод `trigger()`. Аналогичным образом обработчики события также присоединяются к событиям синглтона. Например:

```
use Yii;
use yii\base\Event;
use app\components\Foo;
```

```
Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender); // выводит "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

Преимущество глобальных событий в том, что им не нужен объект, к событию которого бы присоединялся обработчик и объект, с помощью которого бы это событие инициировалось. Вместо этого и для присоединения обработчика, и для инициирования события используется синглтон (например, экземпляр приложения).

Тем не менее, так как пространство имен глобальных событий едино для всего приложения, их имена нельзя назначать бездумно. Например, полезными могут быть искусственные пространства имен ("frontend.mail.sent", "backend.mail.sent").

5.4 Поведения

Поведения (behaviors) — это экземпляры класса `yii\base\Behavior` или класса, унаследованного от него. Поведения, также известные как примеси², позволяют расширять функциональность существующих **компонентов** без необходимости изменения дерева наследования. После прикрепления поведения к компоненту, его методы и свойства “внедряются” в компонент, и становятся доступными так же, как если бы они были объявлены в самом классе компонента. Кроме того, поведение может реагировать на события, создаваемые компонентом, что позволяет тонко настраивать или модифицировать обычное выполнение кода компонента.

5.4.1 Создание поведений

Поведения создаются путем расширения базового класса `yii\base\Behavior` или его наследников. Например,

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;
```

²[http://ru.wikipedia.org/wiki/\T2A\CYRP\T2A\cyrr\T2A\cyri\T2A\cyrn\T2A\cyre\T2A\cyrs\T2A\cyrsftsn_\(\T2A\cyrp\T2A\cyrr\T2A\cyro\T2A\cyrq\T2A\cyrr\T2A\cyra\T2A\cyrn\T2A\cyrn\T2A\cyri\T2A\cyrr\T2A\cyro\T2A\cyrv\T2A\cyra\T2A\cyrn\T2A\cyri\T2A\cyre\)](http://ru.wikipedia.org/wiki/\T2A\CYRP\T2A\cyrr\T2A\cyri\T2A\cyrn\T2A\cyre\T2A\cyrs\T2A\cyrsftsn_(\T2A\cyrp\T2A\cyrr\T2A\cyro\T2A\cyrq\T2A\cyrr\T2A\cyra\T2A\cyrn\T2A\cyrn\T2A\cyri\T2A\cyrr\T2A\cyro\T2A\cyrv\T2A\cyra\T2A\cyrn\T2A\cyri\T2A\cyre))

```
public function getProp2()
{
    return $this->_prop2;
}

public function setProp2($value)
{
    $this->_prop2 = $value;
}

public function foo()
{
    // ...
}
}
```

В приведенном выше примере, объявлен класс поведения `app\components\MyBehavior` содержащий 2 свойства `prop1` и `prop2`, и один метод `foo()`. Обратите внимание, свойство `prop2` объявлено с использованием геттера `getProp2()` и сеттера `setProp2()`. Это возможно, так как `yii\base\Behavior` является дочерним классом для `yii\base\Object`, который предоставляет возможность определения свойств через геттеры и сеттеры.

Так как этот класс является поведением, когда он прикреплен к компоненту, компоненту будут также доступны свойства `prop1` и `prop2`, а также метод `foo()`.

Подсказка: Внутри поведения возможно обращаться к компоненту, к которому оно прикреплено, используя свойство `yii\base\Behavior::$owner`.

5.4.2 Обработка событий компонента

Если поведению требуется реагировать на события компонента, к которому оно прикреплено, то необходимо переопределить метод `yii\base\Behavior::events()`. Например,

```
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)
    {
        // ...
    }
}

```

Метод `events()` должен возвращать список событий и соответствующих им обработчиков. В приведенном выше примере, объявлено событие `EVENT_BEFORE_VALIDATE` и его обработчик `beforeValidate()`. Указать обработчик события, можно одним из следующих способов:

- строка с именем метода текущего поведения, как в примере выше;
- массив, содержащий объект или имя класса, и имя метода, например, `[$object, 'methodName']`;
- анонимная функция.

Функция обработчика события должна выглядеть как показано ниже, где `$event` содержит параметр события. Более детальная информация приведена в разделе События.

```

function ($event) {
}

```

5.4.3 Прикрепление поведений

Прикрепить поведение к компоненту можно как статически, так и динамически. На практике чаще используется статическое прикрепление.

Для того чтобы прикрепить поведение статически, необходимо переопределить метод `behaviors()` компонента, к которому его планируется прикрепить. Метод `behaviors()` должен возвращать список конфигураций поведений. Конфигурация поведения представляет собой имя класса поведения, либо массив его настроек:

```

namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // анонимное поведение, прикрепленное по имени класса
            MyBehavior::className(),

            // именованное поведение, прикрепленное по имени класса
            'myBehavior2' => MyBehavior::className(),

            // анонимное поведение, сконфигурированное с использованием
            массива

```

```

        [
            'class' => MyBehavior::className(),
            'prop1' => 'value1',
            'prop2' => 'value2',
        ],

        // именованное поведение, сконфигурированное с использованием
        массива
        'myBehavior4' => [
            'class' => MyBehavior::className(),
            'prop1' => 'value1',
            'prop2' => 'value2',
        ]
    ];
}
}

```

Вы можете связать имя с поведением, указав его в качестве ключа элемента массива, соответствующего конфигурации поведения. В таком случае, поведение называется *именованным*. В примере выше, два именованных поведения: `myBehavior2` и `myBehavior4`. Если с поведением не связано имя, такое поведение называется *анонимным*.

Для того, чтобы прикрепить поведение динамически, необходимо вызвать метод `yii\base\Component::attachBehavior()` требуемого компонента:

```

use app\components\MyBehavior;

// прикрепляем объект поведения
$component->attachBehavior('myBehavior1', new MyBehavior);

// прикрепляем по имени класса поведения
$component->attachBehavior('myBehavior2', MyBehavior::className());

// прикрепляем используя массив конфигураций
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::className(),
    'prop1' => 'value1',
    'prop2' => 'value2',
]);

```

Использование метода `yii\base\Component::attachBehaviors()` позволяет прикрепить несколько поведения за раз. Например,

```

$component->attachBehaviors([
    'myBehavior1' => new MyBehavior, // именованное поведение
    MyBehavior::className(),        // анонимное поведение
]);

```

Так же, прикрепить поведение к компоненту можно через конфигурацию, как показано ниже:

```

[
    'as myBehavior2' => MyBehavior::className(),

```

```
'as myBehavior3' => [  
    'class' => MyBehavior::className(),  
    'prop1' => 'value1',  
    'prop2' => 'value2',  
],  
]
```

Более детальная информация приведена в разделе Конфигурации.

5.4.4 Использование поведений

Для использования поведения, его необходимо прикрепить к **компоненту** как описано выше. После того, как поведение прикреплено к компоненту, его использование не вызывает сложностей.

Вы можете обращаться к *публичным* переменным или свойствам, объявленным с использованием геттеров и сеттеров в поведении, через компонент, к которому оно прикреплено:

```
// публичное свойство "prop1" объявленное в классе поведения  
echo $component->prop1;  
$component->prop1 = $value;
```

Аналогично, вы можете вызывать *публичные* методы поведения,

```
// публичный метод bar() объявленный в классе поведения  
$component->bar();
```

Обратите внимание, хотя `$component` не имеет свойства `prop1` и метода `foo()`, они могут быть использованы, как будто являются членами этого класса.

В случае, когда два поведения, имеющие свойства или методы с одинаковыми именами, прикреплены к одному компоненту, преимущество будет у поведения, прикрепленного раньше.

Если при прикреплении поведения к компоненту указано имя, можно обращаться к поведению по этому имени, как показано ниже:

```
$behavior = $component->getBehavior('myBehavior');
```

Также можно получить все поведения, прикрепленные к компоненту:

```
$behaviors = $component->getBehaviors();
```

5.4.5 Отвязывание поведений

Чтобы отвязать поведение от компонента, необходимо вызвать метод `yii\base\Component::detachBehavior()`, указав имя, связанное с поведением:

```
$component->detachBehavior('myBehavior1');
```

Так же, возможно отвязать *все* поведения:

```
$component->detachBehaviors();
```

5.4.6 Использование поведения `TimestampBehavior`

В заключении, давайте посмотрим на `yii\behaviors\TimestampBehavior` — поведение, которое позволяет автоматически обновлять атрибуты с метками времени при сохранении `Active Record` моделей.

Для начала, необходимо прикрепить поведение к классу `Active Record`, в котором это необходимо:

```
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'],
                ],
            ],
        ];
    }
}
```

Конфигурация выше описывает следующее:

- при вставке новой записи поведение должно присвоить текущую метку времени атрибутам `created_at` и `updated_at`;
- при обновлении существующей записи поведение должно присвоить текущую метку времени атрибуту `updated_at`.

Теперь, если сохранить объект `User`, то в его атрибуты `created_at` и `updated_at` будут автоматически установлены значения метки времени на момент сохранения записи:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at; // выведет метку времени на момент сохранения записи
```

Поведение `TimestampBehavior` так же содержит полезный метод `touch()`, который устанавливает текущую метку времени указанному атрибуту и сохраняет его в базу данных:

```
$user->touch('login_time');
```


5.4.7 Сравнение с трейтами

Несмотря на то, что поведения схожи с трейтами³ тем, что “внедряют” свои свойства и методы в основной класс, они имеют множество отличий. Они оба имеют свои плюсы и минусы, и, скорее, дополняют друг друга, а не заменяют.

Плюсы поведений

Поведения, как и любые другие классы, поддерживают наследование. Трейты же можно рассматривать как копияейст на уровне языка. Они наследование не поддерживают.

Поведения могут быть прикреплены и отвязаны от компонента динамически, без необходимости модифицирования класса компонента. Для использование трейтов необходимо модифицировать класс.

Поведения, в отличие от трейтов, можно настраивать.

Поведения можно настраивать таким образом, чтобы они реагировали на события компонента.

Конфликты имен свойств и методов поведений, прикрепленных к компоненту, разрешаются на основе порядка их подключения. Конфликты имен, вызванные различными трейтами, требуют ручного переименования конфликтующих свойств или методов.

Плюсы трейтов

Трейты являются гораздо более производительными, чем поведения, которые, являясь объектами, требуют дополнительного времени и памяти.

Многие IDE поддерживают работу с трейтами, так как они являются стандартными конструкциями языка.

5.5 Конфигурации

Конфигурации широко используются в Yii при создании новых объектов или при инициализации уже существующих объектов. Обычно конфигурации включают в себя названия классов создаваемых объектов и список первоначальных значений, которые должны быть присвоены свойствам объекта. Также в конфигурациях можно указать список обработчиков событий объекта, и/или список поведений объекта.

Пример конфигурации подключения к базе данных и дальнейшей инициализации подключения:

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
```

³<http://ru2.php.net/manual/ru/language.oop5.traits.php>

```
'username' => 'root',  
'password' => '',  
'charset' => 'utf8',  
];  
  
$db = Yii::createObject($config);
```

Метод `Yii::createObject()` принимает в качестве аргумента массив с конфигурацией и создаёт объект указанного в них класса. При этом оставшаяся часть конфигурации используется для инициализации свойств, обработчиков событий и поведений объекта.

Если объект уже создан, вы можете использовать `Yii::configure()` для того, чтобы инициализировать свойства объекта массивом с конфигурацией:

```
Yii::configure($object, $config);
```

Обратите внимание, что в этом случае массив с конфигурацией не должен содержать ключ `class`.

5.5.1 Формат конфигурации

Формат конфигурации выглядит следующим образом:

```
[  
    'class' => 'ClassName',  
    'propertyName' => 'propertyValue',  
    'on eventName' => $eventHandler,  
    'as behaviorName' => $behaviorConfig,  
]
```

где

- Элемент `class` указывает абсолютное имя класса создаваемого объекта.
- Элементы `propertyName` указывают первоначальные значения свойств создаваемого объекта. Ключи являются именами свойств создаваемого объекта, а значения — начальными значениями свойств создаваемого объекта. Таким способом могут быть установлены только публичные переменные объекта и его свойства, созданные через геттеры и сеттеры.
- Элементы `on eventName` указывают на то, какие обработчики должны быть прикреплены к событиям объекта. Обратите внимание, что ключи массива начинаются с `on`. Чтобы узнать весь список поддерживаемых видов обработчиков событий обратитесь в раздел события
- Элементы `as behaviorName` указывают на то, какие поведения должны быть внедрены в объект. Обратите внимание, что ключи массива начинаются с `as`; а `$behaviorConfig` представляет собой конфигурацию для создания поведения, похожую на все остальные конфигурации.

Пример конфигурации с установкой первоначальных значений свойств объекта, обработчика событий и поведения:

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... начальные значения свойств ...
    ],
]
```

5.5.2 Использование конфигурации

Конфигурации повсеместно используются в Yii. В самом начале данной главы мы узнали как создать объект с необходимыми параметрами используя метод `Yii::createObject()`. В данном разделе речь пойдет о конфигурации приложения и конфигурациях виджетов — двух основных способов использования конфигурации.

Конфигурация приложения

Конфигурация приложения, пожалуй, самая сложная из используемых в фреймворке. Причина в том, что класс `application` содержит большое количество конфигурируемых свойств и событий. Более того, свойство приложения `components` может принимать массив с конфигурацией для создания компонентов, регистрируемых на уровне приложения. Пример конфигурации приложения для шаблона приложения `basic`.

```
$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',
    ],
],
];

```

Ключ `class` в данной конфигурации не указывается. Причина в том, что класс вызывается по полному имени во входном скрипте:

```
(new yii\web\Application($config))->run();
```

За более подробной документацией о настройках свойства `components` в конфигурации приложения обратитесь к главам приложения и Service Locator.

Конфигурации виджетов

При использовании виджетов часто возникает необходимость изменить параметры виджета с помощью конфигурации. Для создания виджета можно использовать два метода: `yii\base\Widget::widget()` и `yii\base\Widget::beginWidget()`. Оба метода принимают конфигурацию в виде PHP массива:

```

use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'Products', 'url' => ['product/index']],
        ['label' => 'Login', 'url' => ['site/login'], 'visible' => Yii::$app
        ->user->isGuest],
    ],
]);

```

Данный код создает виджет `Menu` и устанавливает параметр виджета `activeItems` в значение `false`. Также устанавливается параметр `items`, состоящий из элементов меню.

Обратите внимание что параметр `class` НЕ передается, так как полное имя уже указано.

5.5.3 Конфигурационные файлы

Если конфигурация очень сложная, то её, как правило, разделяют по нескольким PHP файлам. Такие файлы называют *Конфигурационными файлами*. Конфигурационный файл возвращает массив PHP явля-

ющийся конфигурацией. Например, конфигурацию приложения можно хранить в отдельном файле `web.php`, как показано ниже:

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
    'components' => require(__DIR__ . '/components.php'),
];
```

Параметр `components` также имеет сложную конфигурацию, поэтому можно его хранить в файле `components.php` и подключать в файл `web.php` используя `require` как и показано выше. Содержимое файла `components.php`:

```
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',
    ],
];
```

Чтобы получить конфигурацию, хранящуюся в файле, достаточно подключить файл с помощью `require`:

```
$config = require('path/to/web.php');
(new yii\web\Application($config))->run();
```

5.5.4 Значения конфигурации по умолчанию

Метод `Yii::createObject()` реализован с использованием `dependency injection container`. Это позволяет задавать так называемые *значения конфигурации по умолчанию*, которые будут применены ко ВСЕМ экземплярам классов во время их инициализации методом `Yii::createObject()`. Значения конфигурации по умолчанию указываются с помощью метода `Yii::$container->set()` на этапе предварительной загрузки.

Например, если мы хотим изменить виджет `yii\widgets\LinkPager` так, чтобы все виджеты данного вида показывали максимум 5 кнопок на странице вместо 10 (как это установлено изначально), можно использовать следующий код:

```
\Yii::$container->set('yii\widgets\LinkPager', [  
    'maxButtonCount' => 5,  
]);
```

Без использования значений конфигурации по умолчанию, при использовании `LinkPager`, вам пришлось бы каждый раз задавать значение `maxButtonCount`.

5.5.5 Константы окружения

Конфигурации могут различаться в зависимости от режима, в котором происходит запуск приложения. Например, в окружении разработчика (`development`) вы используете базу данных `mydb_dev`, а в эксплуатационном (`production`) окружении базу данных `mydb_prod`. Для упрощения смены окружений в Yii существует константа `YII_ENV`. Вы можете указать её во входном скрипте своего приложения:

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

`YII_ENV` может принимать следующие значения:

- **prod**: окружение `production`, т.е. эксплуатационный режим сервера. Константа `YII_ENV_PROD` установлена в `true`. Значение по умолчанию.
- **dev**: окружение `development`, т.е. режим для разработки. Константа `YII_ENV_DEV` установлена в `true`.
- **test**: окружение `testing`, т.е. режим для тестирования. Константа `YII_ENV_TEST` установлена в `true`.

Используя эти константы, вы можете задать в конфигурации значения параметров зависящие от текущего окружения. Например, чтобы включить отладочную панель и отладчик в режиме разработки, вы можете использовать следующий код в конфигурации приложения:

```
$config = [...];  
  
if (YII_ENV_DEV) {  
    // значения параметров конфигурации для окружения разработки 'dev'  
    $config['bootstrap'][] = 'debug';  
    $config['modules']['debug'] = 'yii\debug\Module';  
}  
  
return $config;
```

5.6 Псевдонимы

Псевдонимы используются для обозначения путей к файлам или URL адресов и помогают избежать использования абсолютных путей или URL в коде. Для того, чтобы не перепутать псевдоним с обычным путём к файлу или URL, он должен начинаться с `@`. В Yii имеется множество заранее определённых псевдонимов. Например, `@yii` указывает на директорию, в которую был установлен Yii framework, а `@web` можно использовать для получения базового URL текущего приложения.

5.6.1 Создание псевдонимов

Для создания псевдонима пути к файлу или URL используется метод `Yii::setAlias()`:

```
// псевдоним пути к файлу
Yii::setAlias('@foo', '/path/to/foo');

// псевдоним URL
Yii::setAlias('@bar', 'http://www.example.com');
```

Примечание: псевдоним пути к файлу или URL *не* обязательно указывает на существующий файл или ресурс.

Используя уже заданный псевдоним, вы можете получить на основе него новый без вызова `Yii::setAlias()`. Сделать это можно, добавив в его конец `/`, за которым следует один или более сегментов пути. Псевдонимы, определённые при помощи `Yii::setAlias()`, являются *корневыми псевдонимами*, в то время как полученные из них называются *производными псевдонимами*. К примеру, `@foo` является корневым псевдонимом, а `@foo/bar/file.php` — производным.

Вы можете задать новый псевдоним, используя ранее созданный псевдоним (не важно, корневой он или производный):

```
Yii::setAlias('@foobar', '@foo/bar');
```

Корневые псевдонимы, как правило, создаются на этапе предварительной загрузки (bootstrapping). Например, вы можете вызвать `Yii::setAlias()` в входном скрипте. Для удобства, в приложении (Application) предусмотрено свойство `aliases`, которое можно задать через конфигурацию приложения:

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'http://www.example.com',
    ],
];
```

5.6.2 Преобразование псевдонимов

Метод `Yii::getAlias()` преобразует корневой псевдоним в путь к файлу или URL, который этот псевдоним представляет. Этот же метод может работать и с производными псевдонимами:

```
echo Yii::getAlias('@foo');           // выведет: /path/to/foo
echo Yii::getAlias('@bar');           // выведет: http://www.example.com
echo Yii::getAlias('@foo/bar/file.php'); // выведет: /path/to/foo/bar/file.php
```

Путь или URL, представленный производным псевдонимом, определяется путём замены в нём части, соответствующей корневому псевдониму, на соответствующий ему путь или URL.

Примечание: Метод `Yii::getAlias()` не проверяет фактического существования получаемого пути или URL.

Корневой псевдоним может содержать знаки `'/'`. При этом метод `Yii::getAlias()` корректно определит, какая часть псевдонима является корневой и верно сформирует путь или URL:

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php'); // выведет: /path/to/foo/test/file.php
Yii::getAlias('@foo/bar/file.php');  // выведет: /path2/bar/file.php
```

Если бы `@foo/bar` не был объявлен корневым псевдонимом, последняя строка вывела бы `/path/to/foo/bar/file.php`.

5.6.3 Использование псевдонимов

Псевдонимы распознаются во многих частях Yii без необходимости предварительно вызывать `Yii::getAlias()` для получения пути или URL. Например, `yii\caching\FileCache::$cachePath` принимает как обычный путь к файлу, так и псевдоним пути благодаря префиксу `@`, который позволяет их различать.

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Для того, чтобы узнать поддерживает ли метод или свойство псевдонимы, обратитесь к документации API.

5.6.4 Заранее определённые псевдонимы

В Yii заранее определены псевдонимы для часто используемых путей к файлам и URL:

- `@yii`: директория, в которой находится файл `BaseYii.php` (директория фреймворка).
- `@app`: базовый путь текущего приложения.
- `@runtime`: директория `runtime` текущего приложения.
- `@vendor`: `[[yii\base\Application::vendorPath|директория vendor Composer]`.
- `@webroot`: вебрут текущего веб приложения (там где находится входной скрипт `index.php`).
- `@web`: базовый URL текущего приложения.

Псевдоним `@yii` задаётся в момент подключения файла `Yii.php` во входном скрипте. Остальные псевдонимы задаются в конструкторе приложения в момент применения конфигурации.

5.6.5 Псевдонимы расширений

Для каждого расширения, устанавливаемого через Composer, автоматически задаётся псевдоним. Его имя соответствует корневому пространству имён расширения в соответствии с его `composer.json`. Псевдоним представляет путь к корневой директории пакета. Например, если вы установите расширение `yiisoft/yii2-jui`, то вам автоматически станет доступен псевдоним `@yii/jui`. Он создаётся на этапе первоначальной загрузки (bootstrapping) примерно так:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.7 Автозагрузка классов

Поиск и подключение файлов классов в Yii реализовано при помощи автозагрузки классов⁴. Фреймворк предоставляет свой быстрый совместимый с PSR-4⁵ автозагрузчик, который устанавливается в момент подключения `Yii.php`.

Примечание: Для простоты повествования, в этом разделе мы будем говорить только об автозагрузке классов. Тем не менее, всё описанное применимо к интерфейсам и трейтам.

5.7.1 Как использовать автозагрузчик Yii

При использовании автозагрузчика классов Yii следует соблюдать два простых правила создания и именования классов:

- Каждый класс должен принадлежать пространству имён⁶ (то есть `foo\bar\MyClass`).

⁴<http://www.php.net/manual/ru/language.oop5.autoload.php>

⁵<https://github.com/php-fig/fig-standards/blob/master/proposed/psr-4-autoloader/psr-4-autoloader.md>

⁶<http://php.net/manual/ru/language.namespaces.php>

- Каждый класс должен находиться в отдельном файле, путь к которому определяется следующим правилом:

```
// $className - это абсолютное имя класса без начального "\"
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) . '.php'
);
```

Например, если абсолютное имя класса `foo\bar\MyClass`, то псевдоним пути данного файла будет `@foo/bar/MyClass.php`. Для того, чтобы данный псевдоним можно было преобразовать в путь к файлу, необходимо чтобы либо `@foo` либо `@foo/bar` являлся корневым псевдонимом.

При использовании шаблона приложения `basic` вы можете хранить свои классы в пространстве имён `app`. В этом случае они будут загружаться автоматически без создания нового псевдонима. Это работает потому как `@app` является заранее определённым псевдонимом и такое имя класса как `app\components\MyClass` в соответствии с описанным выше алгоритмом преобразуется в путь `директорияПриложения/components/MyClass.php`.

В шаблоне приложения `advanced` каждый уровень приложения обладает собственным корневым псевдонимом. Например, для `frontend` корневым псевдонимом является `@frontend`, а для `backend` — `@backend`. Это позволяет разместить классы `frontend` в пространство имён `frontend`, а классы `backend` в пространство имён `backend`. При этом классы будут загружены автоматически.

5.7.2 Карта классов

Автозагрузчик `Yii` поддерживает *карту классов*. Эта возможность позволяет указать путь к файлу для каждого имени класса. При загрузке класса автозагрузчик проверяет наличие класса в карте. Если он там есть, соответствующий файл будет загружен напрямую без каких-либо дополнительных проверок. Это делает автозагрузку очень быстрой. Все классы самого фреймворка загружаются именно этим способом.

Вы можете добавить класс в карту `Yii::$classMap` следующим образом:

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Для указания путей к файлам классов можно использовать псевдонимы. Карту классов необходимо сформировать в процессе первоначальной загрузки так как она должна быть готова до использования классов.

5.7.3 Использование других автозагрузчиков

Так как `Yii` использует `Composer` в качестве менеджера зависимостей, рекомендуется дополнительно установить его автозагрузчик. Если вы используете какие-либо сторонние библиотеки, в которых есть свои автозагрузчики, эти автозагрузчики также необходимо установить.

При использовании дополнительных автозагрузчиков файл `Yii.php` должен быть подключен *после* их установки. Это позволит автозагрузчику Yii первым попробовать загрузить класс. К примеру, приведённый ниже код взят из входного скрипта шаблона приложения basic. Первая строка устанавливает автозагрузчик Composer, а вторая — автозагрузчик Yii:

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

Вы можете использовать автозагрузчик Composer без автозагрузчика Yii. Однако, скорость автозагрузки в этом случае может уменьшиться. Также вам будет необходимо следовать правилам автозагрузчика Composer.

Информация: Если вы не хотите использовать автозагрузчик Yii, создайте свою версию файла `Yii.php` и подключите его в входном скрипте.

5.7.4 Автозагрузка классов расширений

Автозагрузчик Yii может автоматически загружать классы расширений в том случае, если соблюдается единственное правило. Расширение должно правильно описать раздел `'autoload'` в файле `'composer.json'`. Более подробно об этом можно узнать из официальной документации Composer⁷.

Если вы не используете автозагрузчик Yii, то классы расширений могут быть автоматически загружены с помощью автозагрузчика Composer.

5.8 Service Locator

Service Locator является объектом, предоставляющим всевозможные сервисы (или компоненты), которые могут понадобиться приложению. В Service Locator, каждый компонент представлен единственным экземпляром, имеющим уникальный ID. Уникальный идентификатор (ID) может быть использован для получения компонента из Service Locator.

В Yii Service Locator является экземпляром класса `yii\di\ServiceLocator` или его дочернего класса.

Наиболее часто используемый Service Locator в Yii — это объект *приложения*, который можно получить через `\Yii::$app`. Предоставляемые им службы, такие, как компоненты `request`, `response`, `urlManager`, называют *компонентами приложения*. Благодаря Service Locator вы легко можете настроить эти компоненты или даже заменить их собственными реализациями.

⁷<https://getcomposer.org/doc/04-schema.md#autoload>

Помимо объекта приложения, объект каждого модуля также является Service Locator.

При использовании Service Locator первым шагом является регистрация компонентов. Компонент может быть зарегистрирован с помощью метода `yii\di\ServiceLocator::set()`. Следующий код демонстрирует различные способы регистрации компонентов:

```
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// регистрирует "cache", используя имя класса, которое может быть
// использовано для создания компонента.
$locator->set('cache', 'yii\caching\ApcCache');

// регистрирует "db", используя конфигурационный массив, который может быть
// использован для создания компонента.
$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// регистрирует "search", используя анонимную функцию, которая создаёт
// компонент
$locator->set('search', function () {
    return new app\components\SolrService;
});

// регистрирует "pageCache", используя компонент
$locator->set('pageCache', new FileCache);
```

После того, как компонент зарегистрирован, вы можете обращаться к нему по его ID одним из двух следующих способов:

```
$cache = $locator->get('cache');
// или
$cache = $locator->cache;
```

Как видно выше, `yii\di\ServiceLocator` позволяет обратиться к компоненту как к свойству используя его ID. При первом обращении к компоненту, `yii\di\ServiceLocator` создаст новый экземпляр компонента на основе регистрационной информации и вернёт его. При повторном обращении к компоненту Service Locator вернёт тот же экземпляр.

Чтобы проверить, был ли идентификатор компонента уже зарегистрирован, можно использовать `yii\di\ServiceLocator::has()`. Если вы вызовете `yii\di\ServiceLocator::get()` с несуществующим ID, будет выброшено исключение.

Поскольку Service Locator часто используется с конфигурациями, в нём имеется доступное для записи свойство `components`. Это позволяет

настроить и зарегистрировать сразу несколько компонентов. Следующий код демонстрирует конфигурационный массив, который может использоваться для регистрации компонентов “db”, “cache” и “search” в (то есть в приложении):

```
return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');
            // ... дополнительная инициализация ...
            return $solr;
        },
    ],
];
```

Есть альтернативный приведённому выше способ настройки компонента “search”. Вместо анонимной функции, которая отдаёт экземпляр `SolrService` можно использовать статический метод, возвращающий такую анонимную функцию:

```
class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... дополнительная инициализация ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];
```

Это особенно полезно если вы создаёте компонент для Yii, являющийся обёрткой над какой-либо сторонней библиотекой. Подобный приведённому выше статический метод позволяет скрыть от конечного пользователя сложную логику настройки сторонней библиотеки. Пользователю будет достаточно вызвать статический метод.

5.9 Контейнер внедрения зависимостей

Контейнер внедрения зависимостей - это объект, который знает, как создать и настроить экземпляр класса и зависимых от него объектов. Статья Мартина Фаулера⁸ хорошо объясняет, почему контейнер внедрения зависимостей является полезным. Здесь, преимущественно, будет объясняться использование контейнера внедрения зависимостей, предоставляемого в Yii.

5.9.1 Внедрение зависимостей

Yii обеспечивает функционал контейнера внедрения зависимостей через класс `yii\di\Container`. Он поддерживает следующие виды внедрения зависимостей:

- Внедрение зависимости через конструктор.
- Внедрение зависимости через сеттер и свойство.
- Внедрение зависимости через PHP callback.

Внедрение зависимости через конструктор

Контейнер внедрения зависимостей поддерживает внедрение зависимости через конструктор при помощи указания типов для параметров конструктора. Указанные типы сообщают контейнеру, какие классы или интерфейсы зависят от него при создании нового объекта. Контейнер попытается получить экземпляры зависимых классов или интерфейсов, а затем передать их в новый объект через конструктор. Например,

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}

$foo = $container->get('Foo');
// что равносильно следующему:
$bar = new Bar;
$foo = new Foo($bar);
```

Внедрение зависимости через сеттер и свойство

Внедрение зависимости через сеттер и свойство поддерживается через конфигурацию. При регистрации зависимости или при создании нового объекта, вы можете предоставить конфигурацию, которая будет использована контейнером для внедрения зависимостей через соответствующие сеттеры или свойства. Например,

⁸<http://martinfowler.com/articles/injection.html>

```
use yii\base\Object;

class Foo extends Object
{
    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'),
]);
```

Информация: Метод `yii\di\Container::get()` третьим аргументом принимает массив конфигурации, которым инициализируется создаваемый объект. Если класс реализует интерфейс `yii\base\Configurable` (например, `yii\base\Object`), то массив конфигурации передается в последний параметр конструктора класса. Иначе конфигурация применяется уже *после* создания объекта.

Внедрение зависимости через PHP callback

В данном случае, контейнер будет использовать зарегистрированный PHP callback для создания новых экземпляров класса. Каждый раз при вызове `yii\di\Container::get()` вызывается соответствующий callback. Callback отвечает за разрешения зависимостей и внедряет их в соответствии с вновь создаваемыми объектами. Например,

```
$container->set('Foo', function () {
    $foo = new Foo(new Bar);
    // ... дополнительная инициализация
    return $foo;
});

$foo = $container->get('Foo');
```

Для того, чтобы скрыть сложную логику инициализации нового объекта, можно использовать статический метод, возвращающий callable:

```

class FooBuilder
{
    public static function build()
    {
        return function () {
            $foo = new Foo(new Bar);
            // ... дополнительная инициализация
            return $foo;
        };
    }
}

$container->set('Foo', FooBuilder::build());

$foo = $container->get('Foo');

```

Как вы можете видеть, метод `FooBuilder::build()` возвращает анонимную функцию. Тот, кто будет настраивать класс `Foo`, теперь не обязан знать, как этот класс устроен.

5.9.2 Регистрация зависимостей

Вы можете использовать `yii\di\Container::set()` для регистрации зависимостей. При регистрации требуется имя зависимости, а так же определение зависимости. Именем зависимости может быть имя класса, интерфейса или алиас, так же определением зависимости может быть имя класса, конфигурационным массивом, или PHP callback'ом.

```

$container = new \yii\di\Container;

// регистрация имени класса, как есть. это может быть пропущено.
$container->set('yii\db\Connection');

// регистрация интерфейса
// Когда класс зависит от интерфейса, соответствующий класс
// будет использован в качестве зависимости объекта
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// регистрация алиаса. Вы можете использовать $container->get('foo')
// для создания экземпляра Connection
$container->set('foo', 'yii\db\Connection');

// Регистрация класса с конфигурацией. Конфигурация
// будет применена при создании экземпляра класса через get()
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// регистрация алиаса с конфигурацией класса
// В данном случае, параметр "class" требуется для указания класса

```



```
$container->set('db', [  
    'class' => 'yii\db\Connection',  
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',  
    'username' => 'root',  
    'password' => '',  
    'charset' => 'utf8',  
]);  
  
// регистрация PHP callback'a  
// Callback будет выполняться каждый раз при вызове $container->get('db')  
$container->set('db', function ($container, $params, $config) {  
    return new \yii\db\Connection($config);  
});  
  
// регистрация экземпляра компонента  
// $container->get('pageCache') вернёт тот же экземпляр при каждом вызове  
$container->set('pageCache', new FileCache);
```

Подсказка: Если имя зависимости такое же, как и определение соответствующей зависимости, то её повторная регистрация в контейнере внедрения зависимостей не нужна.

Зависимость, зарегистрированная через `set()` создаёт экземпляр каждый раз, когда зависимость необходима. Вы можете использовать `yii\di\Container::setSingleton()` для регистрации зависимости, которая создаст только один экземпляр:

```
$container->setSingleton('yii\db\Connection', [  
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',  
    'username' => 'root',  
    'password' => '',  
    'charset' => 'utf8',  
]);
```

5.9.3 Разрешение зависимостей

После регистрации зависимостей, вы можете использовать контейнер внедрения зависимостей для создания новых объектов, и контейнер автоматически разрешит зависимости их экземпляра и их внедрений во вновь создаваемых объектах. Разрешение зависимостей рекурсивно, то есть если зависимость имеет другие зависимости, эти зависимости также будут автоматически разрешены.

Вы можете использовать `yii\di\Container::get()` для создания новых объектов. Метод принимает имя зависимости, которым может быть имя класса, имя интерфейса или псевдоним. Имя зависимости может быть или не может быть зарегистрировано через `set()` или `setSingleton()`. Вы можете опционально предоставить список параметров конструктора класса и конфигурацию для настройки созданного объекта. Например,

```
// "db" ранее зарегистрированный псевдоним
$db = $container->get('db');

// эквивалентно: $engine = new \app\components\SearchEngine($apiKey, ['type'
=> 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey], ['type'
=> 1]);
```

За кулисами, контейнер внедрения зависимостей делает гораздо больше работы, чем просто создание нового объекта. Прежде всего, контейнер, осмотрит конструктор класса, чтобы узнать имя зависимого класса или интерфейса, а затем автоматически разрешит эти зависимости рекурсивно.

Следующий код демонстрирует более сложный пример. Класс `UserLister` зависит от объекта, реализующего интерфейс `UserFinderInterface`; класс `UserFinder` реализует этот интерфейс и зависит от объекта `Connection`. Все эти зависимости были объявлены через тип подсказки параметров конструктора класса. При регистрации зависимости через свойство, контейнер внедрения зависимостей позволяет автоматически разрешить эти зависимости и создаёт новый экземпляр `UserLister` простым вызовом `get('userLister')`.

```
namespace app\models;

use yii\base\Object;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends Object implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends Object
{
    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');

$listener = $container->get('userLister');

// что эквивалентно:

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$listener = new UserLister($finder);

```

5.9.4 Практическое использование

Yii создаёт контейнер внедрения зависимостей когда вы подключаете файл `Yii.php` во входном скрипте вашего приложения. Контейнер внедрения зависимостей доступен через `Yii::$container`. При вызове `Yii::createObject()`, метод на самом деле вызовет метод контейнера `get()`, чтобы создать новый объект. Как упомянуто выше, контейнер внедрения зависимостей автоматически разрешит зависимости (если таковые имеются) и внедрит их в только что созданный объект. Поскольку Yii использует `Yii::createObject()` в большей части кода своего ядра для создания новых объектов, это означает, что вы можете настроить глобальные объекты, имея дело с `Yii::$container`.

Например, вы можете настроить по умолчанию глобальное количество кнопок в пейджере `yii\widgets\LinkPager`:

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

Теперь, если вы вызовете в представлении виджет, используя следующий код, то свойство `maxButtonCount` будет инициализировано, как 5, вместо значения по умолчанию 10, как это определено в классе.

```
echo \yii\widgets\LinkPager::widget();
```

Хотя, вы всё ещё можете переопределить установленное значение через контейнер внедрения зависимостей:

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

Другим примером является использование автоматического внедрения зависимости через конструктор контейнера внедрения зависимостей. Предположим, ваш класс контроллера зависит от ряда других объектов, таких как сервис бронирования гостиницы. Вы можете объявить зависимость через параметр конструктора и позволить контейнеру внедрения зависимостей, разрешить её за вас.

```
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);
    }
}
```

Если у вас есть доступ к этому контроллеру из браузера, вы увидите сообщение об ошибке, который жалуется на то, что `BookingInterface` не может быть создан. Это потому что вы должны указать контейнеру внедрения зависимостей, как обращаться с этой зависимостью:

```
\Yii::$container->set('app\components\BookingInterface', 'app\components\
    BookingService');
```

Теперь, если вы попытаетесь получить доступ к контроллеру снова, то экземпляр `app\components\BookingService` будет создан и введён в качестве 3-го параметра конструктора контроллера.

5.9.5 Когда следует регистрировать зависимости

Поскольку зависимости необходимы тогда, когда создаются новые объекты, то их регистрация должна быть сделана как можно раньше. Ниже приведены рекомендуемые практики:

- Если вы разработчик приложения, то вы можете зарегистрировать зависимости во входном скрипте вашего приложения или в скрипте, подключённого во входном скрипте.
- Если вы разработчик распространяемого расширения, то вы можете зарегистрировать зависимости в загрузочном классе расширения.

5.9.6 Итог

Как `dependency injection`, так и `service locator` являются популярными паттернами проектирования, которые позволяют создавать программное обеспечение в слабосвязанной и более тестируемой манере. Мы настоятельно рекомендуем к прочтению статью Мартина Фаулера⁹, для более глубокого понимания `dependency injection` и `service locator`.

Yii реализует свой `service locator` поверх контейнера внедрения зависимостей. Когда `service locator` пытается создать новый экземпляр объекта, он перенаправляет вызов на контейнер внедрения зависимостей. Последний будет разрешать зависимости автоматически, как описано выше.

⁹<http://martinfowler.com/articles/injection.html>

Глава 6

Работа с базами данных

Error: not existing file: db-dao.md

Error: not existing file: db-query-builder.md

Error: not existing file: db-active-record.md

Error: not existing file: db-migrations.md

Error: not existing file: db-sphinx.md

Error: not existing file: db-redis.md

Error: not existing file: db-mongodb.md

Error: not existing file: db-elasticsearch.md

Глава 7

Получение данных от пользователя

Error: not existing file: input-forms.md

7.1 Проверка входящих данных

Как правило, вы никогда не должны доверять данным, полученным от пользователей и всегда проверять их прежде, чем работать с ними и добавлять в базу данных.

Учитывая модель данных которые должен заполнить пользователь, можно проверить эти данные на валидность воспользовавшись методом `[[yii\base\Model::validate()]]`. Метод возвращает логическое значение с результатом валидации ложь/истина. Если данные не валидны, ошибку можно получить воспользовавшись методом `[[yii\base\Model::errors]]`. Рассмотрим пример:

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

// модель заполненная пользовательскими данными
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // все данные корректны
} else {
    // данные не корректны: $errors - массив содержащий сообщения об ошибках
    $errors = $model->errors;
}
```

7.1.1 Правила проверки

Для того, чтобы `validate()` действительно работал, нужно объявить правила проверки атрибутов. Правила для проверки нужно указать в методе `[[yii\base\Model::rules()]]`. В следующем примере показано, как правила для проверки модели `ContactForm`, нужно объявлять:

```
public function rules()
{
    return [
        // атрибут required указывает, что name, email, subject, body
        // обязательны для заполнения
        [['name', 'email', 'subject', 'body'], 'required'],

        // атрибут email указывает, что в переменной email должен быть
        // корректный адрес электронной почты
        ['email', 'email'],
    ];
}
```

Метод должен `[[yii\base\Model::rules()|rules()]]` возвращать массив правил, каждое из которых является массивом в следующем формате:

```
[
    // обязательный, указывает, какие атрибуты должны быть проверены по
    // этому правилу.
    // Для одного атрибута, вы можете использовать имя атрибута не создавая
    // массив
```

```

        ['attribute1', 'attribute2', ...],

        // обязательный, указывает тип правила.
        // Это может быть имя класса, псевдоним валидатора, или метод для
        // проверки
        'validator',

        // необязательный, указывает, в каких случаях() это правило должно
        // применяться
        // если не указан, это означает, что правило применяется ко всем
        // сценариям
        // Вы также можете настроить "except" этот вариант применяет правило ко
        // всем
        // сценариям кроме перечисленных
        'on' => ['scenario1', 'scenario2', ...],

        // необязательный, задает дополнительные конфигурации для объекта
        // validator
        'property1' => 'value1', 'property2' => 'value2', ...
    ]

```

Для каждого правила необходимо указать, по крайней мере, какие атрибуты относятся к этому правилу и тип правила. Вы можете указать тип правила в одном из следующих форматов:

- Псевдонимы основного валидатора, например `required`, `in`, `date` и другие. Пожалуйста, обратитесь к списку Основных валидаторов за более подробной информацией.
- Название метода проверки в модели класса, или анонимную функцию. Пожалуйста, обратитесь к разделу Встроенных валидаторов за более подробной информацией.
- Полное имя класса валидатора. Пожалуйста, обратитесь к разделу Автономных валидаторов за более подробной информацией.

Правило может использоваться для проверки одного или нескольких атрибутов. Атрибут может быть проверен одним или несколькими правилами. Правило может быть применено только к определенным сценариям указав свойство `on`. Если вы не укажете свойство `on`, это означает, что правило будет применяться ко всем сценариям.

Когда вызывается метод `validate()` для проверки, он выполняет следующие действия:

1. Определяет, какие атрибуты должны проверяться путем получения списка атрибутов от `[[yii\base\Model::scenarios()]]` используя текущий `[[yii\base\Model::scenario|scenario]]`. Эти атрибуты называются - *активными атрибутами*.
2. Определяет, какие правила проверки должны использоваться, получив список правил от `[[yii\base\Model::rules()]]` используя текущий `[[yii\base\Model::scenario|scenario]]`. Эти правила называются - *активными правилами*.

3. Каждое активное правило проверяет каждый активный атрибут, который ассоциируется с правилом. Правила проверки выполняются в том порядке, как они перечислены.

Согласно вышеизложенным пунктам, атрибут будет проверяться, если и только если он является активным атрибутом, объявленных в `scenarios()` и связан с одним или несколькими активными правилами, объявленными в `rules()`.

Настройка сообщений об ошибках

Большинство валидаторов имеют сообщения об ошибках по умолчанию, которые будут добавлены к модели когда его атрибуты не проходят проверку. Например, `[[yii\validators\RequiredValidator|required]]` валидатор добавил к модели сообщение об ошибке “Имя пользователя не может быть пустым.” когда атрибут `username` не удовлетворил правила этого валидатора.

Вы можете настроить сообщение об ошибке для каждого правила, указав свойство `message` при объявлении правила, следующим образом:

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Please choose a username.'],
    ];
}
```

Некоторые валидаторы могут поддерживать дополнительные сообщения об ошибках, чтобы более точно описать причину ошибки. Например, `[[yii\validators\NumberValidator|number]]` валидатор поддерживает `[[yii\validators\NumberValidator::tooBig|tooBig]]` и `[[yii\validators\NumberValidator::tooSmall|tooSmall]]` для описания ошибки валидации, когда проверяемое значение является слишком большим и слишком маленьким, соответственно. Вы можете настроить эти сообщения об ошибках, как в настройках валидаторов, так и непосредственно в правилах проверки.

События валидации

Когда вызывается метод `[[yii\base\Model::validate()]]` он инициализирует вызов двух методов, которые можно переопределить, чтобы настроить процесс проверки:

- `[[yii\base\Model::beforeValidate()]]`: выполнение по умолчанию вызовет `[[yii\base\Model::EVENT_BEFORE_VALIDATE]]` событие. Вы можете переопределить этот метод, или обрабатывать это событие, чтобы сделать некоторую предобработку данных (например, форматирование входных данных), метод вызывается до начала валидации.

Этот метод должен возвращать логическое значение, указывающее, следует ли продолжать проверку или нет.

- `[[yii\base\Model::afterValidate()]]`: выполнение по умолчанию вызовет `[[yii\base\Model::EVENT_AFTER_VALIDATE]]` событие. Вы можете либо переопределить этот метод или обрабатывать это событие, чтобы сделать некоторую постобработку данных (Например, отформатировать данные удобным для дальнейшей обработки образом), метод вызывается после валидации.

Условные валидации

Для проверки атрибутов только при выполнении определенных условий, например если один атрибут зависит от значения другого атрибута можно использовать `[[yii\validators\Validator::when|when]]` свойство, чтобы определить такие условия. Например:

```
[
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }],
]
```

Это свойство `[[yii\validators\Validator::when|when]]` принимает PHP callable функцию с следующим описанием:

```
/**
 * @param Model $model модель используемая для проверки
 * @param string $attribute атрибут для проверки
 * @return boolean следует ли применять правило
 */
function ($model, $attribute)
```

Если вам нужна поддержка условной проверки на стороне клиента, вы должны настроить свойство метода `[[yii\validators\Validator::whenClient|whenClient]]` которое принимает строку, представляющую JavaScript функцию, возвращаемое значение определяет, следует ли применять правило или нет. Например:

```
[
    ['state', 'required', 'when' => function ($model) {
        return $model->country == 'USA';
    }, 'whenClient' => "function (attribute, value) {
        return $('#country').val() == 'USA';
    }"],
]
```

Фильтрация данных

Пользователь часто вводит данные которые нужно предварительно отфильтровать или предварительно обработать (очистить). Например, вы

хотите обрезать пробелы вокруг `username`. Вы можете использовать правила валидации для достижения этой цели.

В следующих примерах показано, как обрезать пробелы в входных данных и превратить пустые входные данные в `NULL` с помощью `trim` и указать значения по умолчанию с помощью свойства `default` основного валидатора:

```
[
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
]
```

Вы также можете использовать более сложные фильтрации данных с помощью анонимной функции подробнее об этом `filter`.

Как видите, эти правила валидации на самом деле не проверяют входные данные. Вместо этого, они будут обрабатывать значения и обратно возвращать результат работы. Фильтры по сути выполняют предобработку входящих данных.

Обработка пустых входных данных

Если входные данные представлены из HTML-формы, часто нужно присвоить некоторые значения по умолчанию для входных данных, если они не заполнены. Вы можете сделать это с помощью валидатора `default`. Например:

```
[
    // установим "username" и "email" как NULL, если они пустые
    [['username', 'email'], 'default'],

    // установим "level" как 1 если он пустой
    ['level', 'default', 'value' => 1],
]
```

По умолчанию входные данные считаются пустыми, если их значением является пустая строка, пустой массив или `null`. Вы можете настроить значение по умолчанию с помощью свойства `[[yii\validators\Validator::isEmpty]]` используя анонимную функцию. Например:

```
[
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }],
]
```

Примечание: большинство валидаторов не обрабатывает пустые входные данные, если их `[[yii\base\Validator::skipOnEmpty]]` свойство принимает значение по умолчанию `true`. Они просто будут пропущены во время проверки, если связанные с

ними атрибуты являются пустыми. Среди основных валидаторов, только `captcha`, `default`, `filter`, `required`, и `trim` будут обрабатывать пустые входные данные.

7.1.2 Специальная валидация

Иногда вам нужно сделать специальную валидацию для значений, которые не связаны с какой-либо модели.

Если необходимо выполнить только один тип проверки (например, проверка адреса электронной почты), вы можете вызвать метод `[[yii\validators\Validator::validate()|validate()]]` нужного валидатора. Например:

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'Email is valid.';
} else {
    echo $error;
}
```

Примечание: Не все валидаторы поддерживают такой тип проверки. Примером может служить `unique` валидатор, который предназначен для работы с моделью.

Если необходимо выполнить несколько проверок в отношении нескольких значений, вы можете использовать `[[yii\base\DynamicModel]]`, который поддерживает объявление, как атрибутов так и правил “на лету”. Его использование выглядит следующим образом:

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(compact('name', 'email'), [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // валидация завершилась с ошибкой
    } else {
        // Валидация успешно выполнена
    }
}
```

Метод `[[yii\base\DynamicModel::validateData()]]` создает экземпляр `DynamicModel`, определяет атрибуты, используя приведенные данные (`name` и `email` в этом примере), и затем вызывает `[[yii\base\Model::validate()]]` с данными правилами.

Кроме того, вы можете использовать следующий “классический” синтаксис для выполнения специальной проверки данных:


```

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()) {
        // валидация завершилась с ошибкой
    } else {
        // Валидация успешно выполнена
    }
}

```

После валидации, вы можете проверить успешность выполнения вызвав метод `[[yii\base\DynamicModel::hasErrors()|hasErrors()]]` и затем получить ошибки проверки вызвав метод `[[yii\base\DynamicModel::errors|errors]]` как это делают нормальные модели. Вы можете также получить доступ к динамическим атрибутам, определенным через экземпляр модели, например, `$model->name` и `$model->email`.

7.1.3 Создание Валидаторов

Кроме того, используя основные валидаторы, включенные в релизы Yii, вы также можете создавать свои собственные валидаторы. Вы можете создавать встроенные валидаторы или автономные валидаторы.

Встроенные Валидаторы

Встроенный валидатор наследует методы модели или использует анонимную функцию. Описание метода/функции:

```

/**
 * @param string $attribute атрибут проверяемый в настоящее время
 * @param array $params дополнительные пары имя-значение, заданное в правиле
 */
function ($attribute, $params)

```

Если атрибут не прошел проверку, метод/функция должна вызвать `[[yii\base\Model::addError()]]`, чтобы сохранить сообщение об ошибке в модели, для того чтобы позже можно было получить сообщение об ошибке для представления конечным пользователям.

Ниже приведены некоторые примеры:

```

use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;
}

```

```

public function rules()
{
    return [
        // встроенный валидатор определяется как модель метода
        validateCountry()
        ['country', 'validateCountry'],

        // встроенный валидатор определяется как анонимная функция
        ['token', function ($attribute, $params) {
            if (!ctype_alnum($this->$attribute)) {
                $this->addError($attribute, 'Токен должен содержать
буквы или цифры.');
            }
        }],
    ];
}

public function validateCountry($attribute, $params)
{
    if (!in_array($this->$attribute, ['USA', 'Web'])) {
        $this->addError($attribute, 'Страна должна быть либо "USA" или "
Web".');
    }
}
}

```

Примечание: по умолчанию, встроенные валидаторы не будут применяться, если связанные с ними атрибуты получат пустые входные данные, или если они уже не смогли пройти некоторые правила валидации. Если вы хотите, чтобы, что правило применялось всегда, вы можете настроить свойства `[[yii\validators\Validator::skipOnEmpty|skipOnEmpty]]` и/или `[[yii\validators\Validator::skipOnError|skipOnError]]` свойства `false` в правиле объявления. Например:

```

[
    ['country', 'validateCountry', 'skipOnEmpty' => false, '
skipOnError' => false],
]

```

Автономные валидаторы

Автономный валидатор - это класс, расширяющий `[[yii\validators\Validator]]` или его дочерних класс. Вы можете реализовать свою логику проверки путем переопределения метода `[[yii\validators\Validator::validateAttribute()]]`. Если атрибут не прошел проверку, вызвать `[[yii\base\Model::addError()]]`, чтобы сохранить сообщение об ошибке в модели, как это делают встроенные валидаторы. Например:

```

namespace app\components;

```

```
use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'Страна должна быть либо "
            USA" или "Web".');
        }
    }
}
```

Если вы хотите, чтобы ваш валидатор поддерживал проверку значений, без модели, также необходимо переопределить `[[yii\validators\Validator::validate()]]`. Вы можете также переопределить `[[yii\validators\Validator::validateValue()]]` вместо `validateAttribute()` и `validate()`, потому что по умолчанию последние два метода реализуются путем вызова `validateValue()`.

7.1.4 Валидация на стороне клиента

Проверка на стороне клиента на основе JavaScript целесообразна, когда конечные пользователи вводят входные данные через HTML-формы, так как эта проверка позволяет пользователям узнать, ошибки ввода быстрее, и таким образом улучшает ваш пользовательский интерфейс. Вы можете использовать или реализовать валидатор, который поддерживает валидацию на стороне клиента *в дополнение* к проверке на стороне сервера.

Информация: Проверка на стороне клиента желательна, но необязательна. Её основная цель заключается в предоставлении пользователям более удобного интерфейса. Так как входные данные, поступают от конечных пользователей, вы никогда не должны доверять верификации на стороне клиента. По этой причине, вы всегда должны выполнять верификацию на стороне сервера путем вызова `[[yii\base\Model::validate()]]`, как описано в предыдущих пунктах.

Использование валидации на стороне клиента

Многие основные валидаторы поддерживают проверку на стороне клиента out-of-the-box. Все, что вам нужно сделать, это просто использовать `[[yii\widgets\ActiveForm]]` для построения HTML-форм.

Например, `LoginForm` ниже объявляет два правила: один использует `required` основные валидаторы, который поддерживается на стороне кли-

ента и сервера; другой использует `validatePassword` встроенный валидатор, который поддерживается только на стороне сервера.

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // username и password обязательны для заполнения
            [['username', 'password'], 'required'],

            // проверке пароля с помощью validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Неправильное имя пользователя или
            пароль.');
```

HTML-форма построена с помощью следующего кода, содержит поля для ввода `username` и `password`. Если вы отправите форму, не вводя ничего, вы получите сообщения об ошибках, требующих ввести данные. Сообщения появятся сразу, без обращения к серверу.

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php yii\widgets\ActiveForm::end(); ?>
```

Класс `[[yii\widgets\ActiveForm]]` будет читать правила проверки заявленные в модели и генерировать соответствующий код JavaScript для валидаторов, которые поддерживают проверку на стороне клиента. Когда пользователь изменяет значение поля ввода или отправляет форму, JavaScript на стороне клиента будет срабатывать и проверять введенные данные.

Если вы хотите отключить проверку на стороне клиента полностью,

вы можете настроить свойство `[[yii\widgets\ActiveForm::enableClientValidation]]` установив значение `false`. Вы также можете отключить проверку на стороне клиента отдельных полей ввода, настроив их с помощью свойства `[[yii\widgets\ActiveField::enableClientValidation]]` установив значение `false`.

Реализация проверки на стороне клиента

Чтобы создать валидатор, который поддерживает проверку на стороне клиента, вы должны реализовать метод `[[yii\validators\Validator::clientValidateAttribute()]]` возвращающий фрагмент кода JavaScript, который выполняет проверку на стороне клиента. В JavaScript-коде, вы можете использовать следующие предопределенные переменные:

- **attribute**: имя атрибута для проверки.
- **value**: проверяемое значение.
- **messages**: массив, используемый для хранения сообщений об ошибках, проверки значения атрибута.
- **deferred**: массив, который содержит отложенные объекты (описано в следующем подразделе).

В следующем примере мы создаем `StatusValidator` который проверяет, if an input is a valid status input against the existing status data. Валидатор поддерживает оба способа проверки и на стороне сервера и на стороне клиента.

```
namespace app\components;

use yii\validators\Validator;
use app\models>Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = 'Invalid status input.';
    }

    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 (!$.isArray(value, $statuses)) {
    messages.push($message);
}
JS;
    }
}

```

Совет: приведенный выше код даётся, в основном, чтобы продемонстрировать, как осуществляется поддержка проверки на стороне клиента. На практике вы можете использовать `in` основные валидаторы для достижения той же цели. Вы можете написать проверку, как правило, например:

```

[
    ['status', 'in', 'range' => Status::find()->select('id')->
    asArray()->column()],
]

```

Отложенная валидация

Если Вам необходимо выполнить асинхронную проверку на стороне клиента, вы можете создавать Deferred objects¹. Например, чтобы выполнить пользовательские AJAX проверки, вы можете использовать следующий код:

```

public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}

```

В примере выше переменная `deferred` предусмотрена Yii, которая является массивом Отложенных объектов. `$.get()` метод jQuery создает Отложенный объект, который помещается в массив `deferred`.

Также можно явно создать Отложенный объект и вызвать его методом `resolve()`, тогда выполнется асинхронный вызов к серверу. В следующем примере показано, как проверить размеры загружаемого файла изображения на стороне клиента.

```

public function clientValidateAttribute($model, $attribute, $view)
{

```

¹<http://api.jquery.com/category/deferred-object/>

```
return <<<JS
  var def = $.Deferred();
  var img = new Image();
  img.onload = function() {
    if (this.width > 150) {
      messages.push('Изображение слишком широкое!');
    }
    def.resolve();
  }
  var reader = new FileReader();
  reader.onloadend = function() {
    img.src = reader.result;
  }
  reader.readAsDataURL(file);

  deferred.push(def);
JS;
}
```

Примечание: метод `resolve()` должен быть вызван после того, как атрибут был проверен. В противном случае основная проверка формы не будет завершена.

Для простоты работы с массивом `deferred`, существует упрощенный метод `add()`, который автоматически создает Отложенный объект и добавляет его в `deferred` массив. Используя этот метод, вы можете упростить пример выше, следующим образом:

```
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('Изображение слишком широкое!');
        }
        def.resolve();
      }
      var reader = new FileReader();
      reader.onloadend = function() {
        img.src = reader.result;
      }
      reader.readAsDataURL(file);
    });
  JS;
}
```

АЈАХ валидация

Некоторые проверки можно сделать только на стороне сервера, потому что только сервер имеет необходимую информацию. Например, чтобы

проверить логин пользователя на уникальность, необходимо проверить логин в базе данных на стороне сервера. Вы можете использовать проверку на основе AJAX в этом случае. Это вызовет AJAX-запрос в фоновом режиме, чтобы проверить логин пользователя, сохраняя при этом валидацию на стороне клиента. Выполняя её перед запросом к серверу.

Чтобы включить AJAX-валидацию для всей формы, Вы должны свойство `[[yii\widgets\ActiveForm::enableAjaxValidation]]` выбрать как `true` и указать `id` формы:

```
<?php $form = yii\widgets\ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]); ?>
```

Вы можете также включить AJAX валидацию или выключать для отдельных полей ввода, настроив их с помощью свойства `[[yii\widgets\ActiveField::enableAjaxValidation]]`.

Также необходимо подготовить сервер для обработки AJAX-запросов валидации. Это может быть достигнуто с помощью следующего фрагмента кода, в контроллере действий:

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
{
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

Приведенный выше код будет проверять, является ли текущий запрос AJAX. Если да, он будет отвечать на этот запрос, предварительно выполнив проверку и возвратит ошибки в случае их появления в формате JSON.

Информация: Вы также можете использовать Deferred Validation AJAX валидации. Однако, AJAX-функция проверки, описанные здесь более интегрированная и требует меньше усилий к написанию кода.

Error: not existing file: input-file-upload.md

Error: not existing file: input-multiple-models.md

Глава 8

Отображение данных

8.1 Форматирование данных

Для форматирования вывода Yii предоставляет класс, преобразующий данные в человеко понятный формат. `yii\i18n\Formatter` это класс-помощник, который зарегистрирован как компонент приложения, по умолчанию под именем `formatter`.

Он предоставляет набор методов для форматирования таких данных как дата/время, числа и другие часто используемые в целях локализации форматы. `Formatter` может быть использован двумя различными способами.

1. Напрямую используя методы форматирования (все методы форматирования имеют префикс `as`):

```
echo Yii::$app->formatter->asDate('2014-01-01', 'long'); // выведет:
    January 1, 2014
echo Yii::$app->formatter->asPercent(0.125, 2); // выведет: 12.50%
echo Yii::$app->formatter->asEmail('cebe@example.com'); // выведет: <a
    href="mailto:cebe@example.com">cebe@example.com</a>
echo Yii::$app->formatter->asBoolean(true); // выведет: Yes
// он также умеет отображать null значения:
echo Yii::$app->formatter->asDate(null); // выведет: (Not set)
```

2. Используя метод `format()` и имя формата. Этот метод также используется в виджетах наподобии `yii\grid\GridView` и `yii\widgets\DetailView`, в которых вы можете задать формат отображения данных в колонке через конфигурацию виджета.

```
echo Yii::$app->formatter->format('2014-01-01', 'date'); // выведет:
    January 1, 2014
// вы также можете использовать массивы для настроек метода
// форматирования:
// '2' это значение для $decimals параметра метода asPercent().
echo Yii::$app->formatter->format(0.125, ['percent', 2]); // выведет:
    12.50%
```

Все данные, отображаемые через компонент `formatter`, будут локализованы, если расширение PHP `intl`¹ было установлено. Для этого вы можете настроить свойство `locale`. Если оно не было настроено, то будет использован **язык приложения** в качестве локали. Подробнее см. раздел интернационализация. Компонент форматирования будет выбирать корректный формат для даты и чисел в соответствии с локалью, включая имена месяца и дней недели, переведённые на текущий язык. Форматирование дат также зависит от **временной зоны**, которая также будет из свойства `timeZone` приложения, если она не была задана явно.

Например, форматирование даты, вызванное с разной локалью, отобразит разные результаты::

```
Yii::$app->formatter->locale = 'en-US';
echo Yii::$app->formatter->asDate('2014-01-01'); // выведет: January 1, 2014
Yii::$app->formatter->locale = 'de-DE';
echo Yii::$app->formatter->asDate('2014-01-01'); // выведет: 1. Januar 2014
Yii::$app->formatter->locale = 'ru-RU';
echo Yii::$app->formatter->asDate('2014-01-01'); // выведет: 1 января 2014
г.
```

Обратите внимание, что форматирование может различаться между различными версиями библиотеки ICU, собранных с PHP, а также на основе того установлено ли [расширение PHP `intl`] (<http://php.net/manual/ru/book.intl.php>) или нет. Таким образом, чтобы гарантировать, что ваш сайт будет одинаково отображать данные во всех окружениях рекомендуется установить расширение PHP `intl` во всех окружениях и проверить, что версия библиотеки ICU совпадает. См. также: Настройка PHP окружения для интернационализации.

Отметим также, что даже если установлено расширение PHP `intl`, форматирование даты и времени для значений года ≥ 2038 или ≤ 1901 на 32-ух разрядных системах будет обращаться к реализации PHP, которая не обеспечивает локализованные имена месяца и дня, потому что в этом случае `intl` будет использовать 32-ух битный UNIX timestamp. На 64-битной системе `intl formatter` будет работать во всех случаях, если, конечно, `intl` был установлен.

8.1.1 Настройка форматирования

Форматы по-умолчанию, используемые в методах форматирования, можно настраивать через свойства **класса форматирования**. Вы можете задать форматирование по-умолчанию для всего приложения, настроив компонент `formatter` в вашей конфигурации приложения. Ниже приведён

¹<http://php.net/manual/ru/book.intl.php>

пример конфигурации. Чтобы узнать больше о доступных свойствах см. API документацию к классу `Formatter` и следующим подсекциям.

```
'components' => [  
  'formatter' => [  
    'dateFormat' => 'dd.MM.yyyy',  
    'decimalSeparator' => ',',  
    'thousandSeparator' => ' ',  
    'currencyCode' => 'EUR',  
  ],  
],
```

8.1.2 Форматирование значений даты и времени

Класс форматирования предоставляет различные методы для форматирования значений даты и времени. Например:

- **date** - значение будет отформатировано как дата, например `January 01, 2014`.
- **time** - значение будет отформатировано как время, например `14:23`.
- **datetime** - значение будет отформатировано как дата и время, например `January 01, 2014 14:23`.
- **timestamp** - значение будет отформатировано как unix timestamp², например, `1412609982`.
- **relativeTime** - значение будет отформатировано как временной промежуток между заданной датой и текущий временем в человеко понятном формате, например: `1 час назад`.

Форматирование даты и времени для методов `date`, `time` и `datetime` может быть задано глобально через конфигурацию свойств форматирования `$dateFormat`, `$timeFormat` и `$datetimeFormat`.

По-умолчанию, форматирование использует сокращенный формат, который интерпретируется по-разному в зависимости от активной в данный момент локали. Поэтому дата и время будут отформатированы наиболее часто используемым способом в стране и языке пользователя. Доступны 4 разных сокращенных формата:

- **short** в локале `en_GB` отобразит, например, `06/10/2014` для даты и `15:58` для времени, в то время как
- **medium** будет отображать `6 Oct 2014` и `15:58:42` соответственно,
- **long** будет отображать `6 October 2014` и `15:58:42 GMT` соответственно и
- **full** будет отображать `Monday, 6 October 2014` и `15:58:42 GMT` соответственно.

Дополнительно вы можете задать специальный формат, используя синтаксис, заданный ICU Project³, который описан в руководстве ICU по

²http://en.wikipedia.org/wiki/Unix_time

³<http://site.icu-project.org/>

следующему адресу: <http://userguide.icu-project.org/formatparse/datetime>. Также вы можете использовать синтаксис, который распознаётся РНР-функцией `date()`⁴, используя строку с префиксом `php:`.

```
// ICU форматирование
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06
// PHP date() форматирование
echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

Временные зоны

Для форматирования значений даты и времени Yii будет преобразовывать их в соответствии с **настроенной временной зоной**. Поэтому предполагается, что входные значения будут в UTC, если часовой пояс не был указан явно. По этой причине рекомендуется хранить все значения даты и времени в формате UTC, предпочтительно в виде UNIX timestamp, которая всегда во временной зоне UTC по определению. Если входное значение находится в часовом поясе, отличном от UTC, часовой пояс должен быть указан явно, как в следующем примере:

```
// при условии Yii::$app->timeZone = 'Europe/Berlin';
echo Yii::$app->formatter->asTime(1412599260); // 14:41:00
echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00
echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

Начиная с версии 2.0.1 стало возможно настраивать временную зону для предполагаемых timestamp, которые не включают в себя временную зону, как во втором примере в коде выше. Вы можете задать `yii\i18n\Formatter::$defaultTimeZone` временной зоной, которую вы используете для хранения данных.

Примечание: Поскольку временные зоны являются субъектом ответственности правительств по всему миру и могут часто меняться, это значит, что вы, вероятно, не имеете самую свежую информацию в базе данных временных зон, установленной на вашем сервере. Вы можете обратиться к ICU руководству⁵ для получения подробностей об обновлении базы данных временных зон. См. также: Настройка вашего РНР окружения для интернационализации.

8.1.3 Форматирование чисел

Для форматирования числовых значений класс форматирования предоставляет следующие методы:

⁴<http://php.net/manual/ru/function.date.php>

⁵<http://userguide.icu-project.org/datetime/timezone#TOC-Updating-the-Time-Zone-Data>

- **integer** - значение будет отформатировано как целое число, например 42.
- **decimal** - значение будет отформатировано как дробное число, состоящее из целого и дробной части, например: 2,542.123 или 2.542,123.
- **percent** - значение будет отформатировано как процентное значение, например 42%.
- **scientific** - значение будет отформатировано в научном формате, например: 4.2E4.
- **currency** - значение будет отформатировано в денежном формате, например: \$420.00. Обратите внимание, чтобы эта функция работала правильно, локаль должна включать в себя часть со страной, например: **en_GB** или **en_US** потому что только язык будет неоднозначным в этом случае.
- **size** - значение будет отформатировано как количество байт в человеко понятном формате, например: 410 kibibytes.
- **shortSize** - сокращённая версия **size**, например: 410 KiB.

Форматирование чисел может быть скорректирована с помощью **дробного разделителя** и **тысячного разделителя**, которые были заданы в соответствии с локалью.

Для более сложной конфигурации, `yii\i18n\Formatter::$numberFormatterOptions` и `yii\i18n\Formatter::$numberFormatterTextOptions` могут быть использованы для настройки внутренне используемого класса `NumberFormatter`⁶

Например, чтобы настроить максимальное и минимальное количество знаков после запятой, вы можете настроить свойство `yii\i18n\Formatter::$numberFormatterOptions` как в примере ниже:

```
'numberFormatterOptions' => [  
    NumberFormatter::MIN_FRACTION_DIGITS => 0,  
    NumberFormatter::MAX_FRACTION_DIGITS => 2,  
]
```

8.1.4 Остальное форматирование

Кроме форматирования даты, времени и чисел, Yii предоставляет набор других полезных средств форматирования для различных ситуаций:

- **raw** - значением будет отображено как есть, это псевдо-форматирование, которое не имеет никакого эффекта, кроме значений **null**, которые будут отформатированы в соответствии с **nullDisplay**.
- **text** - значением будет экранированный от HTML текст. Это формат по-умолчанию, используемый в `GridView DataColumn`.
- **ntext** - значением будет экранированный от HTML текст с новыми строками, сконвертированными в разрывы строк.

⁶<http://php.net/manual/ru/class.numberformatter.php>

- `paragraphs` - значением будет экранированный от HTML текст с параграфами, обрамлёнными в `<p>` теги.
- `html` - значение будет очищено, используя `HtmlPurifier` с целью предотвратить XSS атаки. Вы можете задать дополнительные параметры, такие как `['html', ['Attr.AllowedFrameTargets' => ['_blank']]]`.
- `email` - значение будет отформатировано как ссылка `mailto`.
- `image` - значение будет отформатировано как тег картинки.
- `url` - значение будет отформатировано как ссылка `<a>`.
- `boolean` - значение форматируется как логическое. По-умолчанию `true` будет отображено как `Yes` и `false` как `No`, переведенное на язык приложения. Вы можете настроить это через свойство `yii\i18n\Formatter::$booleanFormat`.

8.1.5 null значения

Для значений `null` в PHP, класс форматирования будет отображать вместо пустой строки маркер, по-умолчанию это (`not set`), переведенный на язык приложения. Вы можете настроить свойство `nullDisplay` для установки собственного маркера. Если вы не хотите обрабатывать `null` значения, то установите свойство `nullDisplay` в `null`.

8.2 Постраничное разделение данных

В случае когда требуется отобразить слишком много данных на одной странице, эта страница зачастую разделяется на несколько частей, каждая из которых содержит и отображает только часть данных за один раз. Такие части называются страницами, а сам процесс называется постраничным разделением данных.

Если вы используете источник данных с одним из виджетов данных, то в этом случае будет автоматически использовано постраничное разделение данных. В противном случае вам требуется создать объект `yii\data\Pagination`, заполнить его такими данными как общее количество элементов, количество элементов на одной странице и текущая страница, затем применить его к запросу и передать в элемент нумерации страниц.

Первым делом в действии контроллера мы создаем объект постраничного разделения данных и заполняем его данными:

```
function actionIndex()
{
    $query = Article::find()->where(['status' => 1]);
    $countQuery = clone $query;
    $pages = new Pagination(['totalCount' => $countQuery->count()]);
    $models = $query->offset($pages->offset)
        ->limit($pages->limit)
```



```

        ->all();

    return $this->render('index', [
        'models' => $models,
        'pages' => $pages,
    ]);
}

```

Затем в представлении мы выводим модели для текущей страницы и передаем объект pagination для разделения данных в элемент нумерации страниц:

```

foreach ($models as $model) {
    // отображаем здесь $model
}

// отображаем ссылки на страницы
echo LinkPager::widget([
    'pagination' => $pages,
]);

```

8.3 Сортировка

Иногда выводимые данные требуется отсортировать в соответствии с одним или несколькими атрибутами. Если вы используете источник данных с одним из виджетов данных, сортировка будет применена автоматически. В противном случае вам должны создать экземпляр `yii\data\Sort`, настроить его и применить к запросу. Он также может быть передан в представление, где будет использован для создания ссылок на сортировку по определенным атрибутам.

Ниже приведен типичный пример использования сортировки,

```

function actionIndex()
{
    $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' => 'Name',
        ],
    ],
    ];

    $models = Article::find()
        ->where(['status' => 1])
        ->orderBy($sort->orders)
}

```

```
        ->all();

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

В представлении:

```
// Отображение ссылок на различные действия сортировок
echo $sort->link('name') . ' | ' . $sort->link('age');

foreach ($models as $model) {
    // здесь отображаем модель $model
}
```

В примере выше, мы объявляем два атрибута, которые поддерживают сортировку: `name` and `age`. Мы передаем информацию о сортировке в запрос статьи, поэтому результаты запроса будут отсортированы согласно сортировке, установленной в объекте `Sort`. В представлении, мы отображаем две ссылки, которые ведут на страницы с данными, отсортированными по соответствующим атрибутам.

Класс `Sort` будет автоматически принимать параметры, переданные с запросом и в соответствии с ними настраивать параметры сортировки. Вы можете регулировать список принимаемых параметров через настройку свойства `$params`.

Error: not existing file: output-data-providers.md

Error: not existing file: output-data-widgets.md

Error: not existing file: output-theming.md

Глава 9

Безопасность

Error: not existing file: security-authentication.md

Error: not existing file: security-authorization.md

Error: not existing file: security-passwords.md

Error: not existing file: security-auth-clients.md

Error: not existing file: security-best-practices.md

Глава 10

Кэширование

10.1 Кэширование

Кэширование — это простой и эффективный способ повысить производительность веб-приложения. Сохраняя относительно статичные данные в кэше и извлекая их из кэша, когда потребуется, мы экономим время, затрачиваемое на генерацию данных с нуля каждый раз.

Кэширование может использоваться на различных уровнях и в различных местах веб-приложения. На стороне сервера, на более низком уровне мы используем кэширование для хранения основных данных, таких как список последних полученных из базы данных статей. На более высоком уровне кэш может использоваться для хранения фрагментов или целых веб-страниц. Таких, например, как результат рендеринга последних статей. На стороне клиента для сохранения содержимого недавно посещенных страниц в кэше браузера может использоваться HTTP-кэширование.

Yii поддерживает все эти механизмы кэширования:

- Кэширование данных
- Кэширование фрагментов
- Кэширование страниц
- HTTP-кэширование

10.2 Кэширование данных

Кэширование данных заключается в сохранении некоторой переменной РНР в кэше и последующее её извлечение. Оно является основой для расширенных возможностей, таких как кэширование запросов и кэширование страниц.

Приведённый ниже код является типичным случаем кэширования данных, где `$cache` указывает на компонент кэширования:

```
// Пробуем извлечь $data из кэша.
```

```
$data = $cache->get($key);

if ($data === false) {

    // $data нет в кэше, считаем с нуля.

    // Сохраняем значение $data в кэше. Данные можно получить в следующий
    раз.
    $cache->set($key, $data);
}

// Значение $data доступно здесь.
```

10.2.1 Компоненты кэширования

Кэширование данных опирается на *компоненты кэширования*, которые представляют различные хранилища, такие как память, файлы и базы данных.

Кэш-компоненты, как правило, зарегистрированы в качестве компонентов приложения, так что их можно настраивать и обращаться к ним глобально. Следующий код показывает, как настроить компонент приложения `cache` для использования Memcached¹ с двумя серверами:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

Вы можете получить доступ к компоненту кэша, используя выражение `Yii::$app->cache`.

Поскольку все компоненты кэша поддерживают единый API-интерфейс, вы можете менять основной компонент кэша на другой через конфигурацию приложения. Код, использующий кэш, при этом не меняется. Например, конфигурацию выше для использования APC cache можно изменить следующим образом:

¹<http://memcached.org/>

```
'components' => [  
    'cache' => [  
        'class' => 'yii\caching\ApcCache',  
    ],  
],
```

Совет: Вы можете зарегистрировать несколько кэш-компонентов приложения. Компонент с именем `cache` используется по умолчанию многими классами (например, `yii\web\UrlManager`).

Поддерживаемые хранилища

Yii поддерживает множество хранилищ кэша:

- `yii\caching\ApcCache`: использует расширение PHP APC². Эта опция считается самой быстрой при работе с кэшем в «толстом» централизованном приложении (т.е. один сервер, без выделенного балансировщика нагрузки и т.д.).
- `yii\caching\DbCache`: использует таблицу базы данных для хранения кэшированных данных. Чтобы использовать этот кэш, вы должны создать таблицу, как описано в `yii\caching\DbCache::$cacheTable`.
- `yii\caching\DummyCache`: является кэшем-пустышкой, не реализующим реального кэширования. Смысл этого компонента в упрощении кода, который должен проверить наличие кэша. Вы можете использовать данный тип кэша, например, при разработке или если сервер не поддерживает кэш и переключиться на реальное кэширование позже. Для извлечения данных, в этом случае, используется один и тот же код `Yii::$app->cache->get($key)`. При этом можно не беспокоиться, что `Yii::$app->cache` может быть `null`.
- `yii\caching\FileCache`: использует обычные файлы для хранения кэшированных данных. Замечательно подходит для кэширования больших кусков данных, таких как содержимое страницы.
- `yii\caching\MemCache`: использует расширения PHP memcache³ и memcached⁴. Этот вариант может рассматриваться как самый быстрый при работе в распределенных приложениях (например, с несколькими серверами, балансировкой нагрузки и так далее).
- `yii\redis\Cache`: реализует компонент кэша на основе Redis⁵, хранилища ключ-значение (требуется Redis версии 2.6.12 или выше).
- `yii\caching\WinCache`: использует расширение PHP WinCache⁶ (смотрите также⁷).

²<http://php.net/manual/en/book.apc.php>

³<http://php.net/manual/en/book.memcache.php>

⁴<http://php.net/manual/en/book.memcached.php>

⁵<http://redis.io/>

⁶<http://iis.net/downloads/microsoft/wincache-extension>

⁷<http://php.net/manual/en/book.wincache.php>

- `yii\caching\XCache`: использует расширение PHP XCache⁸.
- Zend Data Cache⁹.

Совет: Вы можете использовать разные способы хранения кэша в одном приложении. Общая стратегия заключается в использовании памяти под хранение небольших часто используемых данных (например, статистические данные). Для больших и реже используемых данных (например, содержимое страницы) лучше использовать файлы или базу данных.

10.2.2 Кэш API,

У всех компонентов кэша один базовый класс `yii\caching\Cache` со следующими методами:

- `get()`: возвращает данные по указанному ключу. Значение `false` будет возвращено, если данные не найдены или устарели.
- `set()`: сохраняет данные по ключу.
- `add()`: сохраняет данные по ключу если такого ключа ещё нет.
- `mget()`: извлекает сразу несколько элементов данных из кэша по заданным ключам.
- `mset()`: сохраняет несколько элементов данных. Каждый элемент идентифицируется ключом.
- `madd()`: сохраняет несколько элементов данных. Каждый элемент идентифицируется ключом. Если ключ уже существует, сохранение не происходит.
- `exists()`: есть ли указанный ключ в кэше.
- `delete()`: удаляет указанный ключ.
- `flush()`: удаляет все данные.

Примечание: Не кэшируйте непосредственно значение `false`, потому что `get()` использует `false` для случая, когда данные не найдены в кэше. Вы можете обернуть `false` в массив и закэшировать его, чтобы избежать данной проблемы.

Некоторые кэш-хранилища, например, MemCache или APC, поддерживают получение нескольких значений в пакетном режиме, что может сократить накладные расходы на получение данных. Данную возможность можно использовать при помощи `mget()` и `madd()`. В случае, если хранилище не поддерживает эту функцию, она будет имитироваться.

Так как `yii\caching\Cache` реализует `ArrayAccess`, компонент кэша можно использовать как массив:

⁸<http://xcache.lighttpd.net/>

⁹http://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm


```
$cache['var1'] = $value1; // эквивалентно: $cache->set('var1', $value1);  
$value2 = $cache['var2']; // эквивалентно: $value2 = $cache->get('var2');
```

Ключи кэша

Каждый элемент данных, хранящийся в кэше, идентифицируется ключом. Когда вы сохраняете элемент данных в кэше, необходимо указать для него ключ. Позже, когда вы извлекаете элемент данных из кэша, вы должны предоставить соответствующий ключ.

Вы можете использовать строку или произвольное значение в качестве ключа кэша. Если ключ не строка, то он будет автоматически сериализован в строку.

Обычно ключ задаётся массивом всех значимых частей. Например, для хранения информации о таблице в `yii\db\Schema` для ключа используются следующие части:

```
[  
    __CLASS__,          // Название класса схемы.  
    $this->db->dsn,       // Данные для соединения с базой.  
    $this->db->username,  // Логин для соединения с базой.  
    $name,              // Название таблицы.  
];
```

Как вы можете видеть, ключ строится так, чтобы однозначно идентифицировать данные таблицы.

Если одно хранилище кэша используется несколькими приложениями, во избежание конфликтов стоит указать префикс ключа. Сделать это можно путём настройки `yii\caching\Cache::$keyPrefix`:

```
'components' => [  
    'cache' => [  
        'class' => 'yii\caching\ApcCache',  
        'keyPrefix' => 'myapp', // уникальный префикс ключей кэша  
    ],  
],
```

Для обеспечения совместимости должны быть использованы только алфавитно-цифровые символы.

Срок действия кэша

Элементы данных, хранимые в кэше, остаются там навсегда если только они не будут удалены из-за особенностей функционирования хранилища (например, место для кэширования заполнено и старые данные удаляются). Чтобы изменить этот режим, вы можете передать истечение срока действия ключа при вызове метода `set()`. Параметр указывает, сколько секунд элемент кэша может считаться актуальным. Если срок годности ключа истёк, `get()` вернёт `false`:

```
// Хранить данные в кэше не более 45 секунд
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // срок действия истек или ключ не найден в кэше
}
```

Зависимости кэша

В добавок к изменению срока действия ключа, элемент может быть признан недействительным из-за *изменения зависимостей*. К примеру, `yii\caching\FileDependency` представляет собой зависимость от времени изменения файла. Когда это время изменяется, и файл. Любые устаревшие данные, найденные в кэше должны быть признаны недействительным, а `get()` в этом случае должен вернуть `false`.

Зависимости кэша представлены в виде объектов потомков класса `yii\caching\Dependency`. Когда вы вызываете метод `set()`, чтобы сохранить элемент данных в кэше, вы можете передать туда зависимость. Например:

```
// Создать зависимость от времени модификации файла example.txt.
$dependency = new \yii\caching\FileDependency(['fileName' => 'example.txt'])
;

// Данные устаревают через 30 секунд.
// Данные могут устареть и раньше, если example.txt будет изменён.
$cache->set($key, $data, 30, $dependency);

// Кэш будет проверен, если данные устарели.
// Он также будет проверен, если указанная зависимость была изменена.
// Вернется false, если какое-либо из этих условий выполнено.
$data = $cache->get($key);
```

Ниже приведен список доступных зависимостей кэша:

- `yii\caching\ChainedDependency`: зависимость меняется, если любая зависимость в цепочке изменяется.
- `yii\caching\DbDependency`: зависимость меняется, если результат некоторого определенного SQL запроса изменён.
- `yii\caching\ExpressionDependency`: зависимость меняется, если результат определенного PHP выражения изменён.
- `yii\caching\FileDependency`: зависимость меняется, если изменилось время последней модификации файла.
- `yii\caching\TagDependency`: Связывает кэшированные данные элемента с одним или несколькими тегами. Вы можете аннулировать кэширование данных элементов с заданным тегом(тегами) по вызову `yii\caching\TagDependency::invalidate()`.

10.2.3 Кэширование запросов

Кэширование запросов - это специальная функция кэширования, построенная на основе кэширования данных. Она предназначена для кэширования результатов запросов к базе данных.

Кэширование запросов требует **DB connection** и приложения действительный **cache**. Простое использование запросов кэширования происходит следующим образом, предполагая что **\$db** это экземпляр **yii\db\Connection**:

```
$result = $db->cache(function ($db) {  
  
    // Результат SQL запроса будет возвращен из кэша если  
    // кэширование запросов включено и результат запроса присутствует в кэше  
    return $db->createCommand('SELECT * FROM customer WHERE id=1')->queryOne()  
    ;  
  
});
```

Кэширование запросов может быть использовано как для DAO, так и для ActiveRecord:

```
$result = Customer::getDb()->cache(function ($db) {  
    return Customer::find()->where(['id' => 1])->one();  
});
```

Информация: Некоторые СУБД (например, MySQL¹⁰) поддерживают кэширование запросов на стороне сервера БД. Вы можете использовать любой механизм кэширования запросов. Кэширование запросов описанное выше, имеет преимущество, поскольку можно указать гибкие зависимости кэша и это более эффективно.

Конфигурации

Кэширование запросов имеет три глобальных конфигурационных параметра через **yii\db\Connection**:

- **enableQueryCache**: включить или выключить кэширование запросов. По умолчанию **true**. Стоит отметить, что для использования кэширования вам может понадобиться компонент **cache**, как показано в **queryCache**.
- **queryCacheDuration**: количество секунд, в течение которых результат кэшируется. Для бесконечного кэша используйте **0**. Именно это значение выставляется **yii\db\Connection::cache()** если не указать время явно.

¹⁰<http://dev.mysql.com/doc/refman/5.1/en/query-cache.html>

- `queryCache`: ID компонента кэширования. По умолчанию `'cache'`. Кэширования запросов работает только если используется компонент приложения кэш.

Использование

Вы можете использовать `yii\db\Connection::cache()`, если у вас есть несколько SQL запросов, которые необходимо закэшировать:

```
$duration = 60; // кэширование результата на 60 секунд
$dependency = ...; // параметры зависимости

$result = $db->cache(function ($db) {

    // ... выполнять SQL запросы здесь ...

    return $result;

}, $duration, $dependency);
```

Любые SQL запросы в анонимной функции будут кэшироваться в течение указанного промежутка времени с заданной зависимостью. Если результат в кэше актуален, запрос будет пропущен и, вместо этого, из кэша будет возвращен результат. Если вы не укажете `'$duration'`, значение `queryCacheDuration` будет использоваться вместо него.

Иногда в пределах `"cache()"` вы можете отключить кэширование запроса. В этом случае вы можете использовать `yii\db\Connection::noCache()`.

```
$result = $db->cache(function ($db) {

    // SQL запросы, которые используют кэширование

    $db->noCache(function ($db) {

        // SQL запросы, которые не используют кэширование

    });

    // ...

    return $result;

});
```

Если вы просто хотите использовать кэширование для одного запроса, вы можете вызвать `yii\db\Command::cache()` при построении команды. Например:

```
// использовать кэширование запросов и установить срок действия кэша на 60 секунд
$customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->cache(60)->queryOne();
```

Вы также можете использовать `yii\db\Command::noCache()`, чтобы отключить кэширование запросов для одной команды. Например:

```
$result = $db->cache(function ($db) {  
  
    // Используется кэширование SQL запросов  
  
    // не использовать кэширование запросов для этой команды  
    $customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->  
        noCache()->queryOne();  
  
    // ...  
  
    return $result;  
});
```

Ограничения

Кэширование запросов не работает с результатами запросов, которые содержат обработчики ресурсов. Например, при использовании типа столбца `BLOB` в некоторых СУБД, в качестве результата запроса будет выведен ресурс обработчик данных столбца.

Некоторые кэш хранилища имеют ограничение в размере данных. Например, `Memcache` ограничивает максимальный размер каждой записи до 1 Мб. Таким образом, если результат запроса превышает этот предел, данные не будут закэшированы.

10.3 Кэширование фрагментов

Кэширование фрагментов относится к кэшированию фрагментов страницы. Например, если страница отображает в таблице суммарные годовые продажи, мы можем сохранить эту таблицу в кэше с целью экономии времени, требуемого для создания таблицы при каждом запросе. Кэширование фрагментов основано на кэшировании данных.

Для кэширование фрагментов используйте следующий код в представлении:

```
if ($this->beginCache($id)) {  
  
    // ... здесь создаём содержимое ...  
  
    $this->endCache();  
}
```

Таким образом заключите то, что вы хотите закэшировать между вызовом `beginCache()` и `endCache()`. Если содержимое будет найдено в кэше, `beginCache()` отобразит закэшированное содержимое и вернёт `false`, минуя генерацию содержимого. В противном случае, будет выполнен код

генерации контента и когда будет вызван `endCache()`, то сгенерированное содержимое будет записано и сохранено в кэше.

Также как и кэширование данных, для кэширования фрагментов требуется уникальный идентификатор для определения кэшируемого фрагмента.

10.3.1 Параметры кэширования

Вызывая метод `beginCache()`, мы можем передать в качестве второго аргумента массив, содержащий параметры кэширования для управления кэшированием фрагмента. Заглядывая за кулисы, можно увидеть, что этот массив будет использоваться для настройки виджета `yii\widgets\FragmentCache`, который реализует фактическое кэширование фрагментов.

Срок хранения

Наверное, наиболее часто используемым параметром является `duration`. Он определяет какое количество секунд содержимое будет оставаться действительным (корректным). Следующий код помещает фрагмент в кэш не более, чем на час:

```
if ($this->beginCache($id, ['duration' => 3600])) {  
  
    // ... здесь создаём содержимое ...  
  
    $this->endCache();  
}
```

Если мы не установим длительность (срок хранения), она будет равна значению по умолчанию (60 секунд). Это значит, что кэшированное содержимое станет недействительным через 60 секунд.

Зависимости

Также как и кэширование данных, кэшируемое содержимое фрагмента тоже может иметь зависимости. Например, отображение содержимого сообщения зависит от того, изменено или нет это сообщение.

Для определения зависимости мы устанавливаем параметр `dependency`, который может быть либо объектом `yii\caching\Dependency`, либо массивом настроек, который может быть использован для создания объекта `yii\caching\Dependency`. Следующий код определяет содержимое фрагмента, зависящее от изменения значения столбца `updated_at`:

```
$dependency = [  
    'class' => 'yii\caching\DbDependency',  
    'sql' => 'SELECT MAX(updated_at) FROM post',  
];
```

```
if ($this->beginCache($id, ['dependency' => $dependency])) {  
    // ... здесь создаём содержимое ...  
    $this->endCache();  
}
```

Вариации

Кэшируемое содержимое может быть изменено в соответствии с некоторыми параметрами. Например, для веб-приложений, поддерживающих несколько языков, одна и та же часть кода может создавать содержимое на нескольких языках. Поэтому у вас может возникнуть желание кэшировать содержимое в зависимости от текущего языка приложения.

Чтобы задать вариации кэша, установите параметр **variations**, который должен быть массивом, содержащим скалярные значения, каждое из которых представляет определенный коэффициент вариации. Например, чтобы кэшировать содержимое в зависимости от языка приложения, вы можете использовать следующий код:

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {  
    // ... здесь создаём содержимое ...  
    $this->endCache();  
}
```

Переключение кэширования

Иногда может потребоваться включать кэширование фрагментов только для определённых условий. Например, страницу с формой мы хотим кэшировать только тогда, когда обращение к ней произошло впервые (посредством GET запроса). Любое последующее отображение формы (посредством POST запроса) не должно быть кэшировано, потому что может содержать данные, введенные пользователем. Для этого мы задаём параметр **enabled**:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {  
    // ... здесь создаём содержимое ...  
    $this->endCache();  
}
```

10.3.2 Вложенное кэширование

Кэширование фрагментов может быть вложенным. Это значит, что кэшируемый фрагмент окружён более крупным фрагментом (содержится

в нём), который также кэшируется. Например, комментарии кэшируются во внутреннем фрагменте кэша, и они же кэшируются вместе с содержимым сообщения во внешнем фрагменте кэша. Следующий код демонстрирует как два фрагмента кэша могут быть вложенными:

```
if ($this->beginCache($id1)) {  
    // логика... создания контента...  
    if ($this->beginCache($id2, $options2)) {  
        // логика... создания контента...  
        $this->endCache();  
    }  
    // логика... создания контента...  
    $this->endCache();  
}
```

Параметры кэширования могут быть различными для вложенных кэшей. Например, внутренний и внешний кэши в вышеприведённом примере могут иметь разные сроки хранения. Даже когда данные внешнего кэша уже не являются актуальными, внутренний кэш может содержать актуальный фрагмент. Тем не менее, обратное не верно. Если внешний кэш актуален, данные будут отдаваться из него даже если внутренний кэш содержит устаревшие данные. Следует проявлять осторожность при выставлении срока хранения и задания зависимостей для вложенных кэшей. В противном случае вы можете получить устаревшие данные.

10.3.3 Динамическое содержимое

Когда используется кэширование фрагментов, вы можете столкнуться с ситуацией когда большой фрагмент содержимого статичен за исключением одного или нескольких мест. Например, заголовок страницы может отображаться в главном меню вместе с именем текущего пользователя. Еще одна проблема в том, что содержимое, которое было закэшировано, может содержать PHP код, который должен выполняться для каждого запроса (например код для регистрации в asset bundle). Обе проблемы могут быть решены с помощью, так называемой функции *динамического содержимого*.

Динамическое содержимое значит, что часть вывода не будет закэширована даже если она заключена в кэширование фрагментов. Чтобы сделать содержимое динамическим постоянно, оно должно быть создано, используя специальный PHP код.

Вы можете вызвать `yii\base\View::renderDynamic()` в пределах кэширования фрагмента для вставки динамического содержимого в нуж-

ное место, как в примере ниже:

```
if ($this->beginCache($id1)) {  
  
    // логика... создания контента...  
  
    echo $this->renderDynamic('return Yii::$app->user->identity->name;');  
  
    // логика... создания контента...  
  
    $this->endCache();  
}
```

Метод `renderDynamic()` принимает некоторую часть PHP кода как параметр. Возвращаемое значение этого кода будет вставлено в динамическое содержимое. Этот PHP код будет выполняться для каждого запроса, независимо от того находится ли он внутри кэширования фрагмента или нет.

10.4 Кэширование страниц

Кэширование страниц — это кэширование всего содержимого страницы на стороне сервера. Позже, когда эта страница будет снова запрошена, сервер вернет её из кэша вместо того чтобы генерировать её заново.

Кэширование страниц осуществляется при помощи фильтра действия `yii\filters\PageCache` и может быть использовано в классе контроллера следующим образом:

```
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',  
            ],  
        ],  
    ];  
}
```

Приведённый код задействует кэширование только для действия `index`. Содержимое страницы кэшируется максимум на 60 секунд и варьируется в зависимости от текущего языка приложения. Кэшированная страница должна быть признана просроченной, если общее количество постов изменилось.

Кэширование страниц очень похоже на кэширование фрагментов. В обоих случаях поддерживаются параметры `duration` (продолжительность), `dependencies` (зависимости), `variations` (вариации), и `enabled` (включен). Главное отличие заключается в том, что кэширование страницы реализовано в виде фильтра действия, а кэширование фрагмента в виде виджета.

Вы можете использовать вместе кэширование фрагмента, динамическое содержимое и кэширование страницы.

10.5 HTTP кэширование

Кроме серверного кэширования, которое мы описали в предыдущих разделах, веб-приложения также могут использовать кэширование на стороне клиента, чтобы сэкономить время для формирования и передачи одного и того же содержания страницы.

Чтобы использовать кэширование на стороне клиента, вы можете настроить `yii\filters\HttpCache` в качестве фильтра для действия контроллера, отображающего результат, который может быть закэширован на стороне клиента. `HttpCache` работает только для `GET` и `HEAD` запросов. Для этих запросов он может обрабатывать три вида HTTP заголовков, относящихся к кэшированию:

- `Last-Modified`
- `Etag`
- `Cache-Control`

10.5.1 Заголовок `Last-Modified`

Заголовок `Last-Modified` использует временную метку `timestamp`, чтобы показать была ли страница изменена после того, как клиент закэшировал её.

Вы можете настроить свойство `yii\filters\HttpCache::$lastModified`, чтобы включить отправку заголовка `Last-Modified`. Свойство должно содержать РНР-функцию, возвращающую временную метку UNIX `timestamp` времени последнего изменения страницы. Сигнатура РНР-функции должна совпадать со следующей,

```
/**
 * @param Action $action объект действия, которое в настоящее время
 *                   обрабатывается
 * @param array $params значение свойства "params"
 * @return integer временная метка UNIX timestamp, возвращающая время
 *                последнего изменения страницы
 */
function ($action, $params)
```

Ниже приведён пример использования заголовка `Last-Modified`:

```

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');
            },
        ],
    ];
}

```

Приведенный выше код устанавливает, что HTTP кэширование должно быть включено только для действия `index`. Он генерирует `Last-Modified` HTTP заголовок на основе времени последнего сообщения. Когда браузер в первый раз посещает страницу `index`, то страница будет сгенерирована на сервере и отправлена в браузер; если браузер снова зайдёт на эту страницу и с тех пор ни один пост не обновится, то сервер не будет пересоздавать страницу и браузер будет использовать закэшированную на стороне клиента версию. В результате, будет пропущено как создание страницы на стороне сервера, так и передача содержания страницы клиенту.

10.5.2 Заголовок ETag

Заголовок “Entity Tag” (или коротко **ETag**) используется для передачи хэша содержания страницы. Если страница была изменена, то хэш страницы тоже изменится. Сравнивая хэш на стороне клиента с хэшем, генерируемым на стороне сервера, кэш может определить, была ли страница изменена и требуется ли её передавать заново.

Вы можете настроить свойство `yii\filters\HttpCache::$etagSeed`, чтобы включить передачу заголовка **ETag**. Свойство должно содержать РНР-функцию, возвращающий `seed` для генерации **ETag** хэша. Сигнатура РНР-функции должна совпадать со следующей,

```

/**
 * @param Action $action объект действия, которое в настоящее время
 *                   обрабатывается
 * @param array $params значение свойства "params"
 * @return string строка используемая как seed для генерации ETag хэша
 */
function ($action, $params)

```

Ниже приведён пример использования заголовка **ETag**:

```

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]);
        },
    ],
];
}

```

Приведенный выше код устанавливает, что HTTP кэширование должно быть включено только для действия `view`. Он генерирует ETag HTTP заголовок на основе заголовка и содержания последнего сообщения. Когда браузер в первый раз посещает страницу `view`, то страница будет сгенерирована на сервере и отправлена в браузер; если браузер снова зайдёт на эту страницу и с тех пор ни один пост не обновится, то сервер не будет пересоздавать страницу и браузер будет использовать закэшированную на стороне клиента версию. В результате, будет пропущено как создание страницы на стороне сервера, так и передача содержания страницы клиенту.

ETags позволяет применять более сложные и/или более точные стратегии кэширования, чем заголовок `Last-Modified`. Например, ETag станет невалидным (некорректным), если на сайте была включена другая тема

Ресурсоёмкая генерация ETag может противоречить цели использования `HttpCache` и внести излишнюю нагрузку, т.к. он должен пересоздаваться при каждом запросе. Попробуйте найти простое выражение, которое инвалидирует кэш, если содержание страницы было изменено.

Примечание: В соответствии с RFC 7232¹¹, `HttpCache` будет отправлять как ETag заголовок, так и `Last-Modified` заголовок, если они оба были настроены. И если клиент отправляет как `If-None-Match` заголовок, так и `If-Modified-Since` заголовок, то только первый из них будет принят.

10.5.3 Заголовок `Cache-Control`

Заголовок `Cache-Control` определяет общую политику кэширования страниц. Вы можете включить его отправку, настроив свойство `yii\filters\HttpCache::$cacheControlHeader`. По-умолчанию будет отправлен следующий заголовок:

```
Cache-Control: public, max-age=3600
```

¹¹<http://tools.ietf.org/html/rfc7232#section-2.4>

10.5.4 Ограничитель кэша сессий

Когда на странице используются сессии, PHP автоматически отправляет некоторые связанные с кэшем HTTP заголовки, определённые в настройке `session.cache_limiter` в `php.ini`. Эти заголовки могут вмешиваться или отключать кэширование, которое вы ожидаете от `HttpCache`. Чтобы предотвратить эту проблему, по умолчанию `HttpCache` будет автоматически отключать отправку этих заголовков. Если вы хотите изменить это поведение, вы должны настроить свойство `yii\filters\HttpCache::$sessionCacheLimiter`. Это свойство может принимать строковое значение, включая `public`, `private`, `private_no_expire` и `nocache`. Пожалуйста, обратитесь к руководству PHP о `session_cache_limiter()`¹² для объяснения этих значений.

10.5.5 SEO подтекст

Поисковые боты, как правило, с уважением относятся к заголовкам кэширования. Поскольку некоторые из поисковых систем имеют ограничение на количество страниц для одного домена, которые они обрабатывают в течение определенного промежутка времени, то предоставление заголовков кэширования может помочь индексации, поскольку будет уменьшено число обрабатываемых страниц.

¹²<http://www.php.net/manual/en/function.session-cache-limiter.php>

Глава 11

Веб-сервисы REST

11.1 Быстрый старт

Yii включает полноценный набор средств для упрощённой реализации RESTful API¹. В частности это следующие возможности:

- Быстрое создание прототипов с поддержкой распространенных API к Active Record;
- Настройка формата ответа (JSON и XML реализованы по умолчанию);
- Получение сериализованных объектов с нужной вам выборкой полей;
- Надлежащее форматирование данных и ошибок при их валидации;
- Поддержка HATEOAS²;
- Эффективная маршрутизация с надлежащей проверкой HTTP методов;
- Встроенная поддержка методов `OPTIONS` и `HEAD`;
- Аутентификация и авторизация;
- HTTP кэширование и кэширование данных;
- Настройка ограничения для частоты запросов (Rate limiting);

Рассмотрим пример, как можно настроить Yii под RESTful API, приложив при этом минимум усилий.

Предположим, вы захотели RESTful API для данных по пользователям. Эти данные хранятся в базе данных и для работы с ними вами была ранее создана модель `ActiveRecord` (класс `app\models\User`).

11.1.1 Создание контроллера

Во-первых, создадим класс контроллера `app\controllers\UserController`:

```
namespace app\controllers;
```

¹<https://ru.wikipedia.org/wiki/REST>

²<http://en.wikipedia.org/wiki/HATEOAS>

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

Класс контроллера наследуется от `yii\rest\ActiveController`. Мы задали `modelClass` как `app\models\User`, тем самым указав контроллеру, к какой модели ему необходимо обращаться для редактирования или выборки данных.

11.1.2 Настройка правил URL

Далее изменим настройки компонента `urlManager` в конфигурации приложения:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

Настройки выше добавляет правило для контроллера `user`, которое предоставляет доступ к данным пользователя через красивые URL и логичные глаголы HTTP.

11.1.3 Пробуем

Вот так просто мы и создали RESTful API для доступа к данным пользователя. API нашего сервиса, сейчас включает в себя:

- GET `/users`: получение постранично списка всех пользователей;
- HEAD `/users`: получение метаданных листинга пользователей;
- POST `/users`: создание нового пользователя;
- GET `/users/123`: получение информации по конкретному пользователю с id равным 123;
- HEAD `/users/123`: получение метаданных по конкретному пользователю с id равным 123;
- PATCH `/users/123` и PUT `/users/123`: изменение информации по пользователю с id равным 123;
- DELETE `/users/123`: удаление пользователя с id равным 123;
- OPTIONS `/users`: получение поддерживаемых методов, по которым можно обратиться к `/users`;
- OPTIONS `/users/123`: получение поддерживаемых методов, по которым можно обратиться к `/users/123`.

Информация: Yii автоматически использует множественное число от имени контроллера в URL.

Пробуем получить ответы по API используя curl:

```
$ curl -i -H "Accept:application/json" "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

[
  {
    "id": 1,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
]
```

Попробуйте изменить заголовок допустимого формата ресурса на `application/xml` и в ответ вы получите результат в формате XML:

```
$ curl -i -H "Accept:application/xml" "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/xml

<?xml version="1.0" encoding="UTF-8"?>
<response>
```

```
<item>
  <id>1</id>
  ...
</item>
<item>
  <id>2</id>
  ...
</item>
...
</response>
```

Подсказка: Вы можете получить доступ к API через веб-браузер, введя адрес `http://localhost/users`. Но в этом случае, для передачи определённых заголовков вам, скорее всего, потребуются дополнительные плагины для браузера.

Если внимательно посмотреть результат ответа, то можно обнаружить, что в заголовках есть информация об общем числе записей, количестве страниц и т.д. Тут так же можно обнаружить ссылки на другие страницы, как, например, `http://localhost/users?page=2`. Перейдя по ней можно получить вторую страницу данных пользователей.

Используя параметры `fields` и `expand` в URL, можно указать, какие поля должны быть включены в результат. Например, по адресу `http://localhost/users?fields=id,email` мы получим информацию по пользователям, которая будет содержать только `id` и `email`.

Информация: Вы наверняка заметили, что при обращении к `http://localhost/users` мы получаем информацию с полями, которые нежелательно показывать, такими как `password_hash` и `auth_key`. Вы можете и должны отфильтровать их как описано в разделе «Форматирование ответа».

11.1.4 Резюме

Используя Yii в качестве RESTful API фреймворка, мы используем реализацию точек входа API как действия контроллеров. Контроллер используется для организации действий, которые относятся к определённому типу ресурса.

Ресурсы представлены в виде моделей данных, которые наследуются от класса `yii\base\Model`. Если необходима работа с базами данных (как с реляционными, так и с NoSQL), рекомендуется использовать для представления ресурсов `ActiveRecord`.

Вы можете использовать `yii\rest\UrlRule` для упрощения маршрутизации точек входа API.

Хоть это не обязательно, рекомендуется отделять RESTful APIs приложение от основного веб-приложения. Такое разделение легче обслуживается.

11.2 Ресурсы

RESTful API строятся вокруг доступа к *ресурсам* и управления ими. Вы можете думать о ресурсах как о моделях из MVC³.

Хотя не существует никаких ограничений на то, как представить ресурс, в Yii ресурсы обычно представляются как объекты `yii\base\Model` или дочерних классов (например `yii\db\ActiveRecord`), потому как:

- `yii\base\Model` реализует интерфейс `yii\base\Arrayable`, который позволяет задать способ отдачи данных ресурса через RESTful API.
- `yii\base\Model` поддерживает валидацию, что полезно для RESTful API реализующего ввод данных.
- `yii\db\ActiveRecord` даёт мощную поддержку работы с БД, что актуально если данные ресурса хранятся в ней.

В этом разделе, мы сосредоточимся на том, как при помощи класса ресурса, наследуемого от `yii\base\Model` (или дочерних классов) задать какие данные будут возвращаться RESTful API. Если класс ресурса не наследуется от `yii\base\Model`, возвращаются всего его `public` свойства.

11.2.1 Поля

Когда ресурс включается в ответ RESTful API, необходимо сериализовать его в строку. Yii разбивает этот процесс на два этапа. Сначала ресурс конвертируется в массив при помощи `yii\rest\Serializer`. На втором этапе массив сериализуется в строку заданного формата (например, JSON или XML) при помощи **форматтера ответа**. Именно на этом стоит сосредоточиться при разработке класса ресурса.

Вы можете указать какие данные включать в представление ресурса в виде массива путём переопределения методов `fields()` и/или `extraFields()`. Разница между ними в том, что первый определяет набор полей, которые всегда будут включены в массив, а второй определяет дополнительные поля, которые пользователь может запросить через параметр `expand`:

```
// вернёт все поля объявленные в fields()
http://localhost/users

// вернёт только поля id и email, если они объявлены в методе fields()
http://localhost/users?fields=id,email

// вернёт все поля объявленные в fields() и поле profile если оно указано в
extraFields()
http://localhost/users?expand=profile

// вернёт только id, email и profile, если они объявлены в fields() и
extraFields()
http://localhost/users?fields=id,email&expand=profile
```

³<http://ru.wikipedia.org/wiki/Model-View-Controller>

Переопределение `fields()`

По умолчанию, `yii\base\Model::fields()` возвращает все атрибуты модели как поля, а `yii\db\ActiveRecord::fields()` возвращает только те атрибуты, которые были объявлены в схеме БД.

Вы можете переопределить `fields()` для того, чтобы добавить, удалить, переименовать или переобъявить поля. Значение, возвращаемое `fields()`, должно быть массивом. Его ключи это имена полей, и значения могут быть либо именами свойств/атрибутов, либо анонимными функциями, которые возвращают значение соответствующих полей. Если имя атрибута такое же, как ключ массива вы можете опустить значение:

```
// явное перечисление всех атрибутов лучше всего использовать когда вы
// хотите быть уверенным что изменение
// таблицы БД или атрибутов модели не повлияет на изменение полей,
// отдаваемых API что( важно для поддержки обратной
// совместимости API).
public function fields()
{
    return [
        // название поля совпадает с названием атрибута
        'id',
        // имя поля "email", атрибут "email_address"
        'email' => 'email_address',
        // имя поля "name", значение определяется callback-ом PHP
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// отбрасываем некоторые поля. Лучше всего использовать в случае
// наследования
public function fields()
{
    $fields = parent::fields();

    // удаляем не безопасные поля
    unset($fields['auth_key'], $fields['password_hash'], $fields['password_reset_token']);

    return $fields;
}
```

Внимание: По умолчанию все атрибуты модели будут включены в ответы API. Вы должны убедиться в том, что отдаются только безопасные данные. В противном случае для исключения небезопасных полей необходимо переопределить метод `fields()`. В приведённом выше примере мы исключаем `auth_key`, `password_hash` и `password_reset_token`.

Переопределение `extraFields()`

По умолчанию, `yii\base\Model::extraFields()` ничего не возвращает, а `yii\db\ActiveRecord::extraFields()` возвращает названия заданных в БД связей.

Формат возвращаемых `extraFields()` данных такой же как у `fields()`. Как правило, `extraFields()` используется для указания полей, значения которых являются объектами. Например учитывая следующее объявление полей

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

запрос `http://localhost/users?fields=id,email&expand=profile` может возвращать следующие JSON данные:

```
[
  {
    "id": 100,
    "email": "100@example.com",
    "profile": {
      "id": 100,
      "age": 30,
    }
  },
  ...
]
```

11.2.2 Ссылки

Согласно HATEOAS⁴, расшифровывающемуся как Hypermedia as the Engine of Application State, RESTful API должны возвращать достаточно информации для того, чтобы клиенты могли определить возможные действия над ресурсами. Ключевой момент HATEOAS заключается том, чтобы возвращать вместе с данными набора гиперссылок, указывающих на связанную с ресурсом информацию.

Поддержку HATEOAS в ваши классы ресурсов можно добавить реализовав интерфейс `yii\web\Linkable`. Этот интерфейс содержит единственный метод `getLinks()`, который возвращает список **ссылок**. Обычно вы должны вернуть хотя бы ссылку `self` с URL самого ресурса:

⁴<http://en.wikipedia.org/wiki/HATEOAS>

```

use yii\db\ActiveRecord;
use yii\web\Link;
use yii\web\Linkable;
use yii\helpers\Url;

class User extends ActiveRecord implements Linkable
{
    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id], true),
        ];
    }
}

```

При отправке ответа объект `User` содержит поле `_links`, значение которого — ссылки, связанные с объектом: ‘ {

```

"id": 100,
"email": "user@example.com",
// ...
"_links" => {
    "self": {
        "href": "https://example.com/users/100"
    }
}
} ,

```

11.2.3 Коллекции

Объекты ресурсов могут группироваться в *коллекции*. Каждая коллекция содержит список объектов ресурсов одного типа.

Несмотря на то, что коллекции можно представить в виде массива, удобнее использовать провайдеры данных так как они поддерживают сортировку и分页ную разбивку. Для RESTful APIs, которые работают с коллекциями, данные возможности используются довольно часто. Например, следующее действие контроллера возвращает провайдер данных для ресурса постов:

```

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(),
        ]);
    }
}

```

```
    });  
  }  
}
```

При отправке ответа RESTful API, `yii\rest\Serializer` сериализует массив объектов ресурсов для текущей страницы. Кроме того, он добавит HTTP заголовки, содержащие информацию о страницах:

- **X-Pagination-Total-Count**: общее количество ресурсов;
- **X-Pagination-Page-Count**: количество страниц;
- **X-Pagination-Current-Page**: текущая страница (начиная с 1);
- **X-Pagination-Per-Page**: количество ресурсов на страницу;
- **Link**: набор ссылок, позволяющий клиенту пройти все страницы ресурсов.

Примеры вы можете найти в разделе «быстрый старт».

11.3 Контроллеры

После создания классов ресурсов и настройки способа форматирования ресурсных данных следующим шагом является создание действий контроллеров для предоставления ресурсов конечным пользователям через RESTful API.

В Yii есть два базовых класса контроллеров для упрощения вашей работы по созданию RESTful-действий: `yii\rest\Controller` и `yii\rest\ActiveController`. Разница между этими двумя контроллерами в том, что у последнего есть набор действий по умолчанию, который специально создан для работы с ресурсами, представленными Active Record. Так что если вы используете Active Record и вас устраивает предоставленный набор встроенных действий, вы можете унаследовать классы ваших контроллеров от `yii\rest\ActiveController`, что позволит вам создать полноценные RESTful API, написав минимум кода.

`yii\rest\Controller` и `yii\rest\ActiveController` имеют следующие возможности, некоторые из которых будут подробно описаны в следующих разделах:

- Проверка HTTP-метода;
- Согласование содержимого и форматирование данных;
- Аутентификация;
- Ограничение частоты запросов.

`yii\rest\ActiveController`, кроме того, предоставляет следующие возможности:

- Набор наиболее часто используемых действий: `index`, `view`, `create`, `update`, `delete` и `options`;
- Авторизация пользователя для запрашиваемых действия и ресурса.

11.3.1 Создание классов контроллеров

При создании нового класса контроллера в имени класса обычно используется название типа ресурса в единственном числе. Например, контроллер, отвечающий за предоставление информации о пользователях, можно назвать `UserController`.

Создание нового действия похоже на создание действия для Web-приложения. Единственное отличие в том, что в RESTful-действиях вместо рендера результата в представлении с помощью вызова метода `render()` вы просто возвращаете данные. Выполнение преобразования исходных данных в запрошенный формат ложится на сериализатор и объект ответа. Например:

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2 Фильтры

Большинство возможностей RESTful API, предоставляемых `yii\rest\Controller`, реализовано на основе фильтров. В частности, следующие фильтры будут выполняться в том порядке, в котором они перечислены:

- `contentNegotiator`: обеспечивает согласование содержимого, более подробно описан в разделе [Форматирование ответа](#);
- `verbFilter`: обеспечивает проверку HTTP-метода;
- `yii\filters\AuthMethod`: обеспечивает аутентификацию пользователя, более подробно описан в разделе [Аутентификация](#);
- `rateLimiter`: обеспечивает ограничение частоты запросов, более подробно описан в разделе [Ограничение частоты запросов](#).

Эти именованные фильтры объявлены в методе `behaviors()`. Вы можете переопределить этот метод для настройки отдельных фильтров, отключения каких-либо из них или для добавления ваших собственных фильтров. Например, если вы хотите использовать только базовую HTTP-аутентификацию, вы можете написать такой код:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```


11.3.3 Наследование от `ActiveController`

Если ваш класс контроллера наследуется от `yii\rest\ActiveController`, вам следует установить значение его свойства `modelClass` равным имени класса ресурса, который вы планируете обслуживать с помощью этого контроллера. Класс ресурса должен быть унаследован от `yii\db\ActiveRecord`.

Настройка действий

По умолчанию `yii\rest\ActiveController` предоставляет набор из следующих действий:

- **index**: постраничный список ресурсов;
- **view**: возвращает подробную информацию об указанном ресурсе;
- **create**: создание нового ресурса;
- **update**: обновление существующего ресурса;
- **delete**: удаление указанного ресурса;
- **options**: возвращает поддерживаемые HTTP-методы.

Все эти действия объявляются в методе `actions()`. Вы можете настроить эти действия или отключить какие-то из них, переопределив метод `actions()`, как показано ниже:

```
public function actions()
{
    $actions = parent::actions();

    // отключить действия "delete" и "create"
    unset($actions['delete'], $actions['create']);

    // настроить подготовку провайдера данных с помощью метода "
    prepareDataProvider()"
    $actions['index']['prepareDataProvider'] = [$this, 'prepareDataProvider'];

    return $actions;
}

public function prepareDataProvider()
{
    // подготовить и вернуть провайдер данных для действия "index"
}
```

Чтобы узнать, какие опции доступны для настройки классов отдельных действий, обратитесь к соответствующим разделам справочника классов.

Выполнение контроля доступа

При предоставлении ресурсов через RESTful API часто бывает нужно проверять, имеет ли текущий пользователь разрешение на доступ к за-

прошенному ресурсу (или ресурсам) и манипуляцию им (или ими). Для `yii\rest\ActiveController` эта задача может быть решена переопределением метода `checkAccess()` следующим образом:

```
/**
 * Проверяет права текущего пользователя.
 *
 * Этот метод должен быть переопределен, чтобы проверить, имеет ли текущий
 * пользователь
 * право выполнения указанного действия над указанной моделью данных.
 * Если у пользователя нет доступа, следует выбросить исключение [[
 * ForbiddenHttpException]].
 *
 * @param string $action ID действия, которое надо выполнить
 * @param \yii\base\Model $model модель, к которой нужно получить доступ.
 * Если null, это означает, что модель, к которой нужно получить доступ,
 * отсутствует.
 * @param array $params дополнительные параметры
 * @throws ForbiddenHttpException если у пользователя нет доступа
 */
public function checkAccess($action, $model = null, $params = [])
{
    // проверить, имеет ли пользователь доступ к $action и $model
    // выбросить ForbiddenHttpException, если доступ следует запретить
}
```

Метод `checkAccess()` будет вызван действиями по умолчанию контроллера `yii\rest\ActiveController`. Если вы создаёте новые действия и хотите в них выполнять контроль доступа, вы должны вызвать этот метод явно в своих новых действиях.

Подсказка: вы можете реализовать метод `checkAccess()` с помощью “Контроля доступа на основе ролей” (RBAC).

11.4 Маршрутизация

Имея готовые классы ресурсов и контроллеров, можно получить доступ к ресурсам, используя URL вроде `http://localhost/index.php?r=user/create`, подобно тому, как вы это делаете с обычными Web-приложениями.

На деле вам обычно хочется включить «красивые» URL-адреса и использовать все преимущества HTTP-методов (HTTP-verbs). Например, чтобы запрос `POST /users` означал обращение к действию `user/create`. Это может быть легко сделано с помощью настройки компонента приложения `urlManager` в конфигурации приложения следующим образом:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
```

```
'rules' => [
    ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
],
]
```

Главная новинка в коде выше по сравнению с управлением URL-адресами в Web-приложениях состоит в использовании `yii\rest\UrlRule` для маршрутизации запросов к RESTful API. Этот особый класс URL-правил будет создавать целый набор дочерних URL-правил для поддержки маршрутизации и создания URL-адресов для указанного контроллера (или контроллеров). Например, приведенный выше код является приближенным аналогом следующего набора правил:

```
[
    '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',
]
```

Этим правилом поддерживаются следующие точки входа в API:

- GET /users: разбитый на страницы список всех пользователей;
- HEAD /users: общая информация по списку пользователей;
- POST /users: создание нового пользователя;
- GET /users/123: подробная информация о пользователе 123;
- HEAD /users/123: общая информация о пользователе 123;
- PATCH /users/123 и PUT /users/123: обновление пользователя 123;
- DELETE /users/123: удаление пользователя 123;
- OPTIONS /users: список HTTP-методов, поддерживаемые точкой входа /users;
- OPTIONS /users/123: список HTTP-методов, поддерживаемые точкой входа /users/123.

Вы можете настроить опции `only` и `except`, явно указав для них список действий, которые поддерживаются или которые должны быть отключены, соответственно. Например:

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

Вы также можете настроить опции `patterns` или `extraPatterns` для переопределения существующих шаблонов или добавления новых шаблонов, поддерживаемых этим правилом. Например, для включения нового действия `search` в точке входа GET /users/search настройте опцию `extraPatterns` следующим образом:

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
]
```

Как вы могли заметить, ID контроллера `user` в этих точках входа используется в форме множественного числа (как `users`). Это происходит потому, что `yii\rest\UrlRule` автоматически приводит идентификаторы контроллеров к множественной форме. Вы можете отключить такое поведение, назначив свойству `yii\rest\UrlRule::$pluralize` значение `false`.

Информация: Приведение ID контроллера к множественной форме производится в методе `yii\helpers\Inflector::pluralize()`. При этом соблюдаются правила английского языка. Например, `box` будет преобразован в `boxes`, а не в `boxs`.

В том случае, если автоматическое приведение к множественному числу вам не подходит, вы можете настроить свойство `yii\rest\UrlRule::$controller`, где указать явное соответствие имени в URL и ID контроллера. Например, код ниже ставит в соответствие имя `u` и ID контроллера `user`.

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

11.5 Форматирование ответа

При обработке RESTful API запросов приложение обычно выполняет следующие шаги, связанные с форматированием ответа:

1. Определяет различные факторы, которые могут повлиять на формат ответа, такие как `media type`, язык, версия и т.д. Этот процесс также известен как согласование содержимого⁵.
2. Конвертирует объекты ресурсов в массивы, как описано в секции Ресурсы. Этим занимается `yii\rest\Serializer`.
3. Конвертирует массивы в строки исходя из формата, определенного на этапе согласования содержимого. Это задача для **формatera ответов**, регистрируемого с помощью компонента приложения `response`.

⁵http://en.wikipedia.org/wiki/Content_negotiation

11.5.1 Согласование содержимого

Yii поддерживает согласование содержимого с помощью фильтра `yii\filters\ContentNegotiator`. Базовый класс контроллера RESTful API - `yii\rest\Controller` - использует этот фильтр под именем `contentNegotiator`. Фильтр обеспечивает соответствие формата ответа и определяет используемый язык. Например, если RESTful API запрос содержит следующий заголовок:

```
Accept: application/json; q=1.0, */*; q=0.1
```

Он получит ответ в JSON-формате такого вида:

```
$ 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

[
  {
    "id": 1,
    ...
  },
  {
    "id": 2,
    ...
  },
  ...
]
```

Под капотом происходит следующее: прежде, чем *действие* RESTful API контроллера будет выполнено, фильтр `yii\filters\ContentNegotiator` проверит HTTP-заголовок `Accept` в запросе и установит, что `формат ответа` должен быть в `'json'`. После того, как *действие* будет выполнено и вернет итоговый объект ресурса или коллекцию, `yii\rest\Serializer` конвертирует результат в массив. И, наконец, `yii\web\JsonResponseFormatter` сериализует массив в строку в формате JSON и включит ее в тело ответа.

По умолчанию, RESTful API поддерживает и JSON, и XML форматы. Для того, чтобы добавить поддержку нового формата, вы долж-

ны установить свою конфигурацию для свойства `formats` у фильтра `contentNegotiator`, например, с использованием поведения такого вида:

```
use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['contentNegotiator']['formats']['text/html'] = Response::
        FORMAT_HTML;
    return $behaviors;
}
```

Ключи свойства `formats` - это поддерживаемые MIME-типы, а их значения должны соответствовать именам форматов ответа, которые установлены в `yii\web\Response::$formatters`.

11.5.2 Сериализация данных

Как уже описывалось выше, `yii\rest\Serializer` - это центральное место, отвечающее за конвертацию объектов ресурсов или коллекций в массивы. Он реализует интерфейсы `yii\base\ArrayableInterface` и `yii\data\DataProviderInterface`. Для объектов ресурсов как правило реализуется интерфейс `yii\base\ArrayableInterface`, а для коллекций - `yii\data\DataProviderInterface`.

Вы можете переконфигурировать сериализатор с помощью настройки свойства `yii\rest\Controller::$serializer`, используя конфигурационный массив. Например, иногда вам может быть нужно помочь упростить разработку клиентской части приложения с помощью добавления информации о пагинации непосредственно в тело ответа. Чтобы сделать это, переконфигурируйте свойство `yii\rest\Serializer::$collectionEnvelope` следующим образом:

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

Тогда вы можете получить следующий ответ на запрос `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

{
  "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
  }
}
```

11.6 Аутентификация

В отличие от Web-приложений, RESTful API обычно не сохраняют информацию о состоянии, а это означает, что сессии и куки использовать не следует. Следовательно, раз состояние аутентификации пользователя не может быть сохранено в сессиях или куках, каждый запрос должен приходить вместе с определенным видом параметров аутентификации. Общепринятая практика состоит в том, что для аутентификации пользователя с каждым запросом отправляется секретный токен доступа. Так

как токен доступа может использоваться для уникальной идентификации и аутентификации пользователя, **запросы к API всегда должны отправляться через протокол HTTPS, чтобы предотвратить атаки «человек посередине»** (англ. “man-in-the-middle”, MitM).

Есть различные способы отправки токена доступа:

- HTTP Basic Auth⁶: токен доступа отправляется как имя пользователя. Такой подход следует использовать только в том случае, когда токен доступа может быть безопасно сохранен на стороне абонента API. Например, если API используется программой, запущенной на сервере.
- Параметр запроса: токен доступа отправляется как параметр запроса в URL-адресе API, т.е. примерно таким образом: `https://example.com/users?access-token=xxxxxxx`. Так как большинство Web-серверов сохраняют параметры запроса в своих логах, такой подход следует применять только при работе с JSONP-запросами, которые не могут отправлять токены доступа в HTTP-заголовках.
- OAuth 2⁷: токен доступа выдается абоненту API сервером авторизации и отправляется API-серверу через HTTP Bearer Tokens⁸, в соответствии с протоколом OAuth2.

Yii поддерживает все выше перечисленные методы аутентификации. Вы также можете легко создавать новые методы аутентификации.

Чтобы включить аутентификацию для ваших API, выполните следующие шаги:

1. У компонента приложения `user` установите свойство `enableSession` равным `false`.
2. Укажите, какие методы аутентификации вы планируете использовать, настроив поведение `authenticator` в ваших классах REST-контроллеров.
3. Реализуйте метод `yii\web\IdentityInterface::findIdentityByAccessToken()` в вашем классе `UserIdentity`.

Шаг 1 не обязателен, но рекомендуется его всё-таки выполнить, так как RESTful API не должен сохранять информацию о состоянии клиента. Когда свойство `enableSession` установлено в `false`, состояние аутентификации пользователя НЕ БУДЕТ сохраняться между запросами с использованием сессий. Вместо этого аутентификация будет выполняться для каждого запроса, что достигается шагами 2 и 3.

⁶http://en.wikipedia.org/wiki/Basic_access_authentication

⁷<http://oauth.net/2/>

⁸<http://tools.ietf.org/html/rfc6750>

Подсказка: если вы разрабатываете RESTful API в пределах приложения, вы можете настроить свойство `enableSession` компонента приложения `user` в конфигурации приложения. Если вы разрабатываете RESTful API как модуль, можете добавить следующую строчку в метод `init()` модуля: ‘php public function init() {

```
parent::init();
\Yii::$app->user->enableSession = false;

} ‘
```

Например, для использования HTTP Basic Auth, вы можете настроить свойство `authenticator` следующим образом:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```

Если вы хотите включить поддержку всех трёх описанных выше методов аутентификации, можете использовать `CompositeAuth`:

```
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;
}
```

Каждый элемент в массиве `authMethods` должен быть названием класса метода аутентификации или массивом настроек.

Реализация метода `findIdentityByAccessToken()` определяется особенностями приложения. Например, в простом варианте, когда у каждого пользователя есть только один токен доступа, вы можете хранить этот

токен в поле `access_token` таблицы пользователей. В этом случае метод `findIdentityByAccessToken()` может быть легко реализован в классе `User` следующим образом:

```
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]);
    }
}
```

После включения аутентификации описанным выше способом при каждом запросе к API запрашиваемый контроллер будет пытаться аутентифицировать пользователя в своем методе `beforeAction()`.

Если аутентификация прошла успешно, контроллер выполнит другие проверки (ограничение частоты запросов, авторизация) и затем выполнит действие. Информация об аутентифицированном пользователе может быть получена из объекта `Yii::$app->user->identity`.

Если аутентификация прошла неудачно, будет возвращен ответ с HTTP-кодом состояния 401 вместе с другими необходимыми заголовками (такими, как заголовок `WWW-Authenticate` для HTTP Basic Auth).

11.6.1 Авторизация

После аутентификации пользователя вы, вероятно, захотите проверить, есть ли у него или у неё разрешение на выполнение запрошенного действия с запрошенным ресурсом. Этот процесс называется *авторизацией* и подробно описан в разделе «Авторизация».

Если ваши контроллеры унаследованы от `yii\rest\ActiveController`, вы можете переопределить метод `yii\rest\Controller::checkAccess()` для выполнения авторизации. Этот метод будет вызываться встроенными действиями, предоставляемыми контроллером `yii\rest\ActiveController`.

11.7 Ограничение частоты запросов

Чтобы избежать злоупотреблений, вам следует подумать о добавлении ограничения частоты запросов к вашим API. Например, вы можете ограничить использование API 100 вызовов API в течение 10 минут для каждого пользователя. Если от пользователя в течение этого периода времени приходит большее количество запросов, будет возвращаться ответ с кодом состояния 429 («слишком много запросов»).

Чтобы включить ограничение частоты запросов, класс `user identity` должен реализовывать интерфейс `yii\filters\RateLimitInterface`. Этот

интерфейс требует реализации следующих трех методов:

- `getRateLimit()`: возвращает максимальное количество разрешенных запросов и период времени, например `[100, 600]`, что означает не более 100 вызовов API в течение 600 секунд.
- `loadAllowance()`: возвращает оставшееся количество разрешенных запросов и *UNIX-timestamp* последней проверки ограничения.
- `saveAllowance()`: сохраняет оставшееся количество разрешенных запросов и текущий *UNIX-timestamp*.

Вы можете использовать два столбца в таблице `user` для хранения количества разрешённых запросов и времени последней проверки. В методах `loadAllowance()` и `saveAllowance()` можно реализовать чтение и сохранение значений этих столбцов в соответствии с данными текущего аутентифицированного пользователя. Для улучшения производительности можно попробовать хранить эту информацию в кэше или NoSQL хранилище.

Как только соответствующий интерфейс будет реализован в классе `identity`, Yii начнёт автоматически проверять ограничения частоты запросов при помощи `yii\filters\RateLimiter`, фильтра действий для `yii\rest\Controller`. При превышении ограничений будет выброшено исключение `yii\web\TooManyRequestsHttpException`.

Вы можете настроить ограничитель частоты запросов в ваших классах REST-контроллеров следующим образом:

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;
    return $behaviors;
}
```

При включенном ограничении частоты запросов каждый ответ, по умолчанию, возвращается со следующими HTTP-заголовками, содержащими информацию о текущих ограничениях:

- **X-Rate-Limit-Limit**: максимальное количество запросов, разрешённое в течение периода времени;
- **X-Rate-Limit-Remaining**: оставшееся количество разрешённых запросов в текущем периоде времени;
- **X-Rate-Limit-Reset**: количество секунд, которое нужно подождать до получения максимального количества разрешённых запросов.

Вы можете отключить эти заголовки, установив свойство `yii\filters\RateLimiter::$enableRateLimitHeaders` в `false`, как показано в примере кода выше.

11.8 Версионирование

Хороший API должен быть *версионирован*: изменения и новые возможности реализуются в новых версиях API, а не в одной и той же версии.

В отличие от Web-приложений, где у вас есть полный контроль и над серверным, и над клиентским кодом, API используются клиентами, код которых вы не контролируете. Поэтому, обратная совместимость (ВС) должна по возможности сохраняться. Если ломающее её изменение необходимо, делать его нужно в новой версии API. Существующие клиенты могут продолжать использовать старую, совместимую с ними версию API. Новые или обновлённые клиенты могут использовать новую версию.

Общей практикой при реализации версионирования API является включение номера версии в URL-адрес вызова API-метода. Например, `http://example.com/v1/users` означает вызов API `/users` версии 1. Другой способ версионирования API, получивший недавно широкое распространение, состоит в добавлении номера версии в HTTP-заголовки запроса, обычно в заголовок `Accept`:

Подсказка: Чтобы узнать больше о выборе версий обратитесь к Semantic Versioning⁹.

Один из типичных способов реализации версионирования — указание версии в URL. Например, `http://example.com/v1/users` соответствует `/users` версии 1.

Ещё один способ, ставший сейчас популярным — передача версии через заголовок HTTP. Чаще всего для этого используется заголовок `Accept`:

```
// как параметр
Accept: application/json; version=v1
// как тип содержимого, определенный поставщиком API
Accept: application/vnd.company.myapp-v1+json
```

Оба способа имеют достоинства и недостатки, и вокруг них много споров. Ниже мы опишем реально работающую стратегию версионирования API, которая является некоторой смесью этих двух способов:

- Помещать каждую мажорную версию реализации API в отдельный модуль, чей ID является номером мажорной версии (например, `v1`, `v2`). Естественно, URL-адреса API будут содержать в себе номера мажорных версий.
- В пределах каждой мажорной версии (т.е. внутри соответствующего модуля) использовать HTTP-заголовок `Accept` для определения номера минорной версии и писать условный код для соответствующих минорных версий.

В каждый модуль, обслуживающий мажорную версию, следует включать классы ресурсов и контроллеров, обслуживающих эту конкретную

⁹<http://semver.org/>

версию. Для лучшего разделения ответственности кода вы можете составить общий набор базовых классов ресурсов и контроллеров, и субклассировать их в каждом отдельно взятом модуле версии. Внутри дочерних классов реализуйте конкретный код вроде метода `Model::fields()`.

Ваш код может быть организован примерно следующим образом:

```
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
```

Конфигурация вашего приложения могла бы выглядеть так:

```
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']],  
            ],  
        ],  
    ],  
]
```

```
    ],
    ],
];
```

В результате `http://example.com/v1/users` возвратит список пользователей API версии 1, в то время как `http://example.com/v2/users` вернет список пользователей версии 2.

Благодаря использованию модулей код API различных мажорных версий может быть хорошо изолирован. И по-прежнему возможно повторное использование кода между модулями через общие базовые классы и другие разделяемые классы.

Для работы с минорными номерами версий вы можете использовать преимущества согласования содержимого, предоставляемого поведением `contentNegotiator`. Определив тип поддерживаемого содержимого, поведение `contentNegotiator` установит значение свойства `yii\web\Response::$acceptParams`.

Например, если запрос посылается с HTTP-заголовком `Accept: application/json; version=v1`, то после согласования содержимого свойство `yii\web\Response::$acceptParams` будет содержать значение `['version' => 'v1']`.

На основе информации о версии из `acceptParams` вы можете выбирать поведение в действиях, классах ресурсов, сериализаторах и т.д.

Так как минорные версии требуют поддержания обратной совместимости, будем надеяться, что в вашем коде не так уж много проверок на номер версии. В противном случае велики шансы, что вам нужна новая мажорная версия.

11.9 Обработка ошибок

Если при обработке запроса к RESTful API в запросе пользователя обнаруживается ошибка или происходит что-то непредвиденное на сервере, вы можете просто выбрасывать исключение, чтобы уведомить пользователя о нештатной ситуации. Если же вы можете установить конкретную причину ошибки (например, запрошенный ресурс не существует), вам следует подумать о том, чтобы выбрасывать исключение с соответствующим кодом состояния HTTP (например, `yii\web\NotFoundException`, соответствующее коду состояния 404). Yii отправит ответ с соответствующим HTTP-кодом и текстом. Он также включит в тело ответа сериализованное представление исключения. Например:

```
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
}
```

Сводный список кодов состояния HTTP, используемых REST-фреймворком Yii:

- 200: ОК. Все сработало именно так, как и ожидалось.
- 201: Ресурс был успешно создан в ответ на POST-запрос. Заголовок `Location` содержит URL, указывающий на только что созданный ресурс.
- 204: Запрос обработан успешно, и в ответе нет содержимого (для запроса `DELETE`, например).
- 304: Ресурс не изменялся. Можно использовать закешированную версию.
- 400: Неверный запрос. Может быть связано с разнообразными проблемами на стороне пользователя, такими как неверные JSON-данные в теле запроса, неправильные параметры действия, и т.д.
- 401: Аутентификация завершилась неудачно.
- 403: Аутентифицированному пользователю не разрешен доступ к указанной точке входа API.
- 404: Запрошенный ресурс не существует.
- 405: Метод не поддерживается. Сверьтесь со списком поддерживаемых HTTP-методов в заголовке `Allow`.
- 415: Неподдерживаемый тип данных. Запрашивается неправильный тип данных или номер версии.
- 422: Проверка данных завершилась неудачно (в ответе на POST-запрос, например). Подробные сообщения об ошибках смотрите в теле ответа.
- 429: Слишком много запросов. Запрос отклонен из-за превышения ограничения частоты запросов.
- 500: Внутренняя ошибка сервера. Возможная причина — ошибки в самой программе.

11.9.1 Свой формат ответа с ошибкой

Вам может понадобиться изменить формат ответа с ошибкой. Например, вместо использования разных статусов ответа HTTP для разных ошибок, вы можете всегда отдавать статус 200, а реальный код статуса отдавать как часть JSON ответа:

```
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
    }
}
```

Для этого можно использовать событие `beforeSend` компонента `response` прямо в конфигурации приложения:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null && !empty(Yii::$app->request->
                    get('suppress_response_code'))) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

Приведённый выше код изменит формат ответа (как для удачного запроса, так и для ошибок) если передан GET-параметр `suppress_response_code`.

Глава 12

Инструменты разработчика

Error: not existing file: tool-debugger.md

Error: not existing file: tool-gii.md

Error: not existing file: tool-api-doc.md

Глава 13

Тестирование

Error: not existing file: test-overview.md

Error: not existing file: test-unit.md

Error: not existing file: test-functional.md

Error: not existing file: test-acceptance.md

Error: not existing file: test-fixtures.md

Глава 14

Расширение Y_i

Error: not existing file: extend-creating-extensions.md

Error: not existing file: extend-customizing-core.md

Error: not existing file: extend-using-libs.md

Error: not existing file: extend-embedding-in-others.md

Error: not existing file: extend-using-v1-v2.md

Error: not existing file: extend-using-composer.md

Глава 15

Специальные темы

15.1 Шаблон приложения advanced

Примечание: Данная глава находится в разработке.

Этот шаблон предназначен для крупных проектов разрабатываемых в командах где администраторская часть (backend) отделена от приложения пользователя (frontend), приложения располагаются на нескольких серверах и т.д. Этот шаблон приложения включает значительное количество возможностей, таких как начальная схема базы данных, регистрация пользователя и восстановление его пароля.

15.1.1 Установка

Установка при помощи Composer

Если у вас ещё не установлен Composer¹, следуйте инструкциям в разделе установка Yii.

Если Composer установлен, вы можете установить приложение используя следующие команды:

```
composer global require "fxp/composer-asset-plugin:1.0.0"
composer create-project --prefer-dist yiisoft/yii2-app-advanced yii-
application
```

Первая команда установит плагин composer asset plugin², который позволит работать с пакетами bower и npm через Composer. Эту команду необходимо выполнить единожды. Вторая команда установит приложение advanced в директорию yii-application. Вы можете выбрать другое имя директорию если пожелаете.

¹<http://getcomposer.org/>

²<https://github.com/francoispluchino/composer-asset-plugin/>

15.1.2 Начало работы

После установки приложения, вам необходимо один раз выполнить приведённые ниже действия для того, чтобы инициализировать установленное приложение.

1. Выполните команду `init` и выберите окружение `dev`.

```
php /path/to/yii-application/init
```

Для производственных серверов удобно выполнять данную команду в неинтерактивном режиме.

```
php /path/to/yii-application/init --env=Production overwrite=All
```

2. Создайте новую базу данных и внесите соответствующие изменения в секцию `components.db` файла `common/config/main-local.php`.
3. Примените миграции при помощи консольной команды `yii migrate`.
4. Настройте на вебсервере URL и корневые директории:
 - для приложения frontend директория `/path/to/yii-application/frontend/web/` и URL `http://yourdomain/frontend/`
 - для приложения backend директория `/path/to/yii-application/backend/web/` и URL `http://yourdomain/backend/`

15.1.3 Структура директорий

Корневая директория содержит следующие поддиректории:

- **backend** - веб приложение администраторской части.
- **common** - общие файлы для всех приложений.
- **console** - приложение для консоли.
- **environments** - настройки для различных окружений.
- **frontend** - веб приложение пользователя.

Корневая директория содержит следующие файлы:

- **.gitignore** содержит список директорий игнорируемых системой контроля версий git. Если вам необходимо предотвратить их попадание в репозиторий, перечислите их в данном файле.
- **composer.json** - Конфигурация Composer, подробно описанная в разделе «Настройка Composer» ниже.
- **init** - скрипт инициализации. Подробно описан ниже в разделе «Конфигурации и окружения».
- **init.bat** - он же для Windows.
- **LICENSE.md** - информация о лицензии. Разместите в нём лицензию вашего проекта. Особенно в случае OpenSource.
- **README.md** - основная информация об установке шаблона. Можете разместить в нём информацию о вашем проекте и его установке.

- `requirements.php` - проверка соответствия требованиям Yii.
- `yii` - входной скрипт консольного приложения.
- `yii.bat` - он же для Windows.

15.1.4 Встроенные псевдонимы путей

- `@yii` - директория фреймворка.
- `@app` - корневая директория выполняемого в данный момент приложения.
- `@common` - директория `common`.
- `@frontend` - директория веб-приложения `frontend`.
- `@backend` - директория веб-приложения `backend`.
- `@console` - директория `console`.
- `@runtime` - директория `runtime` исполняемого приложения.
- `@vendor` - директория `vendor`, содержащая пакеты загруженные Composer'ом.
- `@bower` - директория `vendor`, содержащая пакеты `bower`³.
- `@npm` - директория `vendor`, содержащая пакеты `npm`⁴.
- `@web` - базовый URL исполняемого веб-приложения.
- `@webroot` - корневая веб-директория исполняемого веб-приложения.

Псевдонимы, характерные для структуры директорий приложения `advanced` (`@common`, `@frontend`, `@backend` и `@console`) задаются в `common/config/bootstrap.php`.

15.1.5 Приложения

В шаблоне `advanced` три приложения: `frontend`, `backend` и `console`. `Frontend` это та часть приложения, которая обеспечивает взаимодействие системы с конечным пользователем проекта. `Backend` это административная панель, аналитика и прочая подобная функциональность. `Console` обычно используется для выполнения заданий по расписанию через `cron`, низкоуровневого управления сервером, при развёртывании приложения, работы с миграциями и ресурсами.

Также есть директория `common`, которая содержит файлы используемые более чем одним приложением. Например, модель `User`. Оба веб приложения `frontend` и `backend` содержат директорию `web`. Это корневая директория, которую вы должны настроить в вебсервере.

У каждого приложения есть собственное пространство имён и соответствующий его названию псевдоним. Это же справедливо и для общей директории `common`.

³<http://bower.io/>

⁴<https://www.npmjs.org/>

15.1.6 Конфигурации и окружения

Существует множество проблем при типичном подходе к настройке конфигурации:

- Каждый член команды имеет свою собственную конфигурацию. Изменение конфигурации в общем репозитории повлияет на всех остальных.
- Пароль от эксплуатационной БД и API ключи не должны оказаться в репозитории.
- Существует много окружений: `development` (разработка), `testing` (тестирование), `production` (эксплуатация). Каждое окружение должно иметь свою собственную конфигурацию.
- Настройка всех параметров конфигурации для каждого случая однотипна и отнимает слишком много времени.

Для решения этих проблем Yii вводит простую концепцию окружений. Каждое окружение представлено набором файлов в директории `environments`. Для переключения между окружениями используется команда `init`. Она довольно проста. Всё, что она на самом деле делает - это копирование всех файлов из директории окружения в корневую директорию, где находятся все приложения.

Обычно окружение содержит входные скрипты приложения, такие как `index.php`, и файлы конфигурации, имена которых дополнены суффиксами `-local.php`. Эти файлы добавлены в `.gitignore` и никогда не попадут в репозиторий.

Чтобы избежать дублирования, конфигурации перекрывают друг друга. Например, приложение `frontend` считывает конфигурацию в следующем порядке:

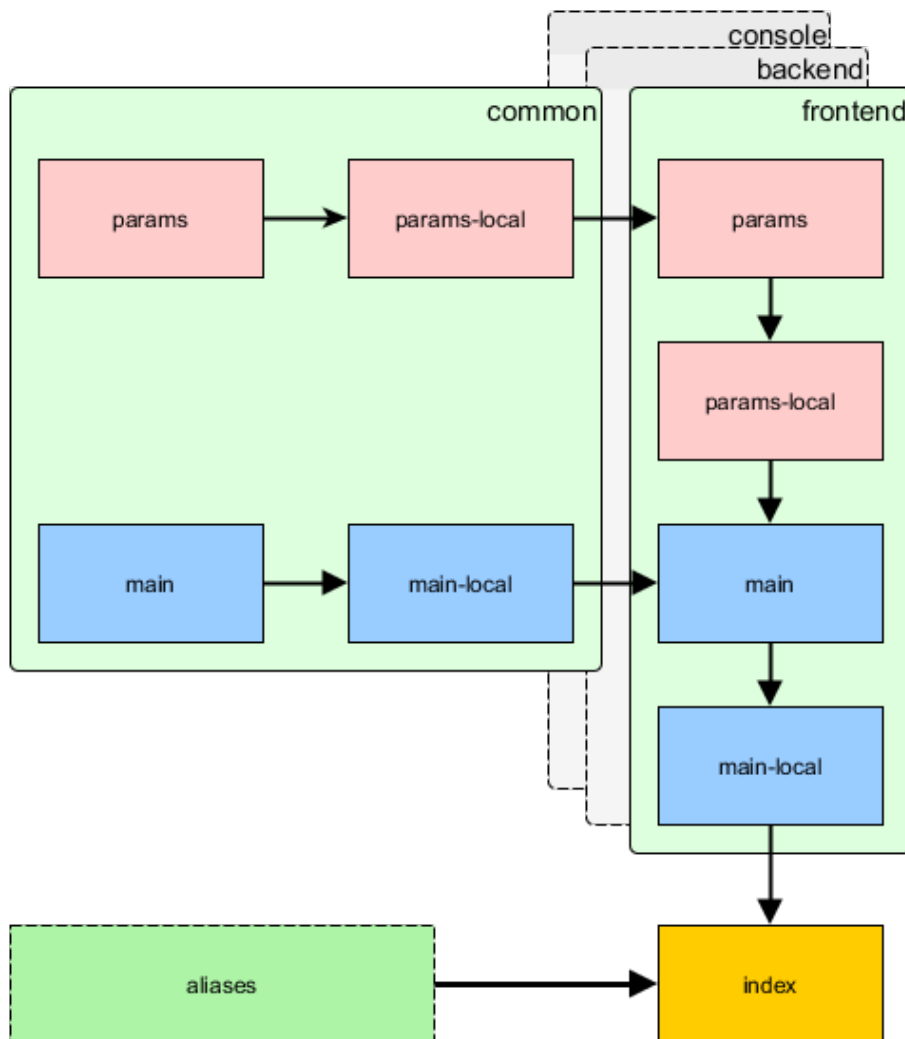
- `common/config/main.php`
- `common/config/main-local.php`
- `frontend/config/main.php`
- `frontend/config/main-local.php`

Параметры считываются в следующем порядке:

- `common/config/params.php`
- `common/config/params-local.php`
- `frontend/config/params.php`
- `frontend/config/params-local.php`

Значения из следующего конфигурационного файла перекрывают аналогичные значения из предыдущих конфигурационных файлов.

Полная схема:



15.1.7 Настройка Composer

После того как шаблон приложения установлен, хорошо бы изменить `composer.json` который находится в корневой директории проекта:

```
{
    "name": "yiisoft/yii2-app-advanced",
    "description": "Yii 2 Advanced Application Template",
    "keywords": ["yii2", "framework", "advanced", "application template"],
    "homepage": "http://www.yiiframework.com/",
    "type": "project",
    "license": "BSD-3-Clause",
    "support": {
        "issues": "https://github.com/yiisoft/yii2/issues?state=open",
        "forum": "http://www.yiiframework.com/forum/"
    }
}
```

```

        "wiki": "http://www.yiiframework.com/wiki/",
        "irc": "irc://irc.freenode.net/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "minimum-stability": "dev",
    "require": {
        "php": ">=5.4.0",
        "yiisoft/yii2": "*",
        "yiisoft/yii2-bootstrap": "*",
        "yiisoft/yii2-swiftmailer": "*"
    },
    "require-dev": {
        "yiisoft/yii2-codeception": "*",
        "yiisoft/yii2-debug": "*",
        "yiisoft/yii2-gii": "*",
        "yiisoft/yii2-faker": "*"
    },
    "config": {
        "process-timeout": 1800
    },
    "extra": {
        "asset-installer-paths": {
            "npm-asset-library": "vendor/npm",
            "bower-asset-library": "vendor/bower"
        }
    }
}

```

Во-первых, мы обновляем основную информацию. Меняем значения `name`, `description`, `keywords`, `homepage` и `support` на соответствующие вашему проекту.

А сейчас интересная часть. вы можете добавить больше пакетов, необходимых для вашего приложения, в раздел `require`. Все они с packagist.org⁵. Стоит его изучить так как там множество пакетов с полезным кодом.

После того как ваш `composer.json` настроен, вы можете выполнить в консоли команду `composer update --prefer-dist`, подождать пока требуемые пакеты загрузятся и установятся, и начать их использовать. Автозагрузка классов этих пакетов будет осуществляться автоматически.

15.1.8 Создание ссылок на frontend из backend

Часто приходится создавать ссылки из приложения backend на приложение frontend. Так как frontend может использовать собственную конфигурацию менеджера URL, вам придётся продублировать её в конфигурации backend под новым именем:

```

return [
    'components' => [
        'urlManager' => [

```

⁵<https://packagist.org/>


```
        // конфигурация основного менеджера URL в конфигурации backend
    ],
    'urlManagerFrontend' => [
        // конфигурация менеджера URL из frontend
    ],
],
];
```

После того, как это будет сделано, вы сможете получить URL, указывающий на frontend, следующим способом:

```
echo Yii::$app->urlManagerFrontend->createUrl(...);
```

Error: not existing file: tutorial-start-from-scratch.md

Error: not existing file: tutorial-console.md

15.2 Интернационализация

Примечание: Этот раздел находится в разработке

Интернационализация (I18N) является частью процесса разработки приложения, которое может быть адаптировано для нескольких языков без изменения программной логики. Это особенно важно для веб-приложений, так как потенциальные пользователи могут приходить из разных стран.

Yii располагает несколькими средствами, призванными помочь с интернационализацией веб-приложения: [переводом сообщений][1], [форматированием чисел и дат][2].

15.2.1 Локализация и языки

В Yii приложении определены два языка: **исходный язык** и `[[yii\base\Application::$language|язык перевода]]`.

На “исходном языке” написаны сообщения в коде приложения. Если мы определяем исходным языком английский, то в коде можно использовать конструкцию:

```
echo \Yii::t('app', 'I am a message!');
```

Язык перевода определяет, в каком виде будет отображаться текущая страница, т.е. на какой язык будут переведены оригинальные сообщения. Этот параметр определяется в конфигурации приложения:

```
return [  
    'id' => 'applicationID',  
    'basePath' => dirname(__DIR__),  
    // ...  
    'language' => 'ru-RU', // <- здесь!  
    // ...  
]
```

Подсказка: значение по умолчанию для **исходного языка** - английский.

Вы можете установить значение текущего языка в самом приложении в соответствии с языком, который выбрал пользователь. Это необходимо сделать до того, как будет сгенерирован какой-либо вывод, чтобы не возникло проблем с его корректностью. Используйте простое переопределение свойства на нужное значение:

```
\Yii::$app->language = 'ru-RU';
```

Формат для установки языка/локали: `ll-cc`, где `ll` - это двух или трёх-буквенный код языка в нижнем регистре в соответствии со стандартом ISO-639⁶, а `cc` - это код страны в соответствии со стандартом ISO-3166⁷.

⁶<http://www.loc.gov/standards/iso639-2/>

⁷<http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html>

Примечание: больше информации о синтаксисе и концепции локалей можно получить в документации проекта ICU⁸.

15.2.2 Перевод сообщений

Перевод используется для локализации сообщений, которые будут выведены в приложении в соответствии с языком, который выбрал пользователь.

По сути, Yii просто находит в файле с сообщениями на выбранном языке строку, соответствующую сообщению на исходном языке приложения. Для перевода сообщений, необходимо в самом приложении заключать их в метод `Yii::t()`. Первый аргумент метода - это категория, которая позволяет группировать сообщения по определённому признаку, а второй - само сообщение.

```
echo \Yii::t('app', 'This is a string to translate!');
```

Yii попытается загрузить файл перевода сообщений, соответствующий текущему **языку приложения** из одного из источников, определённых в `i18n` компонентах приложения. Сообщения - это набор файлов или база данных, которая содержит переведённые строки. Следующая конфигурация определяет, что сообщения должны браться из РНР-файлов:

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                //'sourceLanguage' => 'en-US',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

В приведённой конфигурации, `app*` - это шаблон, который определяет, какие категории обрабатываются источником. В нашем случае, мы обрабатываем все, что начинается с `app`. Файлы с сообщениями находятся в `@app/messages` (папке `messages` в вашем приложении). Массив `fileMap` определяет, какой файл будет подключаться для определённой категории. Если вы не хотите конфигурировать `fileMap`, можно положиться на соглашение, что название категории является именем файла. Например, категория `app/error` относится к файлу `app/error.php` в рамках `basePath`.

⁸<http://userguide.icu-project.org/locale#TOC-The-Locale-Concept>

Переводя сообщение `\Yii::t('app', 'This is a string to translate!')` при установленном языке приложения `ru-RU`, `Yii` сначала будет искать файл `@app/messages/ru-RU/app.php`, чтобы получить список доступных переводов. Если есть файл `ru-RU`, `Yii` также попытается поискать `ru` перед тем, как примет решение, что попытка перевода не удалась.

Кроме хранения в РНР-файлах (используя `PhpMessageSource`), `Yii` предоставляет ещё два класса:

- `yii\i18n\GettextMessageSource`, использующий GNU Gettext для МО или РО файлов.
- `yii\i18n\DbMessageSource`, использующий базу данных.

Именованные указатели

Вы можете добавлять параметры в строку для перевода, которые в выводе будут заменены соответствующими значениями, заключая параметр в фигурные скобки:

```
$username = 'Alexander';  
echo \Yii::t('app', 'Hello, {username}!', [  
    'username' => $username,  
]);
```

Обратите внимание, что в операции присваивания фигурные скобки не используются.

Позиционные указатели

```
$sum = 42;  
echo \Yii::t('app', 'Balance: {0}', $sum);
```

Подсказка: старайтесь сохранять читаемость сообщений и избегать избыточного использования позиционных параметров. Помните, что переводчик, скорее всего, будет располагать только файлом со строками и для него должно быть очевидно, на что будет заменён тот или иной указатель.

Указатели с расширенным форматированием

Чтобы использовать расширенные возможности, вам необходимо установить и включить РНР-расширение `intl`⁹. После этого вам станет доступен расширенный синтаксис указателей, а также сокращённая запись `{placeholderName, argumentType}`, эквивалентная форме `{placeholderName, argumentType, argumentStyle}`, позволяющая определять стиль форматирования.

Полная документация доступна на сайте ICU¹⁰, но далее в документации будут приведены примеры использования расширенных возможностей интернационализации.

⁹<http://www.php.net/manual/en/intro.intl.php>

¹⁰<http://icu-project.org/apiref/icu4c/classMessageFormat.html>

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number}', $sum);
```

Вы можете использовать один из встроенных форматов (`integer`, `currency`, `percent`):

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, currency}', $sum);
```

Или определить свой формат:

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, ,000,000000}', $sum);
```

Описание форматирования¹¹.

```
echo \Yii::t('app', 'Today is {0, date}', time());
```

Встроенные форматы - это `short`, `medium`, `long`, and `full`:

```
echo \Yii::t('app', 'Today is {0, date, short}', time());
```

Используя свой формат:

```
echo \Yii::t('app', 'Today is {0, date, yyyy-MM-dd}', time());
```

Описание форматирования¹².

```
echo \Yii::t('app', 'It is {0, time}', time());
```

Встроенные форматы - это `short`, `medium`, `long`, and `full`:

```
echo \Yii::t('app', 'It is {0, time, short}', time());
```

Используя свой формат:

```
echo \Yii::t('app', 'It is {0, date, HH:mm}', time());
```

Описание форматирования¹³.

```
echo \Yii::t('app', 'Число {n,number} прописью: {n, spellout}', ['n' => 42]);
```

```
echo \Yii::t('app', 'Вы - {n, ordinal} посетитель!', ['n' => 42]);
```

Выведет сообщение “Вы - 42-й посетитель!”.

¹¹http://icu-project.org/apiref/icu4c/classicu_1_1DecimalFormat.html

¹²http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html

¹³http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html

```
echo \Yii::t('app', 'Вы находитесь здесь уже {n, duration}', ['n' => 47]);
```

Выведет сообщение “Вы находитесь здесь уже 47 сек.”.

Множественное число В каждом языке используется свой способ склонения порядковых числительных. Некоторые правила весьма сложны, так что очень удобно, что использование функционала `i18n` не требует определения правил склонения. Требуется только указать формы склоняемого слова в различных ситуациях:

```
echo \Yii::t(
    'app',
    'На диване {n, plural, нет=0{ кошек} лежит=1{ одна кошка} опележит{ #
кошка} fewлежит{ # кошки} manyлежит{ # кошек} otherлежит{ # кошки}}!',
    ['n' => 0]
);
```

Выведет сообщение “На диване нет кошек!”.

В данном правиле

- `=0` означает ноль;
- `=1` соответствует ровно 1;
- `one` - 21, 31, 41 и так далее;
- `few` - от 2 до 4, от 22 до 24 и так далее;
- `many` - 0, от 5 до 20, от 25 до 30 и так далее;
- `other` - для всех прочих чисел (например, дробных).
- Решётка `#` заменяется на значение аргумента `n`.

Для некоторых языков правила могут быть более простыми. Например, для английского будет достаточно указать:

```
echo \Yii::t('app', 'There {n, plural, =0{are no cats} =1{is one cat} other{
are # cats}}!', ['n' => 0]);
```

Следует помнить, что если вы используете указатель дважды и в первый раз он используется, как `plural`, второй раз он должен быть использован, как `number`, иначе вы получите ошибку “Inconsistent types declared for an argument: U_ARGUMENT_TYPE_MISMATCH”:

```
В
корзине: {count, number} {count, plural, oneтовар{} fewтовара{} other
товаров{}}.
```

Подробная документация о формах склонений для различных языков доступна на сайте unicode.org¹⁴.

¹⁴http://unicode.org/repos/cldr-tmp/trunk/diff/supplemental/language_plural_rules.html

Вариации Вы можете указывать критерии форматирования сообщений в зависимости от ключевых слов. Приведённый пример демонстрирует возможность подстановки корректного рода в зависимости от параметра:

```
echo Yii::t('app', '{name} - {gender} и {gender, select, женщинаей{}  
мужчинаему{} otherему{}} нравится Yii!', [  
    'name' => 'Василий',  
    'gender' => 'мужчина',  
]);
```

Выведет сообщение “Василий - мужчина и ему нравится Yii!”.

Вы приведённом выражении, *мужчина* и *женщина* - это возможные варианты пола. На всякий случай, *other* обработает случай, если значение не совпадает с первыми двумя вариантами. Строки в скобках являются вторичными выражениями и могут быть просто строкой или строкой, содержащей дополнительные указатели.

Определение перевода по умолчанию

Вы можете определить переводы, которые будут использованы, как переводы по умолчанию для категорий, которые не попадают в другие переводы. Этот перевод должен быть помечен звёздочкой *** и указан в конфигурации приложения, как:

```
// конфигурация i18n компонента  
  
'i18n' => [  
    'translations' => [  
        '*' => [  
            'class' => 'yii\i18n\PhpMessageSource'  
        ],  
    ],  
],
```

Теперь можно использовать категории без необходимости конфигурировать каждую из них, что похоже на способ, которым была реализована поддержка интернационализации в Yii 1.1. Сообщения для категории будут загружаться из файла с переводом по умолчанию из *basePath*, т.е. *@app/messages*:

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

Сообщение будет загружено из файла *@app/messages/<LanguageCode>/not_specified_category.php*

Перевод сообщений модулей

Если вы хотите перевести сообщения в модуле и при этом не сгружать их все в один файл, можете прибегнуть к следующему приёму:

```

<?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);
    }
}

```

В приведённом примере мы использовали маску для поиска совпадений, и последующую фильтрацию по категориям для искомого файла. Вместо использования `fileMap`, вы можете прибегнуть к соглашению, что имя категории совпадает с именем файла и писать `Module::t('validation', 'your custom validation message')` или `Module::t('form', 'some form label')` напрямую.

Перевод сообщений виджетов

Для виджетов применимо такое же правило, как и для модулей:

```

<?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);
    }
}
```

Вместо использования `fileMap`, вы можете прибегнуть к соглашению, что имя категории совпадает с именем файла и писать `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` напрямую.

Примечание: для виджетов вы можете использовать `i18n` представления. На них распространяются те же правила, что и на контроллеры.

Перевод сообщений фреймворка

Yii поставляется с набором сообщений по умолчанию для ошибок валидации и некоторых других строк. Эти сообщения принадлежат катего-

рии yii. Если возникает необходимость изменить сообщения по умолчанию, переопределите i18n компонент приложения:

```
'i18n' => [
    'translations' => [
        'yii' => [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/messages'
        ],
    ],
],
```

После этого разместите изменённые строки в файле @app/messages/<language>/yii.php.

Обработка недостающих переводов

Если в источнике перевода отсутствует необходимое сообщение, Yii отобразит исходное содержимое сообщения. Данное поведение тем оправданнее, чем вы более стремитесь писать в исходном коде понятный текст сообщений. Тем не менее, иногда этого недостаточно, и может потребоваться произвольная обработка возникшей ситуации, когда источник не содержит искомой строки. Для этого следует использовать обработку события `missingTranslation` компонента `yii\i18n\MessageSource`.

Например, чтобы отметить все непереведённые строки, чтобы их было легче находить на странице, необходимо создать обработчик события. Изменим конфигурацию приложения:

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
            'on missingTranslation' => ['app\components\
TranslationEventHandler', 'handleMissingTranslation']
        ],
    ],
],
```

Создадим обработчик события:

```
<?php

namespace app\components;

use yii\i18n\MissingTranslationEvent;
```

```
class TranslationEventHandler
{
    public static function(MissingTranslationEvent $event) {
        $event->translatedMessage = "@MISSING: {$event->category}.{$event->
message} FOR LANGUAGE {$event->language} @";
    }
}
```

Если `yii\i18n\MissingTranslationEvent::$translatedMessage` установлен, как обработчик события, на странице будет выведен соответствующий результат перевода.

Внимание: каждый источник обрабатывает недостающие переводы самостоятельно. Если вы используете несколько разных источников сообщений и хотите обрабатывать недостающие переводы одинаково для всех, назначьте соответствующий обработчик события для каждого источника.

15.2.3 Представления

Вместо того, чтобы переводить сообщения так, как указано в предыдущем разделе, вы можете использовать `i18n` в ваших представлениях, чтобы обеспечить поддержку нескольких языков. Например, если существует представление `views/site/index.php` и для перевода его на русский язык необходимо отдельное представление, создайте папку `ru-RU` в папке с представлением текущего контроллера или виджета и создайте файл для русского языка: `views/site/ru-RU/index.php`. Yii загрузит файл для текущего языка, если он существует, или использует исходный `views/site/index.php`, если не сможет найти локализацию.

Примечание: если язык был определён, как `en-US` и соответствующих представлений не было найдено, Yii попытается найти представления в папке `en` перед тем, как использовать исходные.

15.2.4 Форматирование чисел и дат

См. описание форматирования дат.

15.2.5 Настройка PHP-окружения

Для работы с большей частью функций интернационализации, Yii использует PHP-расширение `intl`¹⁵. Например, это расширение используют классы, отвечающие за форматирование чисел и дат `yii\i18n\Formatter`

¹⁵<http://php.net/manual/en/book.intl.php>

и за форматирование строк `yii\i18n\MessageFormatter`. Оба класса поддерживают базовый функционал даже в том случае, если расширение `intl` не установлено. Однако, этот запасной вариант более-менее будет работать только для сайтов на английском языке, хотя, даже для них, большая часть широких возможностей расширения `intl` не будет доступна, поэтому его установка настоятельно рекомендуется.

PHP-расширение `intl`¹⁶ основано на библиотеке ICU¹⁷, которая описывает правила форматирования для различных локалей. Поэтому следует помнить, что форматирование чисел и дат вместе с синтаксисом форматирования может отличаться в зависимости от версии библиотеки ICU, которая была скомпилирована в вашем дистрибутиве PHP.

Чтобы сайт работал одинаково во всех окружениях, рекомендуется устанавливать одинаковую версию расширения `intl`, при этом удостоверяться, что везде используется одинаковая версия библиотеки ICU.

Чтобы узнать, какая версия ICU используется текущим PHP интерпретатором, используйте следующий скрипт:

```
<?php
echo "PHP: " . PHP_VERSION . "\n";
echo "ICU: " . INTL_ICU_VERSION . "\n";
```

Чтобы иметь доступ ко всем возможностям, описанным в документации, мы рекомендуем использовать ICU версии 49 или новее. В более ранних версиях отсутствует указатель # в правилах склонений. На сайте <http://site.icu-project.org/download> вы можете ознакомиться со списком доступных версий ICU. Обратите внимание, что схема нумерации версий изменилась после версии 4.8 и последовательность версий выглядит так: ICU 4.8, ICU 49, ICU 50.

¹⁶<http://php.net/manual/en/book.intl.php>

¹⁷<http://site.icu-project.org/>

Error: not existing file: tutorial-mailing.md

Error: not existing file: tutorial-performance-tuning.md

Error: not existing file: tutorial-shared-hosting.md

Error: not existing file: tutorial-template-engines.md

Глава 16

Виджеты

Error: not existing file: widget-bootstrap.md

Error: not existing file: widget-jui.md

Глава 17

Хелперы

Error: not existing file: helper-overview.md

17.1 ArrayHelper

В добавок к богатому набору функций¹ для работы с массивами, которые есть в самом PHP, Yii Array helper предоставляет свои статические функции, которые могут быть вам полезны.

17.1.1 Получение значений

Извлечение значений из массива, объекта или структуры состоящей из них обоих с помощью стандартных средств PHP является довольно скучным занятием. Сперва вам нужно проверить есть ли соответствующий ключ, с помощью `isset`, и, если есть получить, если нет – подставить значение по-умолчанию

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name :
    null;
```

Yii предлагает очень удобный метод для таких случаев:

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

Первый аргумент – массив или объект из которого мы извлекаем значение. Второй аргумент определяет как будут извлекаться данные и может выглядеть как один из таких вариантов: - Имя ключа массива или свойства объекта, значение которого нужно вернуть - Путь к нужному значению значению разделенный точками, как в примере выше. - Callback-функция возвращающая значение

Callback-функция должна выглядеть примерно так:

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

Третий необязательный аргумент определяет значение по-умолчанию, если не установлен – равен `null`. Используется так:

```
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

В случае если вы хотите получить значение и тут же удалить его из массива, вы можете использовать метод `remove`

¹<http://php.net/manual/en/book.array.php>

```
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

После выполнения этого кода переменная `$array` будет содержать `['options' => [1, 2]]`, а в переменной `$type` будет значение `A`. В отличие от метода `getValue`, метод `remove` поддерживает только простое имя ключа.

17.1.2 Проверка наличия ключей

`ArrayHelper::keyExists` работает так же как и стандартный `array_key_exists`², но также может проверять ключи без учёта регистра:

```
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) || !ArrayHelper::
    keyExists('username', $data2, false)) {
    echo "Please provide username.";
}
```

17.1.3 Извлечение колонок

Часто нужно извлечь колонку значений из многомерного массива или объекта. Например, список ID.

```
$data = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

Результатом будет `['123', '345']`.

Если нужны какие-то дополнительные трансформации или способ получения значения специфический, вторым аргументом может быть анонимная функция:

```
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

17.1.4 Переиндексация массивов

Чтобы проиндексировать массив в соответствии с определенным ключом, используется метод `index`. Входящий массив должен быть много-

²<http://php.net/manual/en/function.array-key-exists.php>

мерным или массивом объектов. Ключом может быть имя ключа вложенного массива, имя свойства объекта или анонимная функция, которая будет возвращать значение ключа по переданному массиву.

Если значение ключа равно `null`, то соответствующий элемент массива будет опущен и не попадет в результат.

```
$array = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$result = ArrayHelper::index($array, 'id');
// the result is:
// [
//     '123' => ['id' => '123', 'data' => 'abc'],
//     '345' => ['id' => '345', 'data' => 'def'],
// ]

// using anonymous function
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

17.1.5 Получение пар ключ-значение

Для получения пар ключ-значение из многомерного массива или из массива объектов вы можете использовать метод `map`.

Параметры `$from` и `$to` определяют имена ключей или свойств, которые будут использованы в `map`. Так же, третьим необязательным параметром вы можете задать правила группировки.

```
$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');
// the result is:
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// the result is:
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
```

```
//      '345' => 'ccc',
//      ],
// ]
```

17.1.6 Многомерная сортировка

Метод `multisort` помогает сортировать массивы объектов или вложенные массивы по одному или нескольким ключам. Например:

```
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

После сортировки мы получим:

```
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

Второй аргумент, определяющий ключи для сортировки может быть строкой, если это один ключ, массивом, если используются несколько ключей или анонимной функцией, как в примере ниже:

```
ArrayHelper::multisort($data, function($item) {
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

Третий аргумент определяет способ сортировки – от большего к меньшему или от меньшего к большему. В случае, если мы сортируем по одному ключу, передаем `SORT_ASC` или `SORT_DESC`. Если сортировка осуществляется по нескольким ключам, вы можете назначить направление сортировки для каждого из них с помощью массива.

Последний аргумент – это флаг, который используется в стандартной функции PHP `sort()`. Посмотреть его возможные значения можно тут³.

17.1.7 Определение типа массива

Удобный способ для определения, является массив индексным или ассоциативным. Вот пример:

```
// no keys specified
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// all keys are strings
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

³<http://php.net/manual/en/function.sort.php>

17.1.8 HTML-кодирование и HTML-декодирование значений

Для того, чтобы закодировать или раскодировать специальные символы в массиве строк в HTML-сущности, вы можете пользоваться методами ниже:

```
$encoded = ArrayHelper::htmlEncode($data);  
$decoded = ArrayHelper::htmlDecode($data);
```

По умолчанию кодируются только значения. Если установит второй параметр в `false`, вы можете так же кодировать и ключи массива. Кодирование использует кодировку приложения, которая может быть изменена с помощью третьего аргумента.

17.1.9 Слияние массивов

Слияние двух или больше массивов в один рекурсивно. Если каждый массив имеет одинаковый ключ, последний будет перезаписывать предыдущий (в отличие от функции `array_merge_recursive`). Рекурсивное слияние проводится когда все массивы имеют элемент одного и того же типа с одним и тем же ключом. Для элементов, ключом которого является значение типа `integer`, элементы из последнего будут добавлены к предыдущим массивам. Вы можете добавлять дополнительные массивы для слияние третьи, четвертым, пятым (и так далее) параметром.

```
ArrayHelper::merge($a, $b);
```

17.1.10 Получение массива из объекта

Часто нужно конвертировать объект в массив. Наиболее распространенный случай – конвертация модели Active Record в массив.

```
$posts = Post::find()->limit(10)->all();  
$data = ArrayHelper::toArray($post, [  
    'app\models\Post' => [  
        'id',  
        'title',  
        // the key name in array result => property name  
        'createTime' => 'created_at',  
        // the key name in array result => anonymous function  
        'length' => function ($post) {  
            return strlen($post->content);  
        },  
    ],  
]);
```

Первый аргумент содержит данные, которые вы хотите конвертировать. В нашем случае это Active Record модель `Post`.

Второй аргумент служит для управления процессом конвертации и может быть трех видов:

- просто имя поля
- пара ключ-значение, где ключ определяет ключ в результирующем массиве, а значение – название поля в модели, откуда берется значение.
- пара ключ-значение, где в качестве значения передается callback-функция, которая возвращает значение.

Результат конвертации будет таким:

```
[  
    'id' => 123,  
    'title' => 'test',  
    'createTime' => '2013-01-01 12:00AM',  
    'length' => 301,  
]
```

Вы можете определить способ конвертации из объекта в массив по умолчанию реализовав интерфейс `yii\base\Arrayable` в этом классе

Error: not existing file: helper-html.md

Error: not existing file: helper-url.md

Error: not existing file: helper-security.md