

Полное руководство по Yii 2.0

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

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

Русский translation provided by:

Alexander Makarov,
Timur Melnikov,
Dmitry Brusenskiy,
Vadim Belorussov,
Dmitry Naumenko,
Ivan Bagaev,
Vasiliy Baukin,
Alexander Roman,
Pavel Ivanov,
Alexandr Kalashnikov,
Mark Ragazzo,
Dmitry Korolev,
Evgeniy Tkachenko,
keltstr,
Sergey Anisimov,
Aleksandr Safonov,
Dmitriy Makarov,
Evgeniy Moiseenko,
Maxim Chistyakov,
Anton Abramov,
Peter Efremov,
Taras Gudz,
metheoryt,
Aleksandr Bushlanov,
Disassem,
Eugene Tupikov,
Viktor Pikaev,
Linux2000,
Alex Solomaha,
Alexey Rogachev,
Anton Anisimov,
Beowulfenator

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»	25
2.4	Работа с формами	28
2.5	Работа с базами данных	33
2.6	Генерация кода при помощи Gii	40
2.7	Взгляд в будущее	47
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	68
3.6	Модели	78
3.7	Виды	89
3.8	Модули	104
3.9	Фильтры	110
3.10	Виджеты	119
3.11	Ресурсы	123
3.12	Расширения	143
4	Обработка запросов	155
4.1	Обзор	155
4.2	Предзагрузка	156
4.3	Разбор и генерация URL	157
4.4	Запросы	173
4.5	Ответы	176

4.6	Сессии и куки	182
4.7	Обработка ошибок	189
4.8	Логгирование	193
5	Основные понятия	203
5.1	Компоненты	203
5.2	Свойства	205
5.3	События	207
5.4	Поведения	214
5.5	Конфигурации	221
5.6	Псевдонимы	227
5.7	Автозагрузка классов	229
5.8	Service Locator	231
5.9	Контейнер внедрения зависимостей	234
6	Работа с базами данных	243
6.1	Объекты доступа к данным (DAO)	243
6.2	Построитель запросов	256
6.3	Active Record	268
6.4	Миграции баз данных	303
7	Получение данных от пользователя	323
7.1	Создание форм	323
7.2	Проверка входящих данных	327
7.3	Загрузка файлов	342
7.4	Табличный ввод	346
7.5	Работа с несколькими моделями	348
8	Отображение данных	351
8.1	Форматирование данных	351
8.2	Постраничное разделение данных	356
8.3	Сортировка	357
8.4	Провайдеры данных	358
8.5	Виджеты для данных	365
8.6	Темизация	380
9	Безопасность	383
9.1	Аутентификация	385
9.2	Авторизация	389
9.3	Работа с паролями	405
9.4	Лучшие практики безопасности	408

10 Кеширование	413
10.1 Кэширование	413
10.2 Кэширование данных	413
10.3 Кэширование фрагментов	422
10.4 Кэширование страниц	425
10.5 HTTP кэширование	426
11 Веб-сервисы REST	431
11.1 Быстрый старт	431
11.2 Ресурсы	435
11.3 Контроллеры	439
11.4 Маршрутизация	443
11.5 Форматирование ответа	445
11.6 Аутентификация	448
11.7 Ограничение частоты запросов	451
11.8 Версионирование	453
11.9 Обработка ошибок	455
12 Инструменты разработчика	459
13 Тестирование	463
13.1 Тестирование	463
13.2 Настройка тестового окружения	465
13.3 Модульные тесты	466
13.4 Функциональные тесты	467
13.5 Приёмочное тестирование	467
13.6 Фикстуры	467
13.7 Управление фикстурами	473
14 Специальные темы	477
14.1 Создание своей структуры приложения	479
14.2 Консольное приложение	480
14.3 Встроенные валидаторы	486
14.4 Интернационализация	499
14.5 Отправка почты	510
14.6 Оптимизация производительности	515
14.7 Окружение виртуального хостинга	520
14.8 Использование шаблонизаторов	523
14.9 Работа со сторонним кодом	524
15 Виджеты	529

16 Хелперы	533
16.1 Хелперы	533
16.2 ArrayHelper	535
16.3 Html-помощник	541
16.4 Url хелпер	549

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 архитектурный паттерн 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».

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/team/>

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 Yii 2 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 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\web\Request` 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

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

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

В случае возникновения проблем читайте раздел “Troubleshooting” в документации Composer⁴. Если вы только начинаете использовать Composer, рекомендуем прочитать как минимум раздел “Basic usage”⁵.

В данном руководстве предполагается, что Composer установлен глобально⁶. То есть он доступен через команду `composer`. Если вы использу-

¹<https://getcomposer.org/>

²<https://getcomposer.org/download/>

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

⁴<https://getcomposer.org/doc/articles/troubleshooting.md>

⁵<https://getcomposer.org/doc/01-basic-usage.md>

⁶<https://getcomposer.org/doc/00-intro.md#globally>

ете `composer.phar` из локальной директории, изменяйте команды соответственно.

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

Примечание: Во время установки Yii Composer запрашивает довольно большое количество информации через Github API. Количество запросов варьируется в зависимости от количества зависимостей вашего проекта и может превысить ограничения **Github API**. Если это произошло, Composer спросит логин и пароль от Github. Это необходимо для получения токена для Github API. На быстрых соединениях это может произойти ещё до того, как Composer сможет обработать ошибку, поэтому мы рекомендуем настроить токен доступа до установки Yii. Инструкции приведены в документации Composer о токенах Github API⁷.

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

Установка Yii

```
composer global require "fxp/composer-asset-plugin:~1.2.0"
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

Первая команда устанавливает `composer asset plugin`⁸, который позволяет управлять зависимостями пакетов bower и npm через Composer. Эту команду достаточно выполнить один раз. Вторая команда устанавливает последнюю стабильную версию Yii в директорию `basic`. Если хотите, можете выбрать другое имя директории.

Информация: Если команда `composer create-project` не выполняется нормально, убедитесь, что вы корректно установили `composer asset plugin`. Мы можете сделать это выполнив `composer global show`. Вывод должен содержать `fxp/composer-asset-plugin`. Также можно обратиться к разделу “Troubleshooting” документации Composer⁹. Там описаны другие типичные ошибки. После того, как вы исправили ошибку, запустите `composer update` в директории `basic`.

Подсказка: Если вы хотите установить последнюю нестабильную ревизию Yii, можете использовать следующую команду, в которой присутствует опция `stability`¹⁰:

⁷<https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens>

⁸<https://github.com/francoispluchino/composer-asset-plugin/>

⁹<https://getcomposer.org/doc/articles/troubleshooting.md>

¹⁰<https://getcomposer.org/doc/04-schema.md#minimum-stability>

```
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.com¹¹;
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',
```

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`¹².

2.1.4 Проверка установки

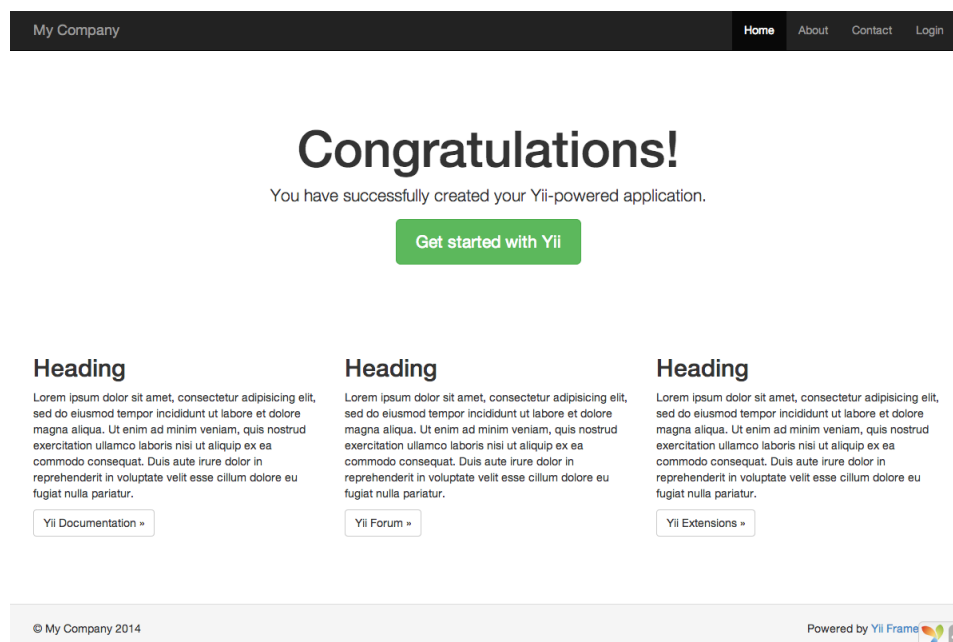
После установки приложение будет доступно по следующему URL:

```
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.

¹¹<http://www.yiiframework.com/download/>

¹²<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md>



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 PDO¹³ 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 развиваете его на рабочем сервере.

Приложение, установленное по инструкциям, приведённым выше, будет работать сразу как с Apache¹⁴, так и с Nginx¹⁵ под Windows и Linux с установленным PHP 5.4 и выше. Yii 2.0 также совместим с HHVM¹⁶. Тем

¹³<http://www.php.net/manual/ru/pdo.installation.php>

¹⁴<http://httpd.apache.org/>

¹⁵<http://nginx.org/>

¹⁶<http://hhvm.com/>

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`.

```
# 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
    # им 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 SAPI¹⁷ для Nginx¹⁸. 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.

¹⁷<http://php.net/manual/ru/install.fpm.php>

¹⁸<http://wiki.nginx.org/>

```
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$ {
        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](#).

Отметим, что в отличие от фреймворка как только приложение установлено, оно становится целиком вашим. Вы можете изменять его код как угодно.

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 пользователь 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 отладчик¹⁹, 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`):

¹⁹<https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md>

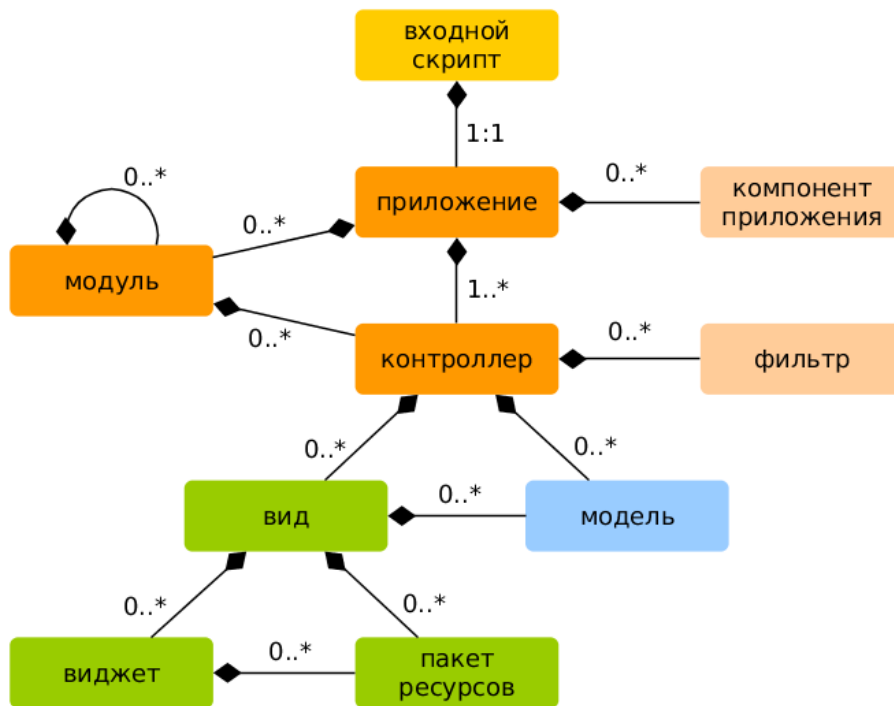
basic/	корневой каталог приложения
composer.json	используется Composerом', содержит описание приложения
config/	конфигурационные файлы
console.php	конфигурация консольного приложения
web.php	конфигурация Web приложения
commands/	содержит классы консольных команд
controllers/	контроллеры
models/	модели
runtime/	файлы, которые генерирует Yii во время выполнения приложения логи(, кэш и тп..)
vendor/	содержит пакеты Composer'a' и, собственно, сам фреймворк Yii
views/	виды приложения
web/	корневая директория Web приложения. Содержит файлы, доступные через Web
assets/	скрипты, используемые приложением (js, css)
index.php	точка входа в приложение Yii. С него начинается выполнение приложения
yii	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 реализован архитектурный паттерн MVC²⁰, которая соответствует структуре директорий приложения. В директории `models` находятся **Модели**, в `views` расположены **Виды**, а в каталоге `controllers` все **Контроллеры** приложения.

Диаграмма ниже демонстрирует внутреннее устройство приложения.

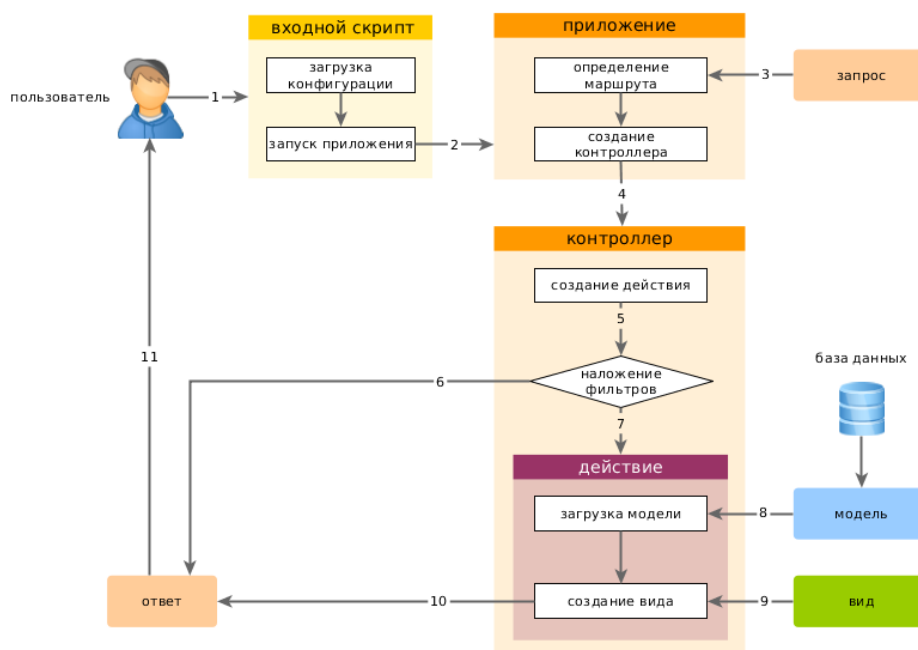
²⁰<http://ru.wikipedia.org/wiki/Model-View-Controller>



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 **ответ**.

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 PHP 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

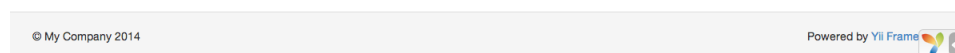
пользователя, который может попытаться провести XSS атаку²¹ путём вставки зловредного JavaScript кода.

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%2Fsay&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`.

²¹http://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 Заключение

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%2Fentry
```

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.

My Company

HomeAboutContactLogin

Name

Name cannot be blank.

Email

Email cannot be blank.

Submit

© My Company 2014

Powered by Yii Framework

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

My Company

HomeAboutContactLogin

You have entered the following information:

Name: Qiang Xue

Email: tester@example.com

© My Company 2014

Powered by Yii Framework

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 Заключение

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

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 PDO²² и драйвер PDO для используемой вами базы данных (н-р `pdo_mysql` для MySQL). Это базовое требование в случае использования вашим

²²<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 подразаемаемой 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**.

Если вам необходимо работать с базами данных, поддержка которых не включена непосредственно в фреймворк, стоит обратить внимание на следующие расширения:

- Informix²³
- IBM DB2²⁴
- Firebird²⁵

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;
```

²³<https://github.com/edgardmessias/yii2-informix>

²⁴<https://github.com/edgardmessias/yii2-ibm-db2>

²⁵<https://github.com/edgardmessias/yii2-firebird>

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

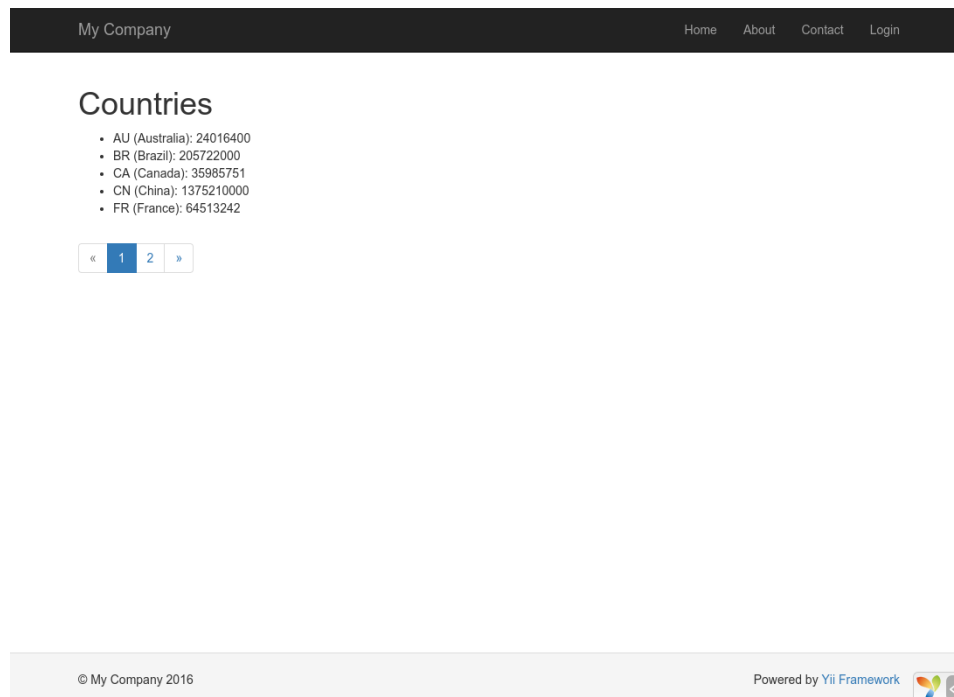
<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

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%2Findex
```



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%2Findex&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 Заключение

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 часто используемые 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'] = [
        'class' => '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`.

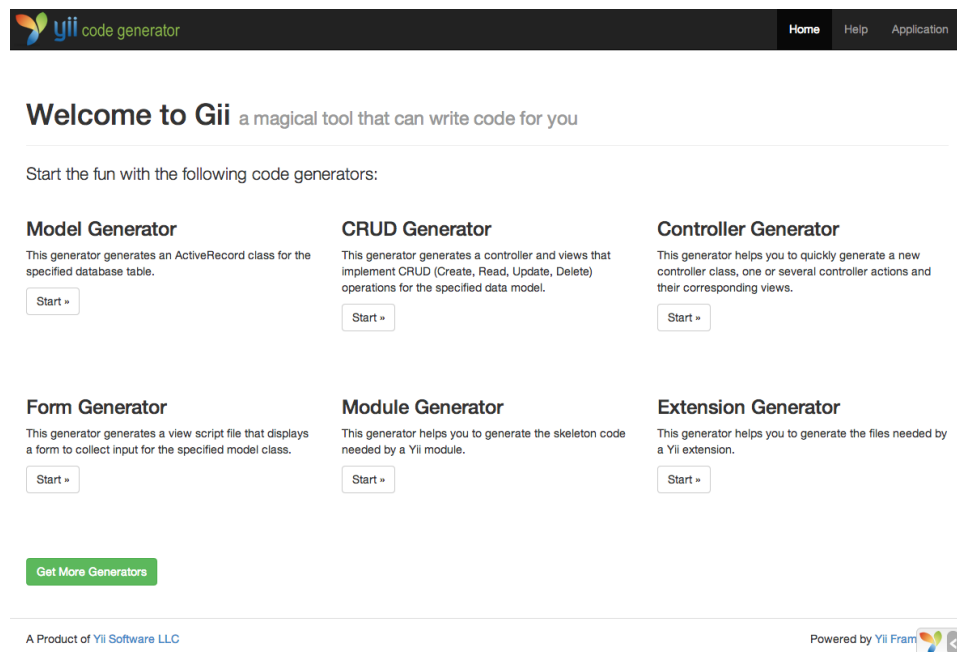
```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

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
```

Примечание: 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`

Yii code generator

Home Help Application

Model Generator

CRUD Generator

Controller Generator

Form Generator

Module Generator

Extension Generator

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

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

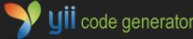
Для перезаписи существующего файла установите флажок рядом с “overwrite” и нажмите кнопку “Generate”. Для создания нового файла вы можете просто нажать “Generate”.

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\yii\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	<input checked="" type="checkbox"/>
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%2Findex>

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

Home About Contact Login

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

Home About Contact Login

Home / Countries / United States / Update

Update Country: United States

Code

Name

Population

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 Gii²⁶ 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 сбора 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

²⁶<https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md>

предоставить вам готовые решения для дальнейшего использования.

– Книги

- Расширения²⁷: 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
 - Форум²⁸
 - Чат Gitter²⁹
 - GitHub³⁰
 - Facebook³¹
 - Twitter³²
 - LinkedIn³³

²⁷<http://www.yiiframework.com/extensions/>

²⁸<http://www.yiiframework.com/forum/>

²⁹<https://gitter.im/yiisoft/yii2/rus>

³⁰<https://github.com/yiisoft/yii2>

³¹<https://www.facebook.com/groups/yiitalk/>

³²<https://twitter.com/yiiframework>

³³<https://www.linkedin.com/groups/yii-framework-1483367>

Глава 3

Структура приложения

3.1 Обзор

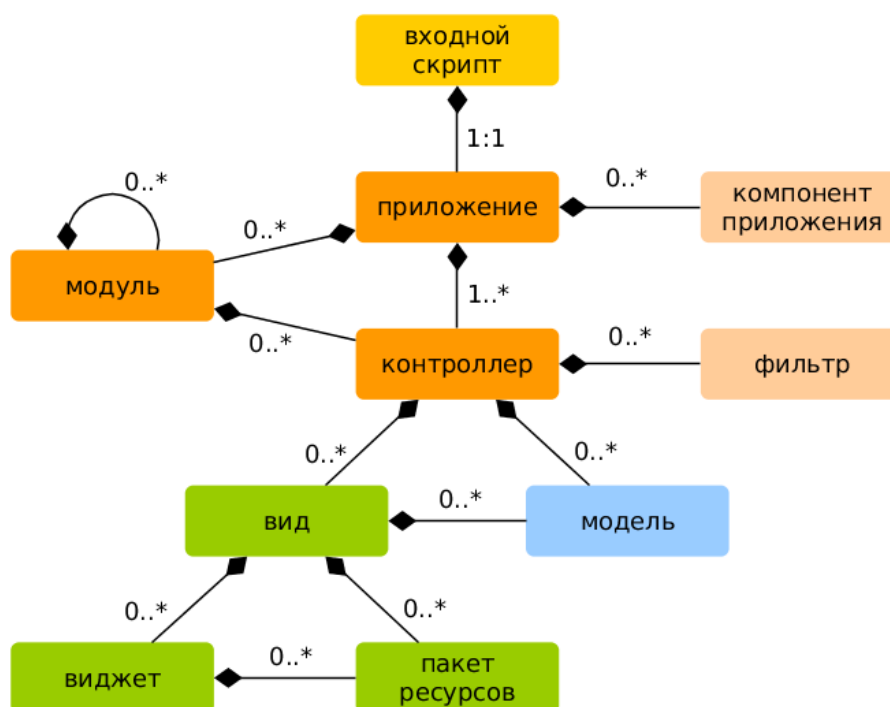
Yii приложения организованы согласно шаблону проектирования модель-представление-контроллер (MVC)¹. **Модели** представляют собой данные, бизнес логику и бизнес правила; **представления** отвечают за отображение информации, в том числе и на основе данных, полученных из моделей; **контроллеры** принимают 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 PHP 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<https://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);

// 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');

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

Информация: 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\jui\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 содержатся 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 PHP 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<https://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<https://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 качестве 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, компонент `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.

Компонент приложения будет создан при первом обращении к нему через вышеуказанное выражение. Любые дальнейшие обращения будут возвращать тот же экземпляр компонента.

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 Компоненты начальной загрузки

Как упоминалось выше, компонент приложения будет создан только при первом обращении к нему. Однако может возникнуть необходимость в наличии созданного компонента при каждом запросе, даже если напрямую к нему ни разу не обращались. Для этого необходимо указать ID компонента в качестве элемента свойства `bootstrap`.

К примеру, при данной конфигурации компонент `log` всегда подгружается при загрузке:

```
[  
    'bootstrap' => [  
        'log',  
    ],  
    'components' => [  
        'log' => [  
            // конфигурация для компонента 'log'  
        ],  
    ],  
]
```

3.4.2 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](#);
- **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](#);
- **yii\swiftmailer\Mailer**: 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`, 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.

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\NotFoundException;

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

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

    public function actionCreate()
    {
        $model = new Post;
```

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

```
        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/post-comment` должен находиться в файле `@app/controllers/admin/PostCommentController.php`.

Информация: Последний пример `admin/post-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 исключения. 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 *отдельное 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, не вызванные 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, то обычно это означает, что вам надо провести его рефакторинг и перенести часть кода в другие места.

В целом, контроллеры

- могут иметь доступ к данным **запроса**;
- 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 Модели

Модели являются частью архитектуры MVC⁹ (Модель-Вид-Контроллер). Они представляют собой объекты бизнес данных, правил и логики.

Вы можете создавать классы моделей путём расширения класса `yii\base\Model` или его дочерних классов. Базовый класс `yii\base\Model` поддерживает много полезных функций:

- Атрибуты: представляют собой рабочие данные и могут быть доступны как обычные свойства объекта или элементы массива;
- Метки атрибутов: задают отображение атрибута;
- Массовое присвоение: поддержка заполнения нескольких атрибутов в один шаг;
- Правила проверки: обеспечивают ввод данных на основе заявленных правил проверки;
- Экспорт Данных: разрешает данным модели быть экспортированными в массивы с настройкой форматов.

Класс `Model` также является базовым классом для многих расширенных моделей, таких как `Active Record`. Пожалуйста, обратитесь к соответствующей документации для более подробной информации об этих расширенных моделях.

Информация: Вы не обязаны основывать свои классы моделей на `yii\base\Model`. Однако, поскольку в yii есть много компонентов, созданных для поддержки `yii\base\Model`, обычно так делать предпочтительнее для базового класса модели.

⁹<http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

3.6.1 Атрибуты

Модели предоставляют рабочие данные в терминах *атрибутов*. Каждый атрибут представляет собой публично доступное свойство модели. Метод `yii\base\Model::attributes()` определяет какие атрибуты имеет класс модели.

Вы можете получить доступ к атрибуту как к обычному свойству объекта:

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

// "name" - это атрибут модели ContactForm
$model->name = 'example';
echo $model->name;
```

Также возможно получить доступ к атрибутам как к элементам массива, спасибо поддержке `ArrayAccess`¹⁰ и `ArrayIterator`¹¹ в `yii\base\Model`:

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

// доступ к атрибутам как к элементам массива
$model['name'] = 'example';
echo $model['name'];

// перебор атрибутов
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

Определение Атрибутов

По умолчанию, если ваш класс модели расширяется напрямую от `yii\base\Model`, то все *не статические публичные* переменные являются атрибутами. Например, у класса модели `ContactForm`, который находится ниже, четыре атрибута: `name`, `email`, `subject` и `body`. Модель `ContactForm` используется для представления входных данных, полученных из HTML формы.

```
namespace app\models;

use yii\base\Model;

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

¹⁰<http://php.net/manual/en/class.arrayaccess.php>

¹¹<http://php.net/manual/en/class.arrayiterator.php>

Вы можете переопределить метод `yii\base\Model::attributes()`, чтобы определять атрибуты другим способом. Метод должен возвращать имена атрибутов в модели. Например `yii\db\ActiveRecord` делает так, возвращая имена столбцов из связанной таблицы базы данных в качестве имён атрибутов. Также может понадобиться переопределить магические методы, такие как `__get()`, `__set()` для того, чтобы атрибуты могли быть доступны как обычные свойства объекта.

Метки атрибутов

При отображении значений или при получении ввода значений атрибутов, часто требуется отобразить некоторые надписи, связанные с атрибутами. Например, если атрибут назван `firstName`, Вы можете отобразить его как `First Name`, что является более удобным для пользователя, в тех случаях, когда атрибут отображается конечным пользователям в таких местах, как форма входа и сообщения об ошибках.

Вы можете получить метку атрибута, вызвав `yii\base\Model::getAttributeLabel()`. Например,

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

// отобразит "Name"
echo $model->getAttributeLabel('name');
```

По умолчанию, метки атрибутов автоматически генерируются из названия атрибута. Генерация выполняется методом `yii\base\Model::generateAttributeLabel()`. Он превращает первую букву каждого слова в верхний регистр, если имена переменных состоят из нескольких слов. Например, `username` станет `Username`, а `firstName` станет `First Name`.

Если Вы не хотите использовать автоматически сгенерированные метки, Вы можете переопределить метод `yii\base\Model::attributeLabels()`, чтобы явно объявить метку атрибута. Например,

```
namespace app\models;

use yii\base\Model;

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

    public function attributeLabels()
    {
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
        ];
    }
}
```

```
        'body' => 'Content',  
    ];  
}  
}
```

Для приложений поддерживающих мультиязычность, Вы можете перевести метки атрибутов. Это можно сделать в методе `attributeLabels()` как показано ниже:

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

Можно даже условно определять метки атрибутов. Например, на основе сценариев и использованной в нём модели, Вы можете возвращать различные метки для одного и того же атрибута.

Для справки: Строго говоря, метки атрибутов являются частью **видов**. Но объявление меток в моделях часто очень удобно и приводит к чистоте кода и повторному его использованию.

3.6.2 Сценарии

Модель может быть использована в различных *сценариях*. Например, модель `User` может быть использована для коллекции входных логинов пользователей, а также может быть использована для цели регистрации пользователей. В различных сценариях, модель может использовать различные бизнес-правила и логику. Например, атрибут `email` может потребоваться во время регистрации пользователя, но не во время входа пользователя в систему.

Модель использует свойство `yii\base\Model::$scenario`, чтобы отслеживать сценарий, в котором она используется. По умолчанию, модель поддерживает только один сценарий с именем `default`. В следующем коде показано два способа установки сценария модели:

```
// сценарий задается как свойство  
$model = new User;  
$model->scenario = User::SCENARIO_LOGIN;  
  
// сценарий задается через конфигурацию  
$model = new User(['scenario' => User::SCENARIO_LOGIN]);
```

По умолчанию сценарии, поддерживаемые моделью, определяются правилами валидации объявленными в модели. Однако, Вы можете изменить это поведение путем переопределения метода `yii\base\Model::scenarios()` как показано ниже:

```
namespace app\models;

use yii\db\ActiveRecord;

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

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

Информация: В приведенном выше и следующих примерах, классы моделей расширяются от `yii\db\ActiveRecord` потому, что использование нескольких сценариев обычно происходит от классов [Active Record](#).

Метод `scenarios()` возвращает массив, ключами которого являются имена сценариев, а значения - соответствующие *активные атрибуты*. Активные атрибуты могут быть массово присвоены и подлежат валидации. В приведенном выше примере, атрибуты `username` и `password` это активные атрибуты сценария `login`, а в сценарии `register` так же активным атрибутом является `email` вместе с `username` и `password`.

По умолчанию реализация `scenarios()` вернёт все найденные сценарии в правилах валидации задекларированных в методе `yii\base\Model::rules()`. При переопределении метода `scenarios()`, если Вы хотите ввести новые сценарии помимо стандартных, Вы можете написать код на основе следующего примера:

```
namespace app\models;

use yii\db\ActiveRecord;

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

    public function scenarios()
```



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

Возможности сценариев в основном используются валидацией и массовым присвоением атрибутов. Однако, Вы можете использовать их и для других целей. Например, Вы можете различным образом объявлять метки атрибутов на основе текущего сценария.

3.6.3 Правила валидации

Когда данные модели, получены от конечных пользователей, они должны быть проверены, для того чтобы убедиться, что данные удовлетворяют определенным правилам (так называемым *правилам валидации* также известными как *бизнес-правила*). Например, дана модель `ContactForm`, возможно Вы захотите убедиться, что все атрибуты являются не пустыми значениями, а атрибут `email` содержит допустимый адрес электронной почты. Если значения нескольких атрибутов не удовлетворяют соответствующим бизнес-правилам, то должны быть показаны соответствующие сообщения об ошибках, чтобы помочь конечному пользователю исправить допущенные ошибки.

Вы можете вызвать `yii\base\Model::validate()` для проверки полученных данных. Данный метод будет использовать правила валидации определённые в `yii\base\Model::rules()` для проверки каждого соответствующего атрибута. Если ошибок не найдено, то возвращается `true`, в противном случае возвращается `false`, а ошибки содержит свойство `yii\base\Model::$errors`. Например,

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

// модель заполнения атрибутов данными, вводимыми пользователем
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // все данные верны
} else {
    // проверка не удалась: $errors - это массив содержащий сообщения об ошибках
    $errors = $model->errors;
}
```

Объявляем правила валидации связанные с моделью, переопределяем метод `yii\base\Model::rules()` возврата правил атрибутов модели которые следует удовлетворить. В следующем примере показаны правила проверки объявленные в модели `ContactForm`:

```
public function rules()
{
    return [
        // name, email, subject и body атрибуты обязательны
        [['name', 'email', 'subject', 'body'], 'required'],

        // атрибут email должен быть правильным email адресом
        ['email', 'email'],
    ];
}
```

Правило может использоваться для проверки одного или нескольких атрибутов, также и атрибут может быть проверен одним или несколькими правилами. Пожалуйста, обратитесь к разделу [Проверка входных значений](#) для более подробной информации о том, как объявлять правила проверки.

Иногда необходимо, чтобы правила применялись только в определенных сценариях. Чтобы это сделать необходимо указать свойство `on` в правилах, следующим образом:

```
public function rules()
{
    return [
        // username, email и password требуются в сценарии "register"
        [['username', 'email', 'password'], 'required', 'on' => self::SCENARIO_REGISTER],

        // username и password требуются в сценарии "login"
        [['username', 'password'], 'required', 'on' => self::SCENARIO_LOGIN
    ],
];
}
```

Если не указать свойство `on`, то правило применяется во всех сценариях. Правило называется *активным правилом* если оно может быть применено в текущем сценарии `scenario`.

Атрибут будет проверяться тогда и только тогда если он является активным атрибутом объявленным в `scenarios()` и связанным с одним или несколькими активными правилами, объявленными в `rules()`.

3.6.4 Массовое Присвоение

Массовое присвоение - это удобный способ заполнения модели данными вводимыми пользователем с помощью одной строки кода. Он заполняет атрибуты модели путем присвоения входных данных непосредственно свойству `yii\base\Model::$attributes`. Следующие два куска кода эквивалентны, они оба пытаются присвоить данные из формы представленные конечными пользователями атрибутам модели `ContactForm`. Ясно, что первый код гораздо чище и менее подвержен ошибкам, чем второй:

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

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

Безопасные Атрибуты

Массовое присвоение применяется только к так называемым *безопасным атрибутам*, которые являются атрибутами, перечисленными в `yii\base\Model::scenarios()` в текущем сценарии `scenario` модели. Например, если модель `User` имеет следующий заданный сценарий, в данном случае это сценарий `login`, то только `username` и `password` могут быть массово присвоены. Любые другие атрибуты останутся нетронутыми.

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

Информация: Причиной того, что массовое присвоение атрибутов применяется только к безопасным атрибутам, является то, что необходимо контролировать какие атрибуты могут быть изменены конечными пользователями. Например, если модель `User` имеет атрибут `permission`, который определяет разрешения, назначенные пользователю, то необходимо быть уверенным, что данный атрибут может быть изменён только администраторами через бэкэнд-интерфейс.

По умолчанию `yii\base\Model::scenarios()` будет возвращать все сценарии и атрибуты найденные в `yii\base\Model::rules()`, если не переопределить этот метод, атрибут будет считаться безопасным только в случае, если он участвует в любом из активных правил проверки.

По этой причине существует специальный валидатор с псевдонимом `safe`, он предоставляет возможность объявить атрибут безопасным без фактической его проверки. Например, следующие правила определяют, что оба атрибута `title` и `description` являются безопасными атрибутами.

```
public function rules()
{
    return [
```

```
        [['title', 'description'], 'safe'],
    ];
}
```

Небезопасные атрибуты

Как сказано выше, метод `yii\base\Model::scenarios()` служит двум целям: определения, какие атрибуты должны быть проверены, и определения, какие атрибуты являются безопасными (т.е. не требуют проверки). В некоторых случаях необходимо проверить атрибут не объявляя его безопасным. Вы можете сделать это с помощью префикса восклицательный знак `!` в имени атрибута при объявлении его в `scenarios()` как атрибут `secret` в следующем примере:

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

Когда модель будет присутствовать в сценарии `login`, то все три эти атрибута будут проверены. Однако, только атрибуты `username` и `password` могут быть массово присвоены. Назначить входное значение атрибуту `secret` нужно явно следующим образом,

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

3.6.5 Экспорт Данных

Часто нужно экспортировать модели в различные форматы. Например, может потребоваться преобразовать коллекцию моделей в JSON или Excel формат. Процесс экспорта может быть разбит на два самостоятельных шага. На первом этапе модели преобразуются в массивы; на втором этапе массивы преобразуются в целевые форматы. Вы можете сосредоточиться только на первом шаге потому, что второй шаг может быть достигнут путем универсального инструмента форматирования данных, такого как `yii\web\JsonResponseFormatter`.

Самый простой способ преобразования модели в массив - использовать свойство `yii\base\Model::$attributes`. Например,

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

По умолчанию, свойство `yii\base\Model::$attributes` возвращает значения *всех* атрибутов объявленных в `yii\base\Model::attributes()`.

Более гибкий и мощный способ конвертирования модели в массив - использовать метод `yii\base\Model::toArray()`. Его поведение по умолчанию такое же как и у `yii\base\Model::$attributes`. Тем не менее, он

позволяет выбрать, какие элементы данных, называемые *полями*, поставить в результирующий массив и как они должны быть отформатированы. На самом деле, этот способ экспорта моделей по умолчанию применяется при разработке в RESTful Web service, как описано в [Response Formatting](#).

Поля

Поле - это просто именованный элемент в массиве, который может быть получен вызовом метода `yii\base\Model::toArray()` модели.

По умолчанию имена полей эквивалентны именам атрибутов. Однако, это поведение можно изменить, переопределив методы `fields()` и/или `extraFields()`. Оба метода должны возвращать список определенных полей. Поля определенные `fields()` являются полями по умолчанию, это означает, что `toArray()` будет возвращать эти поля по умолчанию. Метод `extraFields()` определяет дополнительно доступные поля, которые также могут быть возвращены `toArray()` так много, как Вы укажете их через параметр `$expand`. Например, следующий код будет возвращать все поля определенные в `fields()`, а также поля `prettyName` и `fullAddress`, если они определены в `extraFields()`.

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

Вы можете переопределить `fields()` чтобы добавить, удалить, переименовать или переопределить поля. Возвращаемым значением `fields()` должен быть массив. Ключами массива являются имена полей, а значениями - соответствующие определения полей, которые могут быть либо именами свойств/атрибутов, либо анонимными функциями, возвращающими соответствующие значения полей. В частном случае, когда имя поля совпадает с именем его атрибута, возможно опустить ключ массива. Например,

```
// использовать явное перечисление всех полей, лучше всего тогда, когда вы
// хотите убедиться,
// что изменения в вашей таблице базы данных или атрибуте модели не
// вызывают изменение вашего поля
// для( поддержания обратной совместимости API интерфейса).

public function fields()
{
    return [
        // здесь имя поля совпадает с именем атрибута
        'id',

        // здесь имя поля - "email", соответствующее ему имя атрибута - "
        email_address"
        'email' => 'email_address',

        // здесь имя поля - "name", а значение определяется обратным
        вызовом PHP
    ];
}
```

```
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// использовать фильтрацию нескольких полей лучше тогда, когда вы хотите
// наследовать
// родительскую реализацию и черный список некоторых чувствительных"" полей.

public function fields()
{
    $fields = parent::fields();

    // удаляем поля, содержащие конфиденциальную информацию
    unset($fields['auth_key'], $fields['password_hash'], $fields['
    password_reset_token']);

    return $fields;
}
```

Внимание: по умолчанию все атрибуты модели будут включены в экспортируемый массив, вы должны проверить ваши данные и убедиться, что они не содержат конфиденциальной информации. Если такая информация присутствует, вы должны переопределить `fields()` и отфильтровать поля. В приведенном выше примере мы выбираем и отфильтровываем `auth_key`, `password_hash` и `password_reset_token`.

3.6.6 Лучшие практические методики разработки моделей

Модели являются центральным местом представления бизнес-данных, правил и логики. Они часто повторно используются в разных местах. В хорошо спроектированном приложении, модели, как правило, намного больше, чем [контроллеры](#).

В целом, модели

- могут содержать атрибуты для представления бизнес-данных;
- могут содержать правила проверки для обеспечения целостности и достоверности данных;
- могут содержать методы с реализацией бизнес-логики;
- не следует напрямую задавать запрос на доступ, либо сессии, либо любые другие данные об окружающей среде. Эти данные должны быть введены [контроллерами](#) в модели;
- следует избегать встраивания HTML или другого отображаемого кода - это лучше делать в [видах](#);
- избегайте слишком большого количества сценариев в одной модели.

Рекомендации выше обычно учитываются при разработке больших сложных систем. В таких системах, модели могут быть очень большими, в связи с тем, что они используются во многих местах и поэтому могут содержать множество наборов правил и бизнес-логики. Это часто заканчивается кошмаром при поддержании кода модели, поскольку одним касанием кода можно повлиять на несколько разных мест. Чтобы сделать код модели более легким в обслуживании, Вы можете предпринять следующую стратегию:

- Определить набор базовых классов моделей, которые являются общими для разных **приложений** или **модулей**. Эти классы моделей должны содержать минимальный набор правил и логики, которые являются общими среди всех используемых приложений или модулей.
- В каждом **приложении** или **модуле** в котором используется модель, определить конкретный класс модели (или классы моделей), отходящий от соответствующего базового класса модели. Конкретный класс модели должен содержать правила и логику, которые являются специфическими для данного приложения или модуля.

Например, в шаблоне приложения `advanced`¹², Вы можете определить базовым классом модели `common\models\Post`. Тогда для frontend приложения, Вы определяете и используете конкретный класс модели `frontend\models\Post`, который расширяется от `common\models\Post`. И аналогичным образом для backend приложения, Вы определяете `backend\models\Post`. С помощью такой стратегии, можно быть уверенным, что код в `frontend\models\Post` используется только для конкретного frontend приложения, и если делаются любые изменения в нём, то не нужно беспокоиться, что изменения могут сломать backend приложение.

3.7 Виды

Виды - это часть MVC¹³ архитектуры, это код, который отвечает за представление данных конечным пользователям. В веб приложениях виды создаются обычно в виде *видов - шаблонов*, которые суть PHP скрипты, 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.

¹²<https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md>

¹³<https://ru.wikipedia.org/wiki/Model-View-Controller>

3.7.1 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 скриптинг¹⁴

¹⁴https://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

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>
```

Подсказка: 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 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()`.

3.7.2 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 NotFoundHttpException;
        }

        // 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'][] = 'О 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'] : [],
]) ?>
```

3.7.3 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 отлично 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; ?>

...
```

3.7.4 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 Twig¹⁵, Smarty¹⁶.

Также удобно пользоваться мелкими, но удобными фичами 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:

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

Этот код регистрирует 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

¹⁵<http://twig.sensiolabs.org/>

¹⁶<http://www.smarty.net/>

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 `registerMetaTag()`, вы можете указать ключ вторым параметром при вызове `registerLinkTag()` чтобы избежать дублирования link тэгов одного типа.

3.7.5 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');  
});
```

3.7.6 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%2Fpage&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.

3.7.7 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.

3.8 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.

3.8.1 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 имён 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 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.

Консольные команды в модулях

Ваш модуль также может объявлять команды, которые будут доступны через [консоль](#).

Для того, чтобы команда стала доступна, надо изменить свойство `yii\base\Module::$controllerNamespace` для консольного режима так, чтобы оно содержало пространство имён ваших команд.

Этого можно добиться проверяя класс экземпляра приложения `Yii` в методе `init` модуля:

```
public function init()
{
    parent::init();
    if (Yii::$app instanceof \yii\console\Application) {
        $this->controllerNamespace = 'app\modules\forum\commands';
    }
}
```

Ваши команды будут доступны из командной строки как:

```
yii <module_id>/<command>/<sub_command>
```

3.8.2 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 `yii\debug\Module` - 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',
    ],
]
```

3.8.3 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.

3.8.4 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 такую функциональность, как управление пользователями или управление комментариями, а затем использовать эти модули в будущих разработках.

3.9 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).

3.9.1 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()`.

3.9.2 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);
    }
}
```

3.9.3 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 Auth¹⁷, OAuth 2¹⁸. 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,

¹⁷http://en.wikipedia.org/wiki/Basic_access_authentication

¹⁸<http://oauth.net/2/>

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(),
            '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 дей-

ствии `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 bucket¹⁹. 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()
{
```

¹⁹http://en.wikipedia.org/wiki/Leaky_bucket

```

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 CORS²⁰

- 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;

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

```

Если вам необходимо добавить CORS-фильтрацию к `yii\rest\ActiveController` в вашем API, обратитесь к разделу Контроллеры.

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 `['*']`.

²⁰https://developer.mozilla.org/ru/docs/Web/HTTP/Access_control_CORS

- `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());
}

```

3.10 Виджеты

Виджеты представляют собой многократно используемые 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 UI²¹, виджеты Twitter Bootstrap²². Далее будут представлены базовые сведения о виджетах. Для получения сведений относительно использования конкретного виджета, следует обратиться к документации соответствующего класса.

3.10.1 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,

```

²¹<https://github.com/yiisoft/yii2-jui/blob/master/docs/guide/README.md>

²²<https://github.com/yiisoft/yii2-bootstrap/blob/master/docs/guide/usage-widgets.md>

```

        '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 тэги `<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 содержания.

Задание глобальных умолчаний

Глобальные умолчания для определённого типа виджета могут быть заданы через DI контейнер:

```

Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);

```

Подробнее это описано в подразделе «Практическое использование» раздела «Контейнер внедрения зависимостей».

3.10.2 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()`.

3.10.3 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 [представления](#).

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.

3.11 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.

3.11.1 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.

3.11.2 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 опубликовывать 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::$basePath` к относительному пути, и актуальный URL файла может быть определён путём добавления `yii\web\AssetManager::$baseUrl` 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 описан ниже
- 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

те 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”²³. Задан параметр публикации `beforeCopy`, здесь только `fonts` и `css` поддиректории будут опубликованы.

Bower и NPM Ресурсы

Большинство JavaScript/CSS пакетов управляются Bower²⁴ и/или NPM²⁵. 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.

²³<http://fontawesome.io/>

²⁴<http://bower.io/>

²⁵<https://www.npmjs.org/>

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`.

3.11.3 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:

```
use app\assets\AppAsset;
AppAsset::register($this); // $this - представляет собой объект
                             представления
```

Информация: 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.

Динамические Комплекты Ресурсов

Поскольку комплект ресурсов это обычный РНР класс, он может содержать дополнительную логику, связанную с ним, и может корректировать свои внутренние параметры динамически. Например, вы можете использовать сложную JavaScript библиотеку, которая предоставляет интернационализацию через отдельные исходные файлы: по одному на каждый поддерживаемый язык. Таким образом, вам нужно добавить определенный '.js' файл на вашу страницу, чтобы применить перевод для библиотеки. Этого можно достичь, переопределив метод `yii\web\AssetBundle::init()`:

```
namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class SophisticatedAssetBundle extends AssetBundle
{
    public $sourcePath = '/path/to/sophisticated/src';
    public $js = [
        'sophisticated.js' // file, which is always used
    ];

    public function init()
    {
        parent::init();
        $this->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file
        added
    }
}
```

Конкретный комплект ресурсов может быть также изменен через его экземпляр, возвращенный методом `yii\web\AssetBundle::register()`. Например:

```
use app\assets\SophisticatedAssetBundle;
use Yii;

$bundle = SophisticatedAssetBundle::register(Yii::$app->view);
$bundle->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file added
```

Замечание: несмотря на то что динамическая корректировка комплекта ресурсов поддерживается, ее использование - это **плохая** практика, которая может привести к неожиданным побочным эффектам, и которой следует избегать.

Настройка Комплектов Ресурсов

Yii управляет комплектами ресурсов через компонент приложения называемый `assetManager`, который реализован в `yii\web\AssetManager`. Пу-

тём 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`.

Имейте в виду, что настройки, установленный через `yii\web\AssetManager::$bundles`, применяются в момент создания комплекта ресурсов, т.е. в момент срабатывания конструктора. Таким образом, любые изменения, которые произведены над экземпляром комплекта ресурсов после этого, переключают настройки, установленные на уровне `yii\web\AssetManager::$bundles`. В частности, изменения, произведенные внутри метода `yii\web\AssetBundle::init()` или после регистрации комплекта ресурсов, имеют приоритет над настройками `AssetManager`. Ниже приведены примеры, в которых значения, установленные через `yii\web\AssetManager::$bundles` не возымеют никакого эффекта:

```
// Program source code:

namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class LanguageAssetBundle extends AssetBundle
{
    // ...

    public function init()
    {
        parent::init();
        $this->baseUrl = '@web/i18n/' . Yii::$app->language; // can NOT be
        handled by 'AssetManager'!
    }
}
// ...

$bundle = \app\assets\LargeFileAssetBundle::register(Yii::$app->view);
$bundle->baseUrl = Yii_DEBUG ? '@web/large-files': '@web/large-files/
    minified'; // can NOT be handled by 'AssetManager'!

// Application config :

return [
```

```
// ...
'components' => [
  'assetManager' => [
    'bundles' => [
      'app\assets\LanguageAssetBundle' => [
        'baseUrl' => 'http://some.cdn.com/files/i18n/en' //
        makes NO effect!
      ],
      'app\assets\LargeFileAssetBundle' => [
        'baseUrl' => 'http://some.cdn.com/files/large-files' //
        makes NO effect!
      ],
    ],
  ],
],
];
```

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.

3.11.4 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.

3.11.5 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 LESS²⁶ или SCSS²⁷; а для

²⁶<http://lesscss.org/>

²⁷<http://sass-lang.com/>

JavaScript можно использовать TypeScript²⁸.

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 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:

- LESS²⁹: `.less`
- SCSS³⁰: `.scss`
- Stylus³¹: `.styl`
- CoffeeScript³²: `.coffee`
- TypeScript³³: `.ts`

Yii ориентируется на установленные инструменты конвертации ресурсов препроцессора. Например, используя LESS³⁴, Вы должны установить команду `lessc` препроцессора.

Вы можете настроить команды препроцессора и поддерживать расширенный синтаксис сконфигурировав `yii\web\AssetManager::$converter` следующим образом:

```
return [
```

²⁸<http://www.typescriptlang.org/>

²⁹<http://lesscss.org/>

³⁰<http://sass-lang.com/>

³¹<http://learnboost.github.io/stylus/>

³²<http://coffeescript.org/>

³³<http://www.typescriptlang.org/>

³⁴<http://lesscss.org/>

```
'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 выше. Например, Вы можете использовать инструменты построения, такие как `grunt`³⁵ для отслеживания и автоматической конвертации ресурсов расширенного синтаксиса. В этом случае, Вы должны перечислить конечные CSS/JavaScript файлы в комплекте ресурсов вместо исходных файлов.

3.11.6 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.

³⁵<http://gruntjs.com/>

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 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 обеих 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 Compiler³⁶, YUI Compressor³⁷) 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 А зависит от В, который зависит от С и D, то Вы должны перечислить файлы ресурсов начиная с С и D, затем В, и только после этого А.

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:

```
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 А, В, С 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 А, В и С, только два этих объединённых файла будут включены, то же самое произойдёт и со страницей Y.

³⁶<https://developers.google.com/closure/compiler/>

³⁷<https://github.com/yui/yuicompressor/>

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).

Замечание: этот механизм объединения комплектов ресурсов основан на способности `yii\web\AssetManager::$bundles` перекрывать поля регистрируемых комплектов ресурсов. Однако, как уже было сказано выше, эта возможность не распространяется на изменения, внесенные в комплекты ресурсов на уровне метода `yii\web\AssetBundle::init()` или после регистрации. Вам следует избегать использования динамических комплектов ресурсов в процессе объединения.

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 команду обратный вызов для сжатия файлов JavaScript:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file {to}',
    // Настроить команду обратный вызов для сжатия файлов CSS:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o {to}',
    // Whether to delete asset source after compression:
    'deleteSource' => false,
    // 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 Compiler³⁸ для объединения JavaScript файлов и YUI Compressor³⁹ для объединения CSS файлов. Вы должны установить эти инструменты вручную или настроить данные параметры, чтобы использовать ваши любимые инструменты.

³⁸<https://developers.google.com/closure/compiler/>

³⁹<https://github.com/yui/yuicompressor/>

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.

Примечание: в случае если вы перенастраиваете комплекты ресурсов через `yii\web\AssetManager::$bundles` или `yii\web\AssetManager::$assetMap`, и хотите, чтобы эти настройки применились для исходных файлов для сжатия, вы должны занести эти опции в раздел `assetManager` файла конфигурации для команды `asset`.

Замечание: составляя набор исходных комплектов ресурсов для сжатия, следует избегать использования таких, чьи параметры могут изменяться динамически (т.е. на уровне метода `init()` или после регистрации), поскольку они могут функционировать неправильно после сжатия.

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`⁴⁰ 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:

⁴⁰<http://gruntjs.com/>

```
return [
    ...
    // Укажите выходной комплект для групп:
    '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.

3.12 Расширения

Расширения - это распространяемые программные пакеты, специально разработанные для использования в приложениях Yii и содержащие

готовые функции. Например, расширение `yii2-debug`⁴¹ 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”.

3.12.1 Использование расширений

Чтобы использовать расширение, вам необходимо установить его. Большинство расширений распространяются как пакеты Composer⁴², 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 Composer⁴³, если у вас его нет.

По умолчанию, Composer устанавливает пакеты, зарегистрированные на Packagist⁴⁴ - крупнейшем репозитории для пакетов Composer с открытым исходным кодом. Вы также можете создать свой репозиторий⁴⁵ 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 `yii2-imagine` нужно отредактировать ваш `composer.json` как показано далее:

```
{  
    // ...
```

⁴¹<https://github.com/yiisoft/yii2-debug>

⁴²<https://getcomposer.org/>

⁴³<https://getcomposer.org/>

⁴⁴<https://packagist.org/>

⁴⁵<https://getcomposer.org/doc/05-repositories.md#repository>

```
"require": {  
    // ... другие зависимости  
  
    "yiisoft/yii2-image": "~2.0.0"  
}  
}
```

После установки вы можете увидеть директорию `yiisoft/yii2-image`, находящуюся по пути `BasePath/vendor`. Также вы можете увидеть директорию `image/image`, которая содержит зависимый пакет.

Информация: `yiisoft/yii2-image` является базовым расширением, которое разрабатывает и поддерживает команда разработчиков Yii. Все базовые расширения размещены на Packagist⁴⁶ и называются `yiisoft/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 `yiisoft/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.

⁴⁶<https://packagist.org/>

Если расширение не имеет автозагрузчика классов, но следует стандарту PSR-4⁴⁷, 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',
    ],
]
```

3.12.2 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 Composer⁴⁸, 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`⁴⁹. 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`⁵⁰, 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 Composer⁵¹.

⁴⁷<http://www.php-fig.org/psr/psr-4/>

⁴⁸<https://getcomposer.org/>

⁴⁹<https://github.com>

⁵⁰<https://packagist.org/>

⁵¹<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": "~2.0.0",
        "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 Yii 2.

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-4⁵² или стандарту PSR-0⁵³ 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-imagine` мы используем `yii\imagine` 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
        });
    }
}
```

⁵²<http://www.php-fig.org/psr/psr-4/>

⁵³<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 содержимое конечному пользователю, вы должны попробовать [интернационализировать и локализовать](#) его. В частности,

- Если расширение отображает сообщения, предназначенные для конечных пользователей, сообщения должны быть обернуты 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 версионирования⁵⁴ 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 Packagist⁵⁵. 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.

⁵⁴<http://semver.org>

⁵⁵<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 Markdown⁵⁶ 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`⁵⁷, чтобы узнать, как нужно документировать код.

Информация: Ваши комментарии к коду могут быть написаны в формате Markdown. Расширение `yiisoft/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 фреймворка⁵⁸.

3.12.3 Базовые расширения

Yii предоставляет следующие базовые расширения, которые разрабатывает и поддерживает команда разработчиков Yii. Они все зарегистрированы на Packagist⁵⁹ и могут быть легко установлены, как описано в подразделе Использование расширений.

- `yiisoft/yii2-apidoc`⁶⁰: предоставляет расширяемый и высокопроизводительный генератор документации API. Оно также используется для генерации документации API фреймворка.

⁵⁶<http://daringfireball.net/projects/markdown/>

⁵⁷<https://github.com/yiisoft/yii2/blob/master/framework/base/Object.php>

⁵⁸<https://github.com/yiisoft/yii2/wiki/Core-framework-code-style>

⁵⁹<https://packagist.org/>

⁶⁰<https://github.com/yiisoft/yii2-apidoc>

- `yiiisoft/yii2-authclient`⁶¹: предоставляет набор наиболее часто используемых клиентов авторизации, таких, как Facebook OAuth2 клиент и GitHub OAuth2 клиент.
- `yiiisoft/yii2-bootstrap`⁶²: предоставляет набор виджетов, которые являются компонентами и плагинами Bootstrap⁶³.
- `yiiisoft/yii2-codeception`⁶⁴: предоставляет поддержку тестирования, основанного на Codeception⁶⁵.
- `yiiisoft/yii2-debug`⁶⁶: 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.
- `yiiisoft/yii2-elasticsearch`⁶⁷: предоставляет поддержку использования Elasticsearch⁶⁸. Оно включает в себя поддержку основных поисковых запросов, а также реализует шаблон проектирования *Active Record*, который позволяет хранить записи *Active Record* в Elasticsearch.
- `yiiisoft/yii2-faker`⁶⁹: предоставляет поддержку использования *Faker*⁷⁰ для генерации фиктивных данных.
- `yiiisoft/yii2-gii`⁷¹: предоставляет веб-интерфейс для генерации кода, который является весьма расширяемым и может быть использован для быстрой генерации моделей, форм, модулей, CRUD и т.д.
- `yiiisoft/yii2-httpclient`⁷²: предоставляет HTTP клиент.
- `yiiisoft/yii2-imagine`⁷³: предоставляет часто используемые функции для работы с изображениями, основанные на библиотеке *Imagine*⁷⁴.
- `yiiisoft/yii2-jui`⁷⁵: предоставляет набор виджетов, основанный на взаимодействиях и виджетах *jQuery UI*⁷⁶.
- `yiiisoft/yii2-mongodb`⁷⁷: предоставляет поддержку использования *MongoDB*⁷⁸. Оно включает такие возможности, как базовые запросы, *Active*

⁶¹<https://github.com/yiiisoft/yii2-authclient>

⁶²<https://github.com/yiiisoft/yii2-bootstrap>

⁶³<http://getbootstrap.com/>

⁶⁴<https://github.com/yiiisoft/yii2-codeception>

⁶⁵<http://codeception.com/>

⁶⁶<https://github.com/yiiisoft/yii2-debug>

⁶⁷<https://github.com/yiiisoft/yii2-elasticsearch>

⁶⁸<http://www.elasticsearch.org/>

⁶⁹<https://github.com/yiiisoft/yii2-faker>

⁷⁰<https://github.com/fzaninotto/Faker>

⁷¹<https://github.com/yiiisoft/yii2-gii>

⁷²<https://github.com/yiiisoft/yii2-httpclient>

⁷³<https://github.com/yiiisoft/yii2-imagine>

⁷⁴<http://imagine.readthedocs.org/>

⁷⁵<https://github.com/yiiisoft/yii2-jui>

⁷⁶<http://jqueryui.com/>

⁷⁷<https://github.com/yiiisoft/yii2-mongodb>

⁷⁸<http://www.mongodb.org/>

Record, миграции, кэширование, генерация кода и т.д.

- `yiisoft/yii2-redis`⁷⁹: предоставляет поддержку использования `redis`⁸⁰. Оно включает такие возможности, как базовые запросы, Active Record, кэширование и т.д.
- `yiisoft/yii2-smarty`⁸¹: предоставляет шаблонизатор, основанный на `Smarty`⁸².
- `yiisoft/yii2-sphinx`⁸³: предоставляет поддержку использования `Sphinx`⁸⁴. Оно включает такие возможности, как базовые запросы, Active Record, генерация кода и т.д.
- `yiisoft/yii2-swiftmailer`⁸⁵: предоставляет возможности отправки email, основанные на `swiftmailer`⁸⁶.
- `yiisoft/yii2-twig`⁸⁷: предоставляет шаблонизатор, основанный на `Twig`⁸⁸.

⁷⁹<https://github.com/yiisoft/yii2-redis>

⁸⁰<http://redis.io/>

⁸¹<https://github.com/yiisoft/yii2-smarty>

⁸²<http://www.smarty.net/>

⁸³<https://github.com/yiisoft/yii2-sphinx>

⁸⁴<http://sphinxsearch.com>

⁸⁵<https://github.com/yiisoft/yii2-swiftmailer>

⁸⁶<http://swiftmailer.org/>

⁸⁷<https://github.com/yiisoft/yii2-twig>

⁸⁸<http://twig.sensiolabs.org/>

Глава 4

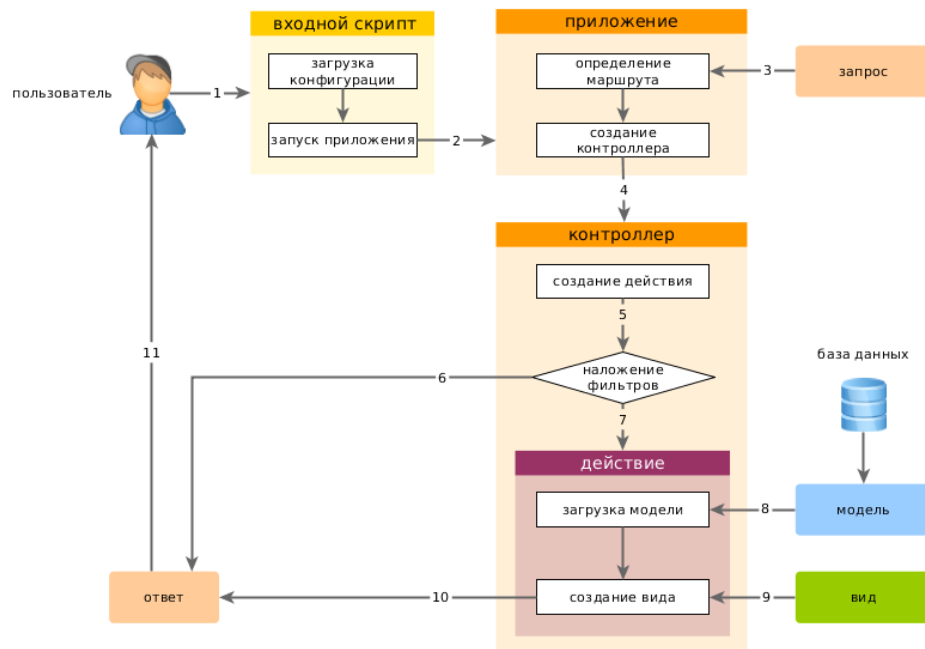
Обработка запросов

4.1 Обзор

Все запросы, обрабатываемые Yii приложением, проходят подобный путь.

1. Пользователь создает запрос ко **входному скрипту** `web/index.php`.
2. Входной скрипт загружает **конфигурацию** и создает экземпляр **приложения** для обработки запроса.
3. Приложение определяет запрошенный **маршрут** при помощи компонента `request`.
4. Приложение создает экземпляр **контроллера** для обработки запроса.
5. Контроллер создает экземпляр **действия** и выполняет фильтры для этого действия.
6. При неудачном выполнении любого **фильтра**, действие не выполняется.
7. При успешном выполнении всех фильтров, выполняется действие.
8. Действие загружает **модель** данных, возможно, из базы данных.
9. Действие рендерит **представление** и передает ему модель данных.
10. Результат рендеринга передается в компонент приложения `response`.
11. Компонент `response` посылает готовые данные пользователю.

Ниже представлена диаграмма обработки запроса приложением.



В данном разделе описаны подробности некоторых этапов обработки запроса.

4.2 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.

4.3 Разбор и генерация URL

При обработке запрошенного URL, Yii приложение первым делом разбирает URL в маршрут. Полученный маршрут используется при создании соответствующего экземпляра действия контроллера для обработки запроса. Этот процесс называется *роутинг*.

Обратный роутингу процесс называется *Создание URL*, он отвечает за создание URL из заданного маршрута и соответствующих параметров запроса. При необходимости, созданный URL всегда может быть преобразован в первоначальные маршрут и параметры запроса.

В основе роутинга и создания URL лежит использование `URL manager`, зарегистрированного в качестве компонента приложения `urlManager`. `URL manager` содержит метод `parseRequest()` для разбора входящего запроса на маршрут и параметры запроса, и метод `createUrl()` для создания URL из заданного маршрута и параметров запроса.

Настройка компонента `urlManager` в конфигурации приложения, позволяет приложению распознавать различные форматы URL без внесе-

¹<http://php.net/manual/ru/intro.opcache.php>

²<http://php.net/manual/ru/book.apc.php>

ния изменений в существующий код приложения. Например, для создания URL для действия `post/view`, можно использовать следующий код:

```
use yii\helpers\Url;

// Url::to() вызывает UrlManager::createUrl() для создания URL
$url = Url::to(['post/view', 'id' => 100]);
```

В зависимости от настройки `urlManager`, URL может быть создан в одном из следующих форматов (или любом другом формате). При последующем запросе URL в таком формате, он будет разобран на исходные маршрут и параметры запроса.

```
/index.php?r=post/view&id=100
/index.php/post/100
/post/100
```

4.3.1 Форматы URL

URL `manager` поддерживает два формата URL:

- Обычный.
- Человекопонятные URL.

Обычный формат URL использует параметр `r` для передачи маршрута и любые другие параметры для передачи остальных параметров запроса. Например, URL `/index.php?r=post/view&id=100` задает маршрут `post/view` и параметр `id`, равный 100. Данный формат не требует специальной конфигурации URL `manager` и работает с любыми настройками Веб сервера.

Человекопонятный формат URL представляет собой дополнительный путь, следующий за именем входного скрипта, описывающий маршрут и остальные параметров запроса. Например, дополнительный путь в URL `/index.php/post/100` - это `/post/100`, который может представлять маршрут `post/view` и параметр `id` со значением равным 100, при наличии соответствующего правила. Для использования ЧПУ, необходимо создать набор правил, соответствующих требованиям к URL.

Переключение между двумя форматами URL осуществляется при помощи свойства `enablePrettyUrl` компонента URL `manager` без внесения изменений в код приложения.

4.3.2 Роутинг

Роутинг осуществляется в два этапа:

- Входящий запрос разбирается в маршрут и параметры запроса.
- Для обработки запроса создается действие контроллера, соответствующее полученному маршруту.

При использовании простого формата URL, получение маршрута из запроса заключается в получении параметра `r` из массива `GET`.

При использовании ЧПУ, компонент `URL manager` ищет среди зарегистрированных правил подходящее для разрешения запроса в маршруте. Если такое правило не найдено, вызывается исключение `yii\web\NotFoundHttpException`.

После того, как из запроса получен маршрут, самое время создать действие контроллера, соответствующее этому маршруту. Маршрут разделяется на несколько частей, метками деления служат прямые слешей. Например, маршрут `site/index` будет разделен на `site` и `index`. Каждая из частей представляет собой *идентификатор*, который может ссылаться на модуль, контроллер или действие. Начиная с первой части маршрута, приложение следует следующему алгоритму для создания модуля (если есть), контроллера и действия:

1. Текущим модулем считаем приложение.
2. Проверяем, содержит ли **карта контроллеров** текущего модуля текущий *идентификатор*. Если содержит, в соответствии с конфигурацией контроллера, найденной в карте, создаем объект контроллера и переходим в п. 5 для обработки оставшейся части маршрута.
3. Проверяем, есть ли модуль, соответствующий *идентификатору* в списке модулей (свойство `modules`) текущего модуля. Если есть, в соответствии с конфигурацией модуля, найденной в списке модулей, создаем модуль и переходим в п. 2, считая только что созданный модуль текущим.
4. Рассматриваем *идентификатор* как идентификатор контроллера и создаем объект контроллера. Для оставшейся части маршрута выполняем п. 5.
5. Контроллер ищет текущий *идентификатор* в его **карте действий**. В случае нахождения, контроллер создает действие, в соответствии с конфигурацией, найденной в карте. Иначе, контроллер пытается создать встроенное действие, описанное методом, соответствующим текущему идентификатору действия.

При возникновении ошибок на любом из описанных выше этапов, вызывается исключение `yii\web\NotFoundHttpException`, указывающее на ошибку в процессе роутинга.

Маршрут по умолчанию

В случае, если в результате разбора запроса получен пустой маршрут, вместо него будет использован, так называемый, маршрут по умолчанию. Изначально, маршрут по умолчанию имеет значение `site/index`, и указывает на действие `index` контроллера `site`. Указать свое значение можно при помощи свойства приложения `defaultRoute`, например так:

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

В добавок к маршруту по умолчанию приложения, существует маршрут по умолчанию модулей. Например, если у нас есть модуль `user` и запрос разбирается в маршрут `user`, `defaultRoute` модуля используется для определения контроллера. По умолчанию имя контроллера — `default`. Если действие не задано в `defaultRoute`, то для его определения используется свойство `defaultAction` контроллера. В данном примере полный маршрут будет `user/default/index`.

Маршрут `catchAll`

Иногда возникает необходимость временно перевести приложение в режим обслуживания и отображать одно информационное сообщение для всех запросов. Существует много вариантов реализации этой задачи. Но одним из самых простых, является использование свойства `yii\web\Application::$catchAll`, например так:

```
[
    // ...
    'catchAll' => ['site/offline'],
];
```

В данном случае, действие `site/offline` будет обрабатывать все входящие запросы.

Свойство `catchAll` должно принимать массив, первый элемент которого определяет маршрут, а остальные элементы (пары ключ-значение) определяют параметры, передаваемые действию.

4.3.3 Создание URL

Для создания разных видов URL из заданных маршрутов и параметров, Yii предоставляет метод-помощник `yii\helpers\Url::to()`. Примеры:

```
use yii\helpers\Url;

// создает URL для маршрута: /index.php?r=post/index
echo Url::to(['post/index']);

// создает URL для маршрута с параметрами: /index.php?r=post/view&id=100
echo Url::to(['post/view', 'id' => 100]);

// создает якорный URL: /index.php?r=post/view&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// создает абсолютный URL: http://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], true);
```

```
// создает абсолютный URL с использованием схемы https: https://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], 'https');
```

Обратите внимание, что в последнем примере подразумевается использование обычного формата URL. При использовании ЧПУ, будут созданы другие URL, соответствующие правилам создания URL.

Маршрут, переданный методу `yii\helpers\Url::to()`, является контекстно зависимым. Он может быть *относительным* или *абсолютным*, в зависимости от следующих правил:

- Если маршрут является пустой строкой, будет использован текущий маршрут;
- Если маршрут не содержит слешей вообще, он рассматривается как *идентификатор* действия текущего контроллера и будет дополнен значением `uniqueId` текущего контроллера в качестве префикса;
- Если маршрут не содержит слеша в начале, он будет рассматриваться как маршрут относительно текущего модуля и будет дополнен значением `uniqueId` текущего модуля, в качестве префикса.

Начиная с версии 2.0.2, при составлении маршрутов, стало возможным использовать *псевдонимы*. В таком случае, псевдоним будет преобразован в маршрут, который будет использован для создания URL по правилам, указанным выше.

Для примера, будем считать, что текущим модулем является `admin`, а текущим контроллером - `post`,

```
use yii\helpers\Url;

// запрошенный маршрут: /index.php?r=admin/post/index
echo Url::to(['']);

// относительный маршрут с указанием только идентификатора действия: /index.php?r=admin/post/index
echo Url::to(['index']);

// относительный маршрут: /index.php?r=admin/post/index
echo Url::to(['post/index']);

// абсолютный маршрут: /index.php?r=post/index
echo Url::to(['/post/index']);

// /index.php?r=post/index    псевдоним "@posts" определен как "/post/index"
echo Url::to(['@posts']);
```

В основе реализации метода `yii\helpers\Url::to()` лежит использование двух методов компонента URL `manager: createUrl()` и `createAbsoluteUrl()`. Ниже будут рассмотрены способы конфигурации URL `manager` для создания URL в различных форматах.

Метод `yii\helpers\Url::to()`, так же, поддерживает создание URL не связанных с маршрутами приложения. В данном случае, нужно передать в качестве первого параметра строку, а не массив. Например,

```
use yii\helpers\Url;

// запрошенный URL: /index.php?r=admin/post/index
echo Url::to();

// URL из псевдонима: http://example.com
Yii::setAlias('@example', 'http://example.com/');
echo Url::to('@example');

// абсолютный URL: http://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

Кроме метода `to()`, класс `yii\helpers\Url` предоставляет и другие удобные методы для создания URL. Например,

```
use yii\helpers\Url;

// домашний URL: /index.php?r=site/index
echo Url::home();

// базовый URL, удобно использовать в случае, когда приложение расположено
// в подкаталоге
// относительно корневого каталога Веб сервера
echo Url::base();

// канонический URL запрошенного URL
// подробнее https://support.google.com/webmasters/answer/139066?hl=ru
echo Url::canonical();

// запомнить запрошенный URL и восстановить его при следующих запросах
Url::remember();
echo Url::previous();
```

4.3.4 Использование человекопонятных URL

Для активации ЧПУ, необходимо настроить компонент `urlManager` в конфигурации приложения следующим образом:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```


Свойство `enablePrettyUrl` является ключевым, активирует формат ЧПУ. Остальные свойства не обязательные. Однако, в примере выше, показан самый популярный вариант конфигурации ЧПУ.

- **showScriptName**: это свойство определяет необходимость включения имени входного скрипта в создаваемый URL. Например, при его значении `false`, вместо `/index.php/post/100`, будет сгенерирован URL `/post/100`.
- **enableStrictParsing**: это свойство позволяет включить строгий разбор URL. Если строгий разбор URL включен, запрошенный URL должен соответствовать хотя бы одному из правил, иначе будет вызвано исключение `yii\web\NotFoundHttpException`. Если строгий разбор URL отключен и ни одно из правил не подходит для разбора запрошенного URL, часть этого URL, представляющая путь, будет использована как маршрут.
- **rules**: это свойство содержит набор правил для разбора и создания URL. Это основное свойство, с которым нужно работать, что бы URL создавались в формате, соответствующем требованиям приложения.

Примечание: Для того, чтобы скрыть имя входного скрипта в создаваемых URL, кроме установки значения свойства `showScriptName` в `false`, необходимо настроить Веб сервер, чтобы он мог правильно определять PHP скрипт, который должен быть запущен, если в запрошенном URL он не указан явно. Рекомендованные настройки для Apache и Nginx описаны в разделе Установка Yii.

Правила URL

Правила URL - это экземпляр класса `yii\web\UrlRule` или класса, унаследованного от него. Каждое правило состоит из шаблона, используемого для поиска пути в запрошенном URL, маршрута и нескольких параметров запроса. Правило может быть использовано для разбора запроса в том случае, если шаблон правила совпадает с запрошенным URL. Правило может быть использовано для создания URL в том случае, если его маршрут и параметры запроса совпадают с заданными.

При включенном режиме ЧПУ, компонент URL `manager` использует правила URL, содержащиеся в его свойстве `rules`, для разбора входящих запросов и создания URL. Обычно, при разборе входящего запроса, URL `manager` проверяет все правила в порядке их следования, до *первого* правила, соответствующего запрошенному URL. Найденное правило используется для разбора URL на маршрут и параметры запроса. Аналогично для создания URL компонент URL `manager` ищет первое правило, соответствующее заданному маршруту и параметрам и использует его

для создания URL.

Правила задаются ассоциативным массивом, где ключи определяют шаблоны, а значения соответствующие маршруты. Каждая пара шаблон-маршрут составляет правило разбора URL. Например, следующие **правила** определяют два правила разбора URL. Первое правило задает соответствие URL `post` маршруту `post/index`. Второе правило задает соответствие URL, соответствующего регулярному выражению `post/(\d+)` маршруту `post/view` и параметру `id`.

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

Примечание: Шаблон правила используется для поиска соответствия с частью URL, определяющей путь. Например, в URL `/index.php/post/100?source=ad` путь определяет часть `post/100` (начальный и конечный слеш игнорируются), соответствующая регулярному выражению `post/(\d+)`.

Правила URL можно определять не только в виде пар шаблон-маршрут, но и в виде массива. Каждый массив используется для определения одного правила. Такой вид определения правил используется в случаях, когда необходимо указать другие параметры правила URL. Например,

```
[
    // другие... правила URL...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

По умолчанию, если в конфигурации правила URL не указан явно параметр `class`, будет создано правило класса `yii\web\UrlRule`.

Именованные параметры

Правило URL может содержать несколько именованных параметров запроса, которые указываются в шаблоне в следующем формате: `<ParamName:RegExp>`, где `ParamName` определяет имя параметра, а `RegExp` - необязательное регулярное выражение, используемое для определения значения параметра. В случае, если `RegExp` не указан, значением параметра будет любая последовательность символов кроме слешей.

Примечание: Возможно указание только регулярного выражения для параметров. В таком случае, остальная часть шаблона будет считаться простым текстом.

После разбора URL, параметры запроса, соответствующие шаблону правила, будут доступны в массиве `$_GET` через компонент приложения `request`. При создании URL, значения указанных параметров будут вставлены в URL в соответствии с шаблоном правила.

Рассмотрим несколько примеров работы с именованными параметрами. Допустим, мы определили следующие три правила URL:

```
[
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

При разборе следующих URL:

- `/index.php/posts` будет разобран в маршрут `post/index` при помощи второго правила;
- `/index.php/posts/2014/php` будет разобран на маршрут `post/index` и параметры `year` со значением `2014`, `category` со значением `php` при помощи первого правила;
- `/index.php/post/100` будет разобран на маршрут `post/view` и параметр `id` со значением `100` при помощи третьего правила;
- `/index.php/posts/php` вызовет исключение `yii\web\NotFoundHttpException`, если `yii\web\UrlManager::$enableStrictParsing` имеет значение `true`, так как правило для разбора данного URL отсутствует. Если `yii\web\UrlManager::$enableStrictParsing` имеет значение `false` (по умолчанию), значение `posts/php` будет возвращено в качестве маршрута.

При создании URL:

- `Url::to(['post/index'])` создаст `/index.php/posts` при помощи второго правила;
- `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` создаст `/index.php/posts/2014/php` при помощи первого правила;
- `Url::to(['post/view', 'id' => 100])` создаст `/index.php/post/100` при помощи третьего правила;
- `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` создаст `/index.php/post/100?source=ad` при помощи третьего правила. Параметр `source` не указан в правиле, поэтому он добавлен в созданный URL в качестве параметра запроса.
- `Url::to(['post/index', 'category' => 'php'])` создаст `/index.php/post/index?category=php` без использования правил. При отсутствии подходящего правила, URL будет создан простым соединением маршрута, как части пути, и параметров, как части запроса.

Параметры в маршрутах

В маршруте правила URL возможно указание имен параметров. Это позволяет использовать правило URL для обработки нескольких маршрутов. Например, следующие правила содержат параметры `controller` и `action` в маршрутах.

```
[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
    => '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

Для разбора URL `/index.php/comment/100/create` будет использовано первое правило, которое установит значения параметров `controller` равным `comment` и `action` равным `create`. Таким образом, маршрут `<controller>/<action>` дубет разрешен в `comment/create`.

Аналогично, для маршрута `comment/index`, при помощи третьего правила, будет создан URL `comment/index`.

Примечание: Использование параметров в маршрутах позволяет значительно уменьшить количество правил URL и улучшить производительность компонента `URL manager`.

По умолчанию, все параметры, указанные в правиле, являются обязательными. Если запрошенный URL не содержит обязательный параметр, или если URL создается без обязательного параметра, данное правило не будет применено. Свойство `yii\web\UrlRule::$defaults` позволяет сделать нужные параметры не обязательными. Параметры, перечисленные в данном свойстве, будут иметь заданные значения, в случае если они пропущены.

В следующем правиле описаны необязательные параметры `page` и `tag`, которые примут значения 1 и пустая строка в случае, если они будут пропущены.

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

Выше приведенное правило может быть использовано для разбора или создания следующих URL:

- `/index.php/posts: page равно 1, tag равно ''`.
- `/index.php/posts/2: page равно 2, tag равно ''`.
- `/index.php/posts/2/news: page равно 2, tag равно 'news'`.

- `/index.php/posts/news`: `page` равно 1, `tag` равно `'news'`.

Без использования необязательных параметров понадобилось бы создать 4 правила для достижения того же результата.

Правила с именами серверов

Существует возможность включать имена серверов в шаблон правил URL. Главным образом, это удобно, когда требуется разное поведение приложения, в зависимости от разных имен Веб серверов. Например, следующее правило позволит разобрать URL `http://admin.example.com/login` в маршрут `admin/user/login` и `http://www.example.com/login` в `site/login`.

```
[
    'http://admin.example.com/login' => 'admin/user/login',
    'http://www.example.com/login' => 'site/login',
]
```

Также возможно комбинирование параметров и имени сервера для динамического извлечения данных из него. Например, следующее правило позволит разобрать URL `http://en.example.com/posts` на маршрут и параметр `language=en`.

```
[
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

Примечание: Правила, содержащие имя сервера, НЕ должны содержать в шаблоне подкаталог пути ко входному скрипту. Например, если приложение расположено в `http://www.example.com/sandbox/blog`, шаблон должен быть `http://www.example.com/posts`, вместо `http://www.example.com/sandbox/blog/posts`. Это позволит изменять расположение приложения без необходимости внесения изменений в его код.

Суффиксы в URL

Компонент предоставляет возможность добавления к URL суффиксов. Например, можно добавить к URL `.html`, что бы они выглядели как статические HTML страницы; можно добавить к URL суффикс `.json`, для указания на ожидаемый тип данных ответа. Настроить суффиксы в URL можно при помощи соответствующего свойства `yii\web\UrlManager::$suffix` в конфигурации приложения:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
        ],
    ],
]
```

```
        'suffix' => '.html',
        'rules' => [
            // ...
        ],
    ],
],
]
```

Данная конфигурация позволяет компоненту `URL manager` разбирать и создавать URL с суффиксом `.html`.

Подсказка: При установке суффикса `/`, все URL будут заканчиваться слешем.

Примечание: При настроенном суффиксе, все URL не содержащие этот суффикс будут расценены как неизвестные URL. Такое поведение рекомендовано для SEO (поисковая оптимизация).

Иногда возникает необходимость использовать разные суффиксы для разных URL. Добиться этого можно настройкой свойства `suffix` у каждого правила. Когда это свойство установлено, оно имеет приоритет перед общей конфигурацией компонента `URL manager`. Например, следующая конфигурация содержит правило URL, которое использует `.json` в качестве суффикса вместо глобального `.html`.

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
],
]
```

Нормализация URL

Начиная с версии 2.0.10 `UrlManager` может быть настроен на использование `yii\web\UrlNormalizer`, что позволяет справиться с вариациями одного и того же URL с присутствующим или отсутствующим слешем

в конце. Технически `http://example.com/path` и `http://example.com/path/` являются разными URL, отдача одинакового содержимого в обоих вариантах может негативно повлиять на SEO. По умолчанию нормализатор заменяет повторяющиеся слешы на один и либо убирает, либо добавляет завершающие слешы в зависимости от суффикса и производит редирект 301³ на нормализованный URL. Нормализатор может быть настроен как глобально для менеджера URL, так и индивидуально для каждого правила. По умолчанию все правила используют нормализатор, заданный в менеджере URL. Вы можете выставить `yii\web\UrlRule::$normalizer` в `false` для отключения нормализации для конкретного правила.

Ниже приведён пример конфигурации `UrlNormalizer`:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'suffix' => '.html',
            'normalizer' => [
                'class' => 'yii\web\UrlNormalizer',
                'action' => UrlNormalizer::ACTION_REDIRECT_TEMPORARY, //
используем временный редирект вместо постоянного
            ],
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '/',
                    'normalizer' => false, // отключаем нормализатор для
этого правила
                ],
                [
                    'pattern' => 'tags',
                    'route' => 'tag/index',
                    'normalizer' => [
                        'collapseSlashes' => false, // не убираем
дублирующиеся слешы для этого правила
                    ],
                ],
            ],
        ],
    ],
]
```

Примечание: по умолчанию `yii\web\UrlManager::$normalizer` отключен. Чтобы использовать нормализацию его необходимо сконфигурировать.

³https://en.wikipedia.org/wiki/HTTP_301

HTTP методы

При реализации RESTful API, зачастую бывает необходимость в том, чтобы один и тот же URL был разобран в разные маршруты, в зависимости от HTTP метода запроса. Это легко достигается указанием HTTP методов, поддерживаемых правилом в начале шаблона. Если правило поддерживает несколько HTTP методов, их имена разделяются запятыми. Например, следующие правила имеют шаблон `post/<id:\d+>` с разными поддерживаемыми HTTP методами. Запрос `PUT post/100` будет разобран в маршрут `post/create`, в то время, как запрос `GET post/100` будет разобран в `post/view`.

```
[
    'PUT,POST post/<id:\d+>' => 'post/create',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

Примечание: Если правило URL содержит HTTP метод в шаблоне, это правило будет использовано только при разборе URL. Такое правило не будет учитываться компонентом `URL manager` при создании URL.

Подсказка: Для упрощения маршрутизации RESTful API, Yii предоставляет специальный класс `yii\rest\UrlRule`, который достаточно эффективен и предоставляет такие удобные возможности, как автоматическое приведение идентификаторов контроллеров к множественной форме. Более подробную информацию можно найти в разделе Веб-сервисы REST Роутинг.

Гибкая настройка правил

В предыдущих примерах, преимущественно, приводились правила URL, заданные парами шаблон-маршрут. Это самый распространенный, краткий формат. В некоторых случаях возникает необходимость более гибкой настройки правил, например указание суффикса при помощи свойства `yii\web\UrlRule::$suffix`. Пример конфигурации правила URL при помощи массива был рассмотрен в главе Суффиксы в URL:

```
[
    // другие... правила URL...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```


Информация: По умолчанию, если в конфигурации правила явно не задан параметр `class`, будет создано правило класса `yii\web\UrlRule`.

Добавление правил URL динамически

Правила URL могут быть динамически добавлены в компонент `URL manager`. Часто это необходимо подключаемым модулям для настройки своих правил URL. Для того, чтобы динамически добавленные правила могли влиять на процесс роутинга, они должны быть добавлены в процессе [предзагрузки](#). В частности, модули должны реализовываться интерфейс `yii\base\BootstrapInterface` и добавлять правила в методе `bootstrap()`, например:

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // правила URL описываются здесь
    ], false);
}
```

Так же, необходимо включить данный модуль в `yii\web\Application::bootstrap()`, чтобы он смог участвовать в процессе [предзагрузки](#).

Создание классов правил

Несмотря на то, что встроенный класс `yii\web\UrlRule` достаточно функционален для большинства проектов, иногда возникает необходимость в создании своего класса правил URL. Например, на сайте продавца автомобилей существует необходимость поддержки URL в таком формате: `/Manufacturer/Model`, где и `Manufacturer` и `Model` должны соответствовать данным, хранящимся в базе данных. Стандартный класс `yii\web\UrlRule` не подойдет, так как он рассчитан на работу со статичными шаблонами.

Для решения данной проблемы можно создать такой класс правила URL.

```
namespace app\components;

use yii\web\UrlRuleInterface;
use yii\base\Object;

class CarUrlRule extends Object implements UrlRuleInterface
{

    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {

```

```

        return $params['manufacturer'];
    }
}
return false; // данное правило не применимо
}

public function parseRequest($manager, $request)
{
    $pathInfo = $request->getPathInfo();
    if (preg_match('%^(\\w+)/?(\\w+)?$%', $pathInfo, $matches)) {
        // Ищем совпадения $matches[1] и $matches[3]
        // с данными manufacturer и model в базе данных
        // Если нашли, устанавливаем $params['manufacturer'] и/или/
        $params['model']
        // и возвращаем ['car/index', $params]
    }
    return false; // данное правило не применимо
}
}

```

И использовать новый класс `yii\web\UrlManager::$rules` при определении правил URL:

```

[
    // другие... правила...

    [
        'class' => 'app\components\CarUrlRule',
        // настройка... других параметров правила...
    ],
]

```

4.3.5 Производительность

При разработке сложных Веб приложений, важно оптимизировать правила URL так, чтобы разбор запросов и создание URL занимали минимальное время.

Использование параметров в маршрутах позволяет уменьшить количество правил, что значительно увеличивает производительность.

При разборе или создании URL, компонент `URL manager` проверяет правила в порядке их определения. Поэтому следует более узконаправленные и/или часто используемые правила размещать раньше прочих.

В случае, если несколько правил имеют один и тот же префикс в шаблоне или маршруте, можно рассмотреть использование `yii\web\GroupUrlRule`, что позволит компоненту `URL manager` более эффективно обрабатывать правила группами. Часто это бывает полезно в случае, если приложение состоит из модулей, каждый из которых имеет свой набор правил с идентификатором модуля в качестве общего префикса.

4.4 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`.

4.4.1 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"
$params = $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`.

4.4.2 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 ($request->isAjax) { /* текущий запрос является AJAX запросом */ }
if ($request->isGet) { /* текущий запрос является GET запросом */ }
if ($request->isPost) { /* текущий запрос является POST запросом */ }
if ($request->isPut) { /* текущий запрос является PUT запросом */ }
```

4.4.3 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.

4.4.4 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')) { /* в запросе есть 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.

4.4.5 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;
```

4.5 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.

4.5.1 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 состояния⁴. Например, чтобы показать, что запрос был успешно обработан, вы можете установить значение кода состояния равным 200:

```
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 предусмотрены следующие HTTP-исключения:

- `yii\web\BadRequestHttpException`: код состояния 400.

⁴<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- `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);
```

4.5.2 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()`.

4.5.3 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.

4.5.4 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.

4.5.5 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-Sendfile⁵
- Lighttpd v1.4: X-LIGHTTPD-send-file⁶
- Lighttpd v1.5: X-Sendfile⁷
- Nginx: X-Accel-Redirect⁸
- Cherokee: X-Sendfile and X-Accel-Redirect⁹

4.5.6 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()`.
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.

⁵http://tn123.org/mod_xsendfile

⁶<http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file>

⁷<http://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file>

⁸<http://wiki.nginx.org/XSendfile>

⁹http://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

4.6 Сессии и куки

Сессии и куки позволяют сохранять пользовательские данные между запросами. При использовании чистого PHP можно получить доступ к этим данным через глобальные переменные `$_SESSION` и `$_COOKIE`, соответственно. Yii инкапсулирует сессии и куки в объекты, что дает возможность обращаться к ним в объектно-ориентированном стиле и дает дополнительное удобство в работе.

4.6.1 Сессии

По аналогии с [запросами](#) и [ответами](#), к сессии можно получить доступ через `session` компонент приложения, который по умолчанию является экземпляром `yii\web\Session`.

Открытие и закрытие сессии

Открыть и закрыть сессию можно следующим образом:

```
$session = Yii::$app->session;

// проверяем что сессия уже открыта
if ($session->isActive) ...

// открываем сессию
$session->open();

// закрываем сессию
$session->close();

// уничтожаем сессию и все связанные с ней данные.
$session->destroy();
```

Можно вызывать `open()` и `close()` многократно без возникновения ошибок; внутри компонента все методы проверяют сессию на факт того, открыта она или нет.

Доступ к данным сессии

Получить доступ к сохраненным в сессию данным можно следующим образом:

```
$session = Yii::$app->session;

// получение переменной из сессии. Следующие способы использования
// эквивалентны:
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;

// запись переменной в сессию. Следующие способы использования эквивалентны:
```

```
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';

// Удаление переменной из сессии. Следующие способы использования
эквивалентны:
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// проверка на существование переменной в сессии. Следующие способы
использования эквивалентны:
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// Обход всех переменных в сессии. Следующие способы использования
эквивалентны:
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

Информация: При получении данных из сессии через компонент `session`, сессия будет автоматически открыта, если она не была открыта до этого. В этом заключается отличие от получения данных из глобальной переменной `$_SESSION`, которое требует обязательного вызова `session_start()`.

При работе с сессионными данными, являющимися массивами, компонент `session` имеет ограничение, запрещающее прямую модификацию отдельных элементов массива. Например,

```
$session = Yii::$app->session;

// следующий код НЕ БУДЕТ работать
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// а этот будет:
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// этот код также будет работать:
echo $session['captcha']['lifetime'];
```

Для решения этой проблемы можно использовать следующие обходные приемы:

```
$session = Yii::$app->session;

// прямое использование $_SESSION убедитесь(, что Yii::$app->session->open()
был вызван)
```

```
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;

// получите весь массив, модифицируйте и сохраните обратно в сессию
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// используйте ArrayObject вместо массива
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// записывайте данные с ключами, имеющими одинаковый префикс
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

Для улучшения производительности и читаемости кода рекомендуется использовать последний прием. Другими словами, вместо того, чтобы хранить массив как одну переменную сессии, мы сохраняем каждый элемент массива как обычную сессионную переменную с общим префиксом.

Пользовательское хранилище для сессии

По умолчанию класс `yii\web\Session` сохраняет данные сессии в виде файлов на сервере. Однако Yii предоставляет ряд классов, которые реализуют различные способы хранения данных сессии:

- `yii\web\DbSession`: сохраняет данные сессии в базе данных.
- `yii\web\CacheSession`: хранение данных сессии в предварительно сконфигурированном компоненте кэша [кэш](#).
- `yii\redis\Session`: хранение данных сессии в [redis](#)¹⁰.
- `yii\mongodb\Session`: хранение сессии в [MongoDB](#)¹¹.

Все эти классы поддерживают одинаковый набор методов API. В результате вы можете переключаться между различными хранилищами сессий без модификации кода приложения.

Примечание: Если вы хотите получить данные из переменной `$_SESSION` при использовании пользовательского хранилища, вы должны быть уверены, что сессия уже стартовала `yii\web\Session::open()`, в связи с тем, что обработчики хранения пользовательских сессий регистрируются в этом методе.

Чтобы узнать, как настроить и использовать эти компоненты, обратитесь к документации по API. Ниже приведен пример конфигурации `yii\web\DbSession` для использования базы данных для хранения сессии:

¹⁰<http://redis.io/>

¹¹<http://www.mongodb.org/>

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb', // ID компонента для взаимодействия с БД.
            По умолчанию 'db'.
            // 'sessionTable' => 'my_session', // название таблицы для
            хранения данных сессии. По умолчанию 'session'.
        ],
    ],
];
```

Также необходимо создать таблицу для хранения данных сессии:

```
CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

где 'BLOB' соответствует типу данных предпочитаемой вами DBMS. Ниже приведены примеры соответствия типов BLOB в наиболее популярных DBMS:

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

Примечание: В зависимости от настроек параметра `session.hash_function` в вашем `php.ini`, может понадобиться изменить длину поля `id`. Например, если `session.hash_function=sha256`, нужно установить длину поля в 64 вместо 40.

Flash-сообщения

Flash-сообщения - это особый тип данных в сессии, которые устанавливаются один раз во время запроса и доступны только на протяжении следующего запроса, затем они автоматически удаляются. Такой способ хранения информации в сессии наиболее часто используется для реализации сообщений, которые будут отображены конечному пользователю один раз, например подтверждение об успешной отправке формы.

Установить и получить flash-сообщения можно через компонент приложения `session`. Например:

```
$session = Yii::$app->session;

// Запрос #1
// установка flash-сообщения с названием "postDeleted"
$session->setFlash('postDeleted', 'Вы успешно удалили пост.');
```

```
// Запрос #2
```

```
// отображение flash-сообщения "postDeleted"
echo $session->getFlash('postDeleted');

// Запрос #3
// переменная $result будет иметь значение false, так как flash-сообщение
    было автоматически удалено
$result = $session->hasFlash('postDeleted');
```

Так как flash-сообщения хранятся в сессии как обычные данные, в них можно записывать произвольную информацию, и она будет доступна лишь в следующем запросе.

При вызове `yii\web\Session::setFlash()`, происходит перезаписывание flash-сообщений с таким же названием. Для того, чтобы добавить новые данные к уже существующему flash-сообщению, необходимо вызывать `yii\web\Session::addFlash()`. Например:

```
$session = Yii::$app->session;

// Запрос #1
// добавить новое flash-сообщение с названием "alerts"
$session->addFlash('alerts', 'Вы успешно удалили пост.');
```

```
$session->addFlash('alerts', 'Вы успешно добавили нового друга.');
```

```
$session->addFlash('alerts', 'Благодарим.');
```

```
// Запрос #2
// Переменная $alerts теперь содержит массив flash-сообщений с названием "
    alerts"
$alerts = $session->getFlash('alerts');
```

Примечание: Старайтесь не использовать `yii\web\Session::setFlash()` совместно с `yii\web\Session::addFlash()` для flash-сообщений с одинаковым названием. Это связано с тем, что последний метод автоматически преобразует хранимые данные в массив, чтобы иметь возможность хранить и добавлять новые данные в flash-сообщения с тем же названием. В результате, при вызове `yii\web\Session::getFlash()` можно обнаружить, что возвращается массив, в то время как ожидалась строка.

4.6.2 Куки

Yii представляет каждую куку как объект `yii\web\Cookie`. Оба компонента приложения `yii\web\Request` и `yii\web\Response` поддерживают коллекции кук через свойство `cookies`. В первом случае коллекция кук является их представлением из HTTP-запроса, во втором - представляет куки, которые будут отправлены пользователю.

Чтение кук

Получить куки из текущего запроса можно следующим образом:

```
// получение коллекции кук (yii\web\CookieCollection) из компонента "request"
$cookies = Yii::$app->request->cookies;

// получение куки с названием "language". Если кука не существует, "en"
// будет возвращено как значение по-умолчанию.
$language = $cookies->getValue('language', 'en');

// альтернативный способ получения куки "language"
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// теперь переменную $cookies можно использовать как массив
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// проверка на существование куки "language"
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

Отправка кук

Отправить куку конечному пользователю можно следующим образом:

```
// получение коллекции (yii\web\CookieCollection) из компонента "response"
$cookies = Yii::$app->response->cookies;

// добавление новой куки в HTTP-ответ
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// удаление куки...
$cookies->remove('language');
// что... эквивалентно следующему:
unset($cookies['language']);
```

Кроме свойств `name` и `value`, класс `yii\web\Cookie` также предоставляет ряд свойств для получения информации о куках: `domain`, `expire`. Эти свойства можно сконфигурировать и затем добавить куку в коллекцию для HTTP-ответа.

Примечание: Для большей безопасности значение свойства `yii\web\Cookie::$httpOnly` по умолчанию установлено в `true`. Это уменьшает риски доступа к защищенной куке на клиентской стороне (если браузер поддерживает такую возмож-

ность). Вы можете обратиться к `httpOnly` wiki¹² для дополнительной информации.

Валидация кук

Во время записи и чтения кук через компоненты `request` и `response`, как будет показано в двух последующих подразделах, фреймворк предоставляет автоматическую валидацию, которая обеспечивает защиту кук от модификации на стороне клиента. Это достигается за счет подписи каждой куки секретным ключом, позволяющим приложению распознать куку, которая была модифицирована на клиентской стороне. В таком случае кука НЕ БУДЕТ доступна через свойство `cookie collection` компонента `request`.

Примечание: Валидация кук защищает только от их модификации. Если валидация не была пройдена, получить доступ к кукам все еще можно через глобальную переменную `$_COOKIE`. Это связано с тем, что дополнительные пакеты и библиотеки могут манипулировать куками без вызова валидации, которую обеспечивает Yii.

По-умолчанию валидация кук включена. Её можно отключить, установив свойство `yii\web\Request::$enableCookieValidation` в `false`, однако мы настоятельно не рекомендуем это делать.

Примечание: Куки, которые напрямую читаются/пишутся через `$_COOKIE` и `setcookie()` НЕ БУДУТ валидироваться.

При использовании валидации кук необходимо указать значение свойства `yii\web\Request::$cookieValidationKey`, которое будет использовано для генерации вышеупомянутого секретного ключа. Это можно сделать, настроив компонент `request` в конфигурации приложения:

```
return [  
    'components' => [  
        'request' => [  
            'cookieValidationKey' => 'fill in a secret key here',  
        ],  
    ],  
];
```

Примечание: Свойство `cookieValidationKey` является секретным значением и должно быть известно только людям, которым вы доверяете. Не помещайте эту информацию под систему контроля версий.

¹²<https://www.owasp.org/index.php/HttpOnly>

4.7 Обработка ошибок

В состав Yii входит встроенный **обработчик ошибок**, делающий работу с ошибками гораздо более приятным занятием. А именно:

- Все не фатальные ошибки PHP (то есть warning, notice) конвертируются 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.

4.7.1 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 не фатальные 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\NotFoundException;  
  
throw new NotFoundException();
```

4.7.2 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 использованы 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  
}  
}
```

4.8 Логгирование

Yii предоставляет мощную, гибко настраиваемую и легко расширяемую систему логгирования. Эта система логгирования позволяет удобным способом сохранять сообщения разных типов и фильтровать их. Сообщения могут быть сохранены в файлы, базы данных или отправлены на email.

Использование Системы логгирования Yii включает следующие шаги:

- Запись сообщений лога в различных частях кода приложения;
- Настройка целей лога в конфигурации приложения;
- Изучение отфильтрованных сообщений лога, например, при помощи Отладчика Yii.

В данном разделе, будем рассматривать первые два шага.

4.8.1 Сообщения лога

Запись сообщений лога осуществляется вызовом одного из следующих методов:

- `Yii::trace()`: записывает сообщения для отслеживания выполнения кода приложения. Используется, в основном, при разработке.
- `Yii::info()`: записывает сообщение, содержащее какую-либо полезную информацию.
- `Yii::warning()`: записывает *тревожное* сообщение при возникновении неожиданного события.
- `Yii::error()`: записывает критическую ошибку, на которую нужно, как можно скорее, обратить внимание.

Эти методы позволяют записывать сообщения разных *уровней важности* и *категорий*. Они имеют одинаковое описание функции `function ($message, $category = 'application')`, где `$message` передает сообщение для записи, а `$category` - категорию сообщения. В следующем примере будет записано *trace* сообщение с категорией по умолчанию `application`:

```
Yii::trace('start calculating average revenue');
```

Примечание: Сообщение может быть как строкой так и объектом или массивом. За корректную работу с содержимым

сообщения отвечают цели лога. По умолчанию, если сообщение не является строкой, оно будет приведено к строковому типу при помощи `yii\helpers\VarDumper::export()`.

Для упрощения работы с сообщениями лога и их фильтрации, рекомендуется явно указывать подходящую категорию для каждого сообщения. Возможно использование иерархической системы именования категорий, что значительно упростит целям лога фильтрацию сообщений по категориям. Простым и эффективным способом именования категорий является использование магической PHP константы `__METHOD__`. Такой подход используется в ядре фреймворка Yii. Например,

```
Yii::trace('начало вычисления среднего дохода', __METHOD__);
```

Константа `__METHOD__` вычисляется как имя метода (включая полное имя класса), в котором она использована. Например, её значение будет вычислено как `'app\controllers\RevenueController::calculate'`, если показанный выше код вызывается в соответствующем методе.

Информация: методы логгирования, описанные выше являются, на самом деле, ярлыками для метода `log()` объекта **логгера**, который доступен как синглтон `Yii::getLogger()`. При определенном количестве записанных сообщений или завершении приложения, объект логгера вызывает `message dispatcher` для отправки записанных сообщений зарегистрированным целям логов.

4.8.2 Цели логов

Цель логов - это экземпляр класса `yii\log\Target` или класса, унаследованного от него. Цель фильтрует сообщения логов по уровню важности и категории, а затем выгружает их в соответствующее хранилище. Например, `database target` выгружает отфильтрованные сообщения логов в таблицу базы данных, а `email target` отправляет сообщения логов на заданные адреса email.

При помощи компонента приложения `log` возможна регистрация нескольких целей логов. Пример конфигурации приложения:

```
return [
    // Компонент "log" должен быть загружен на этапе предзагрузки
    'bootstrap' => ['log'],

    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ]
            ]
        ]
    ]
];
```



```

        ],
        [
            'class' => 'yii\log\EmailTarget',
            'levels' => ['error'],
            'categories' => ['yii\db\*'],
            'message' => [
                'from' => ['log@example.com'],
                'to' => ['admin@example.com', 'developer@example.com'],
                'subject' => 'Ошибки базы данных на сайте example.com'
            ],
        ],
    ],
],
];

```

Примечание: Компонент `log` должен быть загружен в процессе `предзагрузки`, тогда он сможет оперативно передавать сообщения целям логов. Поэтому он указан в массиве `bootstrap`.

В приведенном выше коде в свойстве `yii\log\Dispatcher::$targets` зарегистрированы две цели логов:

- первая цель выбирает ошибки и предупреждения и сохраняет их в базу данных;
- вторая цель выбирает ошибки с категорией, имя которой начинается с `yii\db\` и шлет сразу на два адреса email `admin@example.com` и `developer@example.com`.

На данный момент, Yii содержит следующие встроенные цели логов. В документации по API подробно описана настройка и использование этих классов.

- `yii\log\DbTarget`: сохраняет сообщения логов в таблицу базы данных.
- `yii\log\EmailTarget`: шлет сообщения логов на заранее указанный email.
- `yii\log\FileTarget`: сохраняет сообщения логов в файлы.
- `yii\log\SyslogTarget`: сохраняет сообщения логов в системный лог используя функцию PHP `syslog()`.

Дальше рассмотрим общие для этих четырех классов возможности.

Фильтрация сообщений

Для каждой цели можно настроить свойства `levels` и `categories`, которые указывают уровни важности и категории сообщений логов, которые цель должна обрабатывать.

Свойство `levels` принимает массив, содержащий одно или несколько следующих значений:

- `error`: соответствует сообщениям, сохраненным методом `Yii::error()`.
- `warning`: соответствует сообщениям, сохраненным методом `Yii::warning()`.
- `info`: соответствует сообщениям, сохраненным методом `Yii::info()`.
- `trace`: соответствует сообщениям, сохраненным методом `Yii::trace()`.
- `profile`: соответствует сообщениям, сохраненным методами `Yii::beginProfile()` и `Yii::endProfile()`, подробнее о которых написано в подразделе Профилирование производительности.

Если свойство `levels` не задано, цель логов будет обрабатывать сообщения с *любым* уровнем важности.

Свойство `categories` принимает массив, содержащий имена категорий или шаблоны. Цель будет обрабатывать только те сообщения, категория которых совпадает с одним из значений или шаблонов этого массива. Шаблон категории должен состоять из префикса имени категории и звездочки `*` на конце. Имя категории совпадает с шаблоном, если оно начинается с префикса шаблона. Например, `yii\db\Command::execute` и `yii\db\Command::query` используются в качестве имен категорий сообщений, записанных в классе `yii\db\Command`. Оба они совпадают с шаблоном `yii\db*`.

Если свойство `categories` не задано, цель будет обрабатывать сообщения любой категории.

Кроме списка включаемых категорий, заданного свойством `categories`, при помощи свойства `except` возможно задать список исключаемых категорий. Если категория сообщения совпадает со значением или шаблоном из списка исключаемых категорий, такое сообщение не будет обработано.

В следующем примере показан вариант конфигурации цели логов, которая должна обрабатывать только сообщения об ошибках и предупреждениях в категориях `yii\db*` и `yii\web\HttpException:*`, за исключением `yii\web\HttpException:404`.

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

Примечание: При обработке HTTP исключения *обработчиком ошибок*, сообщение будет сохранено с категорией вида `yii\web\HttpException:ErrorCode`. Например, исключение `yii`

`\web\NotFoundHttpException` вызовет сообщение об ошибке с категорией `yii\web\HttpException:404`.

Форматирование сообщений

Цели логов выгружают отфильтрованные сообщения в определенном формате. Например, цель класса `yii\log\FileTarget` сохранит сообщение следующего формата в файле `runtime/log/app.log`:

```
2014-10-04 18:10:15 [::1] [] [-][trace][yii\base\Module::getModule] Loading
module: debug
```

По умолчанию сообщения логов формируются методом `yii\log\Target::formatMessage()`:

```
Временная
метка [IP адрес] [ID пользователя] [ID сессии] Уровень [ важности Категория] []
Текст сообщения
```

Этот формат может быть изменен при помощи свойства `yii\log\Target::$prefix`, которое получает анонимную функцию, возвращающую нужный префикс сообщения. Например, следующий код позволяет настроить вывод идентификатор текущего пользователя в качестве префикса для всех сообщений.

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') : null
        ;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

Кроме префиксов сообщений, также возможно добавление общей информации для каждого набора сообщений лога. По умолчанию, включаются значения следующих глобальных PHP переменных: `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION` и `$_SERVER`. Эта возможность настраивается при помощи свойства `yii\log\Target::$logVars`, содержащего массив имен переменных, которые необходимо включить в лог. Например, следующий код позволяет настроить цель логов так, чтобы к сообщениям присоединялось только содержимое переменной `$_SERVER`.

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

При задании значением свойства `logVars` пустого массива, общая информация не будет выводиться. Для определения собственного алгоритма подключения общей информации, следует переопределить метод `yii\log\Target::getContextMessage()`.

Уровень отслеживания выполнения кода

В процессе разработки, часто бывает необходимость видеть источники сообщений. Для этого нужно использовать свойство `traceLevel` компонента `log`. Например,

```
return [  
    'bootstrap' => ['log'],  
    'components' => [  
        'log' => [  
            'traceLevel' => YII_DEBUG ? 3 : 0,  
            'targets' => [...],  
        ],  
    ],  
];
```

При такой настройке свойство `traceLevel` будет равно 3 при `YII_DEBUG` равном `true` и 0 при `YII_DEBUG` равном `false`. Это означает, что при включенном `YII_DEBUG`, каждое сообщение лога будет содержать до трех уровней стека вызовов, а при выключенном `YII_DEBUG` информация о стеке вызовов не будет включаться в лог.

Информация: Получение информации стека вызовов является не простым процессом. Поэтому такую возможность следует использовать только при разработке или отладке приложения.

Передача на обработку и выгрузка сообщений

Как упоминалось выше, сообщения логов обрабатываются в массиве объектом логгера. Для ограничения объема памяти, занятого этим массивом, при накоплении определенного числа сообщений, логгер передает их на обработку целям логов. Максимальное количество сообщений определяется свойством `flushInterval` компонента `log`:

```
return [  
    'bootstrap' => ['log'],  
    'components' => [  
        'log' => [  
            'flushInterval' => 100,    // по умолчанию 1000  
            'targets' => [...],  
        ],  
    ],  
];
```

Информация: При завершении приложения, так же происходит передача сообщений на обработку.

После передачи сообщений объектом логгера в цели логов, сообщения не выгружаются немедленно. Вместо этого, выгрузка сообщений про-

исходит когда цель логов накопит определенное количество фильтрованных сообщений. Максимальное количество сообщений определяется свойством `exportInterval` цели логов. Например,

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100, // по умолчанию 1000
]
```

Из-за того, что значения максимального количества сообщений для передачи и выгрузки по умолчанию достаточно велико, при вызове метода `Yii::trace()`, или любого другого метода логгирования, сообщение не появится сразу в файле или таблице базы данных. Такое поведение может стать проблемой, например, в консольных приложениях с большим временем исполнения. Для того, чтобы все сообщения логов сразу же попадали в лог, необходимо установить значения свойств `flushInterval` и `exportInterval` равными 1, например так:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

Примечание: Частая передача и выгрузка сообщений может сильно снизить производительность приложения.

Переключение целей логов

Свойство `enabled` отвечает за включение или отключение цели логов. Возможно управление этим свойством как в конфигурации приложения, так и при помощи следующего PHP кода:

```
Yii::$app->log->targets['file']->enabled = false;
```

В данном примере используется цель логов `file`, которая может быть настроена в конфигурации приложения следующим образом:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
```

```
        'file' => [
            'class' => 'yii\log\FileTarget',
        ],
        'db' => [
            'class' => 'yii\log\DbTarget',
        ],
    ],
],
],
];
```

Создание новых целей

Создание новой цели логов не является сложной задачей. В общем случае, нужно реализовать метод `yii\log\Target::export()`, выгружающий массив `yii\log\Target::$messages` в место хранения логов. Возможно использование метода `yii\log\Target::formatMessage()` для форматирования сообщения. Детали реализации можно посмотреть в исходном коде любого из классов целей логов, включенных в состав Yii.

4.8.3 Профилирование производительности

Профилирование производительности - это специальный тип сообщений логов, используемый для измерения времени выполнения определенных участков кода и определения проблем производительности. Например, класс `yii\db\Command` использует профилирование производительности для определения времени исполнения каждого запроса базы данных.

Для использования профилирования производительности нужно определить участок кода для измерения и *обернуть* его вызовами методов `Yii::beginProfile()` и `Yii::endProfile()`. Например,

```
\Yii::beginProfile('myBenchmark'); участок
... кода для профилирования...
\Yii::endProfile('myBenchmark');
```

где `myBenchmark` является уникальным идентификатором данного измеряемого участка кода. В дальнейшем, при изучении результатов профилирования, уникальный идентификатор поможет определить время выполнения соответствующего участка кода.

Очень важно соблюдать уровни вложенности пар `beginProfile` и `endProfile`. Например,

```
\Yii::beginProfile('block1');

// код для профилирования

\Yii::beginProfile('block2');
```

```
// другой код для профилирования
\Yii::endProfile('block2');

\Yii::endProfile('block1');
```

Если пропустить `\Yii::endProfile('block1')` или поменять местами `\Yii::endProfile('block1')` и `\Yii::endProfile('block2')`, профилирование производительности не будет работать.

Для каждого участка кода, будет записано сообщение лога с уровнем важности `profile`. Для сбора таких сообщений можно настроить цель логов или воспользоваться Отладчиком Yii, который имеет встроенную панель профилирования производительности, отображающую результаты измерений.

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 `yii\jui\DatePicker`, 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 придется 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 extends 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 РНР 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 *не статических* геттеров и/или сеттеров. Статические методы не будут обрабатываться подобным образом.
- Обычный вызов `property_exists()` не работает для магических свойств. Для них необходимо использовать `canGetProperty()` или `canSetProperty()`.

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:

- 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**.

¹<http://www.php.net/manual/ru/language.types.callable.php>

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 ($event) {
    var_dump($event->sender); // выводит "null"
});

Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

Обратите внимание, что в данном случае `$event->sender` имеет значение `null` вместо экземпляра класса, который инициировал событие.

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 Обработчики событий на уровне интерфейсов

Существует еще более абстрактный способ обработки событий. Вы можете создать отдельный интерфейс для общего события и реализовать его в классах, где это необходимо.

Например, создадим следующий интерфейс:

```
interface DanceEventInterface
{
    const EVENT_DANCE = 'dance';
}
```

И два класса, которые его реализовывают:

```
class Dog extends Component implements DanceEventInterface
{
    public function meetBuddy()
    {
        echo "Woof!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}

class Developer extends Component implements DanceEventInterface
{
    public function testsPassed()
    {
        echo "Yay!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}
```

Для обработки события EVENT_DANCE, инициализированного любым из этих классов, вызовите `Event::on()`, передав ему в качестве первого параметра имя интерфейса.

```
Event::on('DanceEventInterface', DanceEventInterface::EVENT_DANCE, function
($event) {
    Yii::trace($event->sender->className . ' just danced'); // Оставит
    запись в журнале о том, что кто-то танцевал
});
```

Вы можете также инициализировать эти события:

```
Event::trigger(DanceEventInterface::className(), DanceEventInterface::
EVENT_DANCE);
```

Однако, невозможно инициализировать событие во всех классах, которые реализуют интерфейс:

```
// НЕ БУДЕТ РАБОТАТЬ
Event::trigger('DanceEventInterface', DanceEventInterface::EVENT_DANCE); //
ошибка
```

Отсоединить обработчик события можно с помощью метода `Event::off()`. Например:

```
// отсоединяет $handler
Event::off('DanceEventInterface', DanceEventInterface::EVENT_DANCE, $handler
);

// отсоединяются все обработчики DanceEventInterface::EVENT_DANCE
Event::off('DanceEventInterface', DanceEventInterface::EVENT_DANCE);
```


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;

    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`.

Примечание: При переопределении метода поведения `yii\base\Behavior::__get()` и/или `yii\base\Behavior::__set()` необходимо также переопределить `yii\base\Behavior::canGetProperty()` и/или `yii\base\Behavior::canSetProperty()`.

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 foo() объявленный в классе поведения
$component->foo();
```

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 через `insert()`, `update()` или `save()`.

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'],
                ],
            ],
        ],
    }
}
```

```

        // если вместо метки времени UNIX используется datetime:
        // 'value' => new Expression('NOW()'),
    ],
    ],
}
}

```

Конфигурация выше описывает следующее:

- при вставке новой записи поведение должно присвоить текущую метку времени UNIX атрибутам `created_at` и `updated_at`;
- при обновлении существующей записи поведение должно присвоить текущую метку времени UNIX атрибуту `updated_at`.

Примечание: Для того, чтобы приведённая выше конфигурация работала с MySQL, тип `created_at` и `updated_at` должен быть `int(11)`. В нём будет храниться UNIX timestamp.

Теперь, если сохранить объект `User`, то в его атрибуты `created_at` и `updated_at` будут автоматически установлены значения метки времени UNIX 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 Другие поведения

Кроме затронутых выше, есть и другие уже реализованные поведения. Как встроенные, так и сторонние:

- `yii\behaviors\BlameableBehavior` - автоматически заполняет указанные атрибуты ID текущего пользователя.
- `yii\behaviors\SluggableBehavior` - автоматически заполняет указанный атрибут пригодным для URL текстом, получаемым из другого атрибута.
- `yii\behaviors\AttributeBehavior` - автоматически задаёт указанное значение одному или нескольким атрибутам `ActiveRecord` при срабатывании определённых событий.
- `yii2tech\ar\softdelete\SoftDeleteBehavior`³ - предоставляет методы для «мягкого» удаления и восстановления `ActiveRecord`. То есть

³<https://github.com/yii2tech/ar-softdelete>

выставляет статус или флаг, который показывает, что запись удалена.

- `yii2tech\ar\position\PositionBehavior`⁴ - позволяет управлять порядком записей через специальные методы. Информация сохраняется в целочисленном поле.

5.4.8 Сравнение с трейтами

Несмотря на то, что поведения схожи с трейтами⁵ 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 копипейст на уровне языка. Они наследование не поддерживают.

Поведения могут быть прикреплены и отвязаны от компонента динамически, без необходимости модифицирования класса компонента. Для использования 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**

⁴<https://github.com/yii2tech/ar-position>

⁵<http://ru2.php.net/manual/ru/language.oop5.traits.php>

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',  
    '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::begin()`. 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 являющийся 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`: 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 `yii2-jui`, то вам автоматически станет доступен псевдоним `@yii/jui`. Он создаётся на этапе [первоначальной загрузки](#) (`bootstrapping`) примерно так:

```
Yii::setAlias('@yii/jui', 'VendorPath/yii2-jui');
```

5.7 Автозагрузка классов

Поиск и подключение файлов классов в Yii реализовано при помощи автозагрузки классов⁶. Фреймворк предоставляет свой быстрый совместимый с PSR-4⁷ 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.

⁶<http://www.php.net/manual/ru/language.oop5.autoload.php>

⁷<https://github.com/php-fig/fig-standards/blob/master/proposed/psr-4-autoloader/psr-4-autoloader.md>

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 имён⁸ (то есть `foo\bar\MyClass`).
- 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.

⁸<http://php.net/manual/ru/language.namespaces.php>

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 Composer⁹.

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

⁹<https://getcomposer.org/doc/04-schema.md#autoload>

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.

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 Service Locator (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 приведённый 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 Фаулера¹⁰ 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 метод;
- 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);
```

¹⁰<http://martinfowler.com/articles/injection.html>

Внедрение зависимости через метод

Обычно зависимости класса передаются в конструктор и становятся доступными внутри класса в течение всего времени его существования. При помощи инъекции через метод возможно задать зависимость, которая необходима в единственном методе класса. Передавать такую зависимость через конструктор либо невозможно, либо это влечёт за собой ненужные накладные расходы в большинстве случаев.

Метод класса может быть определён так же, как `doSomething()` в примере ниже:

```
class MyClass extends \yii\base\Component
{
    public function __construct(/*Легковесные зависимости тут*/, $config = [])
    {
        // ...
    }

    public function doSomething($param1, \my\heavy\Dependency $something)
    {
        // Работаем с $something
    }
}
```

Метод можно вызвать либо передав экземпляр `\my\heavy\Dependency` самостоятельно, либо использовав `yii\di\Container::invoke()`:

```
$obj = new MyClass(/*...*/);
Yii::$container->invoke([$obj, 'doSomething'], ['param1' => 42]); //
    $something будет предоставлено DI-контейнером
```

Внедрение зависимости через сеттер и свойство

Внедрение зависимости через сеттер и свойство поддерживается через [конфигурации](#). 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,

```
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()
    {
        $foo = new Foo(new Bar);
        // ... дополнительная инициализация ...
        return $foo;
    }
}

$container->set('Foo', ['app\helper\FooBuilder', 'build']);

$foo = $container->get('Foo');

```

Теперь тот, кто будет настраивать класс Foo, 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 также 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 экземпляры 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);  
$lister = new UserLister($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 Фаулера¹¹, 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.

¹¹<http://martinfowler.com/articles/injection.html>

Глава 6

Работа с базами данных

6.1 Объекты доступа к данным (DAO)

Построенные поверх PDO¹, Yii DAO (объекты доступа к данным) обеспечивают объектно-ориентированный API для доступа к реляционным базам данных. Это основа для других, более продвинутых, методов доступа к базам данных, включая [построитель запросов](#) и [active record](#).

При использовании Yii DAO вы в основном будете использовать чистый SQL и массивы PHP. Как результат, это самый эффективный способ доступа к базам данных. Тем не менее, так как синтаксис SQL может отличаться для разных баз данных, используя Yii DAO вам нужно будет приложить дополнительные усилия, чтобы сделать приложение не зависящим от конкретной базы данных.

Yii DAO из коробки поддерживает следующие базы данных:

- MySQL²
- MariaDB³
- SQLite⁴
- PostgreSQL⁵
- CUBRID⁶: версии 9.3 или выше.
- Oracle⁷
- MSSQL⁸: версии 2008 или выше.

Примечание: Новая версия pdo_oci для PHP 7 на данный момент существует только в форме исходного кода. Исполь-

¹<http://php.net/manual/ru/book.pdo.php>

²<http://www.mysql.com/>

³<https://mariadb.com/>

⁴<http://sqlite.org/>

⁵<http://www.postgresql.org/>

⁶<http://www.cubrid.org/>

⁷<http://www.oracle.com/us/products/database/overview/index.html>

⁸<https://www.microsoft.com/en-us/sqlserver/default.aspx>

зуйте инструкции сообщества по компиляции⁹.

6.1.1 Создание подключения к базе данных

Для доступа к базе данных, вы сначала должны подключиться к ней, создав экземпляр класса `yii\db\Connection`:

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

Так как подключение к БД часто нужно в нескольких местах, распространённой практикой является его настройка как компонента приложения:

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

Теперь вы можете получить доступ к подключению к БД с помощью выражения `Yii::$app->db`.

Подсказка: Вы можете настроить несколько компонентов подключения, если в вашем приложении используется несколько баз данных.

При настройке подключения, вы должны обязательно указывать Имя Источника Данных (DSN) через параметр `dsn`. Формат DSN отличается для разных баз данных. Дополнительное описание смотрите в справочнике PHP¹⁰. Ниже представлены несколько примеров:

- MySQL, MariaDB: `mysql:host=localhost;dbname=mydatabase`
- SQLite: `sqlite:/path/to/database/file`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID: `cubrid:dbname=demodb;host=localhost;port=33000`

⁹<https://github.com/yiisoft/yii2/issues/10975#issuecomment-248479268>

¹⁰<http://php.net/manual/ru/pdo.construct.php>

- MS SQL Server (via sqlsrv driver): `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (via dblib driver): `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (via mssql driver): `mssql:host=localhost;dbname=mydatabase`
- Oracle: `oci:dbname=//localhost:1521/mydatabase`

Заметьте, что если вы подключаетесь к базе данных через ODBC, вам необходимо указать свойство `yii\db\Connection::$driverName`, чтобы Yii знал какой тип базы данных используется. Например,

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

Кроме свойства `dsn`, вам необходимо указать `username` и `password`. Смотрите `yii\db\Connection` для того, чтоб посмотреть полный список свойств.

Информация: При создании экземпляра соединения к БД, фактическое соединение с базой данных будет установлено только при выполнении первого SQL запроса или при явном вызове метода `open()`.

Подсказка: Иногда может потребоваться выполнить некоторые запросы сразу после соединения с базой данных, для инициализации переменных окружения. Например, чтобы задать часовой пояс или кодировку. Сделать это можно зарегистрировав обработчик для события `afterOpen` в конфигурации приложения:

```
'db' => [
    // ...
    'on afterOpen' => function($event) {
        // $event->sender содержит соединение с базой данных
        $event->sender->createCommand("SET time_zone = 'UTC'")->
            execute();
    }
],
```

6.1.2 Выполнение SQL запросов

После создания экземпляра соединения, вы можете выполнить SQL запрос, выполнив следующие шаги:

1. Создать `yii\db\Command` из запроса SQL;

2. Привязать параметры (не обязательно);
3. Вызвать один из методов выполнения SQL из `yii\db\Command`.

Следующий пример показывает различные способы получения данных из базы данных:

```
// возвращает набор строк. каждая строка - это ассоциативный массив с
// именами столбцов и значений.
// если выборка ничего не вернёт, то будет получен пустой массив.
$post = Yii::$app->db->createCommand('SELECT * FROM post')
    ->queryAll();

// вернёт одну строку первую( строку)
// false, если ничего не будет выбрано
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=1')
    ->queryOne();

// вернёт один столбец первый( столбец)
// пустой массив, при отсутствии результата
$title = Yii::$app->db->createCommand('SELECT title FROM post')
    ->queryColumn();

// вернёт скалярное значение
// или false, при отсутствии результата
$count = Yii::$app->db->createCommand('SELECT COUNT(*) FROM post')
    ->queryScalar();
```

Примечание: Чтобы сохранить точность, данные извлекаются как строки, даже если тип поля в базе данных является числовым.

Привязка параметров

При создании команды из SQL запроса с параметрами, вы почти всегда должны использовать привязку параметров для предотвращения атак через SQL инъекции. Например,

```
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status')
    ->bindValue(':id', $_GET['id'])
    ->bindValue(':status', 1)
    ->queryOne();
```

В SQL запрос, вы можете встраивать один или несколько маркеров (например `:id` в примере выше). Маркеры должны быть строкой, начинающейся с двоеточия. Далее вам нужно вызвать один из следующих методов для привязки значений к параметрам:

- `bindValue()`: привязка одного параметра по значению
- `bindValues()`: привязка нескольких параметров в одном вызове

- `bindParam()`: похоже на `bindValue()`, но привязка происходит по ссылке.

Следующий пример показывает альтернативный путь привязки параметров:

```
$params = [':id' => $_GET['id'], ':status' => 1];

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status')
    ->bindValues($params)
    ->queryOne();

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND
    status=:status', $params)
    ->queryOne();
```

Привязка переменных реализована через подготавливаемые запросы¹¹. Помимо предотвращения атак путём SQL инъекций, это увеличивает производительность, так как запрос подготавливается один раз, а потом выполняется много раз с разными параметрами. Например,

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
// ...
```

Так как `bindParam()` поддерживает привязку параметров по ссылке, следующий код может быть написан следующим образом:

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id')
    ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();

$id = 2;
$post2 = $command->queryOne();
// ...
```

Обратите внимание что вы связываете маркер `$id` с переменной перед выполнением запроса, и затем меняете это значение перед каждым последующим выполнением (часто это делается в цикле). Выполнении запросов таким образом может быть значительно более эффективным, чем выполнение запроса для каждого значения параметра.

Выполнение **Не-SELECT** запросов

В методах `queryXyz()`, описанных в предыдущих разделах, вызываются **SELECT** запросы для извлечения данных из базы. Для запросов не воз-

¹¹<http://php.net/manual/ru/mysqli.quickstart.prepared-statements.php>

вращающих данные, вы должны использовать метод `yii\db\Command::execute()`. Например,

```
Yii::$app->db->createCommand('UPDATE post SET status=1 WHERE id=1')
->execute();
```

Метод `yii\db\Command::execute()` возвращает количество строк обработанных SQL запросом.

Для запросов INSERT, UPDATE и DELETE, вместо написания чистого SQL, вы можете вызвать методы `insert()`, `update()`, `delete()`, соответственно, для создания указанных SQL конструкций. Например,

```
// INSERT (table name, column values)
Yii::$app->db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
]);

// UPDATE (table name, column values, condition)
Yii::$app->db->createCommand()->update('user', ['status' => 1], 'age > 30')-
->execute();

// DELETE (table name, condition)
Yii::$app->db->createCommand()->delete('user', 'status = 0')->execute();
```

Вы можете также вызвать `batchInsert()` для вставки множества строк за один вызов. Это более эффективно чем вставлять записи по одной за раз:

```
// table name, column names, column values
Yii::$app->db->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
]);
```

Обратите внимание, что перечисленные методы лишь создают запрос. Чтобы его выполнить нужно вызывать `execute()`.

6.1.3 Экранирование имён таблиц и столбцов

При написании независимого от базы данных кода, правильно экранировать имена таблиц и столбцов довольно трудно, так как в разных базах данных правила экранирования разные. Чтобы преодолеть данную проблему вы можете использовать следующий синтаксис экранирования используемый в Yii:

- `[[column name]]`: заключайте имя столбца в двойные квадратные скобки;
- `{{table name}}`: заключайте имя таблицы в двойные фигурные скобки.

Yii DAO будет автоматически преобразовывать подобные конструкции в SQL в правильно экранированные имена таблиц и столбцов. Например,

```
// executes this SQL for MySQL: SELECT COUNT('id') FROM 'employee'
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{employee}}")
    ->queryScalar();
```

Использование префиксов таблиц

Если большинство ваших таблиц использует общий префикс в имени, вы можете использовать свойство Yii DAO для указания префикса.

Сначала, укажите префикс таблиц через свойство `yii\db\Connection::$tablePrefix`:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

Затем в коде, когда вам нужно сослаться на таблицу, имя которой содержит такой префикс, используйте синтаксис `{{%table name}}`. Символ процента будет автоматически заменён на префикс таблицы, который вы указали во время конфигурации соединения с базой данных. Например,

```
// для MySQL будет выполнен следующий SQL: SELECT COUNT('id') FROM 'tbl_employee'
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{%employee}}")
    ->queryScalar();
```

6.1.4 Исполнение транзакций

Когда вы выполняете несколько зависимых запросов последовательно, вам может потребоваться обернуть их в транзакцию для обеспечения целостности вашей базы данных. Если в любом из запросов произойдёт ошибка, база данных откатится на состояние, которое было до выполнения запросов.

Следующий код показывает типичное использование транзакций:

```
Yii::$app->db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...
});
```

Код выше эквивалентен приведённому ниже. Разница в том, что в данном случае мы получаем больше контроля над обработкой ошибок:

```
$db = Yii::$app->db;
$transaction = $db->beginTransaction();

try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...

    $transaction->commit();
} catch(\Exception $e) {

    $transaction->rollBack();

    throw $e;
}
```

При вызове метода `beginTransaction()`, будет запущена новая транзакция. Транзакция представлена объектом `yii\db\Transaction` сохранённым в переменной `$transaction`. Потом, запросы будут выполняться в блоке `try...catch...`. Если запросы будут выполнены удачно, будет выполнен метод `commit()`. Иначе, будет вызвано исключение, и будет вызван метод `rollBack()` для отката изменений сделанных до неудачно выполненного запроса внутри транзакции.

Указание уровня изоляции

Yii поддерживает настройку [уровня изоляции] для ваших транзакций. По умолчанию, при старте транзакции, будет использован уровень изоляции настроенный в вашей базе данных. Вы можете переопределить уровень изоляции по умолчанию, как указано ниже:

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

Yii::$app->db->transaction(function ($db) {
    ....
}, $isolationLevel);

// or alternatively

$transaction = Yii::$app->db->beginTransaction($isolationLevel);
```

Yii предоставляет четыре константы для наиболее распространённых уровней изоляции:

- `yii\db\Transaction::READ_UNCOMMITTED` - низший уровень, «Грязное» чтение, не повторяющееся чтение и фантомное чтение.
- `yii\db\Transaction::READ_COMMITTED` - предотвращает «Грязное» чтение.

- `yii\db\Transaction::REPEATABLE_READ` - предотвращает «Грязное» чтение и не повторяющееся чтение.
- `yii\db\Transaction::SERIALIZABLE` - высший уровень, предотвращает все вышеуказанные проблемы.

Помимо использования приведённых выше констант для задания уровня изоляции, вы можете, также, использовать строки поддерживаемые вашей СУБД. Например, в PostgreSQL, вы можете использовать `SERIALIZABLE READ ONLY DEFERRABLE`.

Заметьте что некоторые СУБД допускают настраивать уровень изоляции только для всего соединения. Следующие транзакции будут получать тот же уровень изоляции, даже если вы его не укажете. При использовании этой функции может потребоваться установить уровень изоляции для всех транзакций, чтоб избежать явно конфликтующих настроек. На момент написания этой статьи страдали от этого ограничения только MSSQL и SQLite.

Примечание: SQLite поддерживает только два уровня изоляции, таким образом вы можете использовать только `READ UNCOMMITTED` и `SERIALIZABLE`. Использование других уровней изоляции приведёт к генерации исключения.

Примечание: PostgreSQL не допускает установки уровня изоляции до старта транзакции, так что вы не сможете установить уровень изоляции прямо при старте транзакции. Вы можете использовать `yii\db\Transaction::setIsolationLevel()` в таком случае после старта транзакции.

Вложенные транзакции

Если ваша СУБД поддерживает Savepoint, вы можете вкладывать транзакции как показано ниже:

```
Yii::$app->db->transaction(function ($db) {  
    // outer transaction  
  
    $db->transaction(function ($db) {  
        // inner transaction  
    });  
});
```

Или так,

```
$db = Yii::$app->db;  
$outerTransaction = $db->beginTransaction();  
try {  
    $db->createCommand($sql1)->execute();  
  
    $innerTransaction = $db->beginTransaction();  
    try {
```

```

        $db->createCommand($sql2)->execute();
        $innerTransaction->commit();
    } catch (\Exception $e) {
        $innerTransaction->rollBack();
    }

    $outerTransaction->commit();
} catch (\Exception $e) {
    $outerTransaction->rollBack();
}

```

6.1.5 Репликация и разделение запросов на чтение и запись

Многие СУБД поддерживают репликацию баз данных¹² для лучшей доступности базы данных и уменьшения времени ответа сервера. С репликацией базы данных, данные копируются из *master servers* на *slave servers*. Все вставки и обновления должны происходить на основном сервере, хотя чтение может производиться и с подчинённых серверов.

Чтоб воспользоваться преимуществами репликации и достичь разделения чтения и записи, вам необходимо настроить компонент `yii\db\Connection` как указано ниже:

```

[
    'class' => 'yii\db\Connection',

    // настройки для мастера
    'dsn' => 'dsn for master server',
    'username' => 'master',
    'password' => '',

    // общие настройки для подчинённых
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // используем небольшой таймаут для соединения
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // список настроек для подчинённых серверов
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]

```

¹²[http://en.wikipedia.org/wiki/Replication_\(computing\)#Database_replication](http://en.wikipedia.org/wiki/Replication_(computing)#Database_replication)

Вышеуказанная конфигурация определяет систему с одним мастером и несколькими подчинёнными. Один из подчинённых будет подключен и использован для чтения, в то время как мастер будет использоваться для запросов записи. Такое разделение чтения и записи будет осуществлено автоматически с указанной конфигурацией. Например,

```
// создание экземпляра соединения, использующего вышеуказанную конфигурацию
Yii::$app->db = Yii::createObject($config);

// запрос к одному из подчинённых
$rows = Yii::$app->db->createCommand('SELECT * FROM user LIMIT 10')->
    queryAll();

// запрос к мастеру
Yii::$app->db->createCommand("UPDATE user SET username='demo' WHERE id=1")->
    execute();
```

Информация: Запросы выполненные через `yii\db\Command::execute()` определяются как запросы на запись, а все остальные запросы через один из “query” методов `yii\db\Command` воспринимаются как запросы на чтение. Вы можете получить текущий статус соединения к подчинённому серверу через `$db->slave`.

Компонент `Connection` поддерживает балансировку нагрузки и переключение при сбое для подчинённых серверов. При выполнении первого запроса на чтение, компонент `Connection` будет случайным образом выбирать подчинённый сервер и попытается подключиться к нему. Если сервер окажется “мёртвым”, он попытается подключиться к другому. Если ни один из подчинённых серверов не будет доступен, он подключится к мастеру. Если настроить **кеш статуса серверов**, то недоступность серверов может быть запомнена, чтоб не использоваться в течении **заданного промежутка времени**.

Информация: В конфигурации выше, таймаут соединения к подчинённому серверу настроен на 10 секунд. Это означает, что если сервер не ответит за 10 секунд, он будет считаться “мёртвым”. Вы можете отрегулировать этот параметр исходя из настроек вашей среды.

Вы также можете настроить несколько основных и несколько подчинённых серверов. Например,

```
[
    'class' => 'yii\db\Connection',

    // общая конфигурация для основных серверов
    'masterConfig' => [
```

```
'username' => 'master',
'password' => '',
'attributes' => [
    // используем небольшой таймаут для соединения
    PDO::ATTR_TIMEOUT => 10,
],
],

// список настроек для основных серверов
'masters' => [
    ['dsn' => 'dsn for master server 1'],
    ['dsn' => 'dsn for master server 2'],
],

// общие настройки для подчинённых
'slaveConfig' => [
    'username' => 'slave',
    'password' => '',
    'attributes' => [
        // используем небольшой таймаут для соединения
        PDO::ATTR_TIMEOUT => 10,
    ],
],

// список настроек для подчинённых серверов
'slaves' => [
    ['dsn' => 'dsn for slave server 1'],
    ['dsn' => 'dsn for slave server 2'],
    ['dsn' => 'dsn for slave server 3'],
    ['dsn' => 'dsn for slave server 4'],
],
]
```

Конфигурация выше, определяет два основных и четыре подчинённых серверов. Компонент `Connection` поддерживает балансировку нагрузки и переключение при сбое между основными серверами, также как и между подчинёнными. Различие заключается в том, что когда ни к одному из основных серверов не удастся подключиться будет выброшено исключение.

Примечание: Когда вы используете свойство `masters` для настройки одного или нескольких основных серверов, все остальные свойства для настройки соединения с базой данных (такие как `dsn`, `username`, `password`) будут проигнорированы компонентом `Connection`.

По умолчанию, транзакции используют соединение с основным сервером. И в рамках транзакции, все операции с БД будут использовать соединение с основным сервером. Например,

```
$db = Yii::$app->db;
// Транзакция запускается на основном сервере
```

```
$transaction = $db->beginTransaction();

try {
    // оба запроса выполняются на основном сервере
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE id=1")->
        execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

Если вы хотите запустить транзакцию на подчинённом сервере, вы должны указать это явно, как показано ниже:

```
$transaction = Yii::$app->db->slave->beginTransaction();
```

Иногда может потребоваться выполнить запрос на чтение через подключение к основному серверу. Это может быть достигнуто с использованием метода `useMaster()`:

```
$rows = Yii::$app->db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

Вы также можете явно установить `$db->enableSlaves` в ложь, чтоб направлять все запросы к соединению с мастером.

6.1.6 Работа со схемой базы данных

Yii DAO предоставляет целый набор методов для управления схемой базы данных, таких как создание новых таблиц, удаление столбцов из таблицы, и т.д.. Эти методы описаны ниже:

- `createTable()`: создание таблицы
- `renameTable()`: переименование таблицы
- `dropTable()`: удаление таблицы
- `truncateTable()`: удаление всех записей в таблице
- `addColumn()`: добавление столбца
- `renameColumn()`: переименование столбца
- `dropColumn()`: удаление столбца
- `alterColumn()`: преобразование столбца
- `addPrimaryKey()`: добавление первичного ключа
- `dropPrimaryKey()`: удаление первичного ключа
- `addForeignKey()`: добавление внешнего ключа
- `dropForeignKey()`: удаление внешнего ключа
- `createIndex()`: создания индекса
- `dropIndex()`: удаление индекса

Эти методы могут быть использованы, как указано ниже:

```
// CREATE TABLE
Yii::$app->db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

Вы также сможете получить описание схемы таблицы через вызов метода `getTableSchema()`. Например,

```
$table = Yii::$app->db->getTableSchema('post');
```

Метод вернёт объект `yii\db\TableSchema`, который содержит информацию о столбцах таблицы, первичных ключах, внешних ключах, и т.д.. Вся эта информация используется главным образом для **построителя запросов** и **active record**, чтоб помочь вам писать независимый от базы данных код.

6.2 Построитель запросов

Построенный поверх **DAO**, построитель запросов позволяет конструировать SQL выражения в программируемом и независимом от СУБД виде. В сравнении с написанием чистого SQL выражения, использование построителя помогает вам писать более читаемый связанный с SQL код и генерировать более безопасные SQL выражения.

Использование построителя запросов, как правило, включает два этапа:

1. Создание объекта `yii\db\Query` представляющего различные части (такие как `SELECT`, `FROM`) SQL выражения `SELECT`.
2. Выполнить запрос методом `yii\db\Query` (таким как `all()`) для извлечения данных из базы данных.

Следующий код показывает обычное использование построителя запросов:

```
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->all();
```

Приведённый выше код создаёт и выполняет следующее SQL выражение, где параметр `:last_name` привязывается к строке `'Smith'`.

```
SELECT 'id', 'email'
FROM 'user'
WHERE 'last_name' = :last_name
LIMIT 10
```

Информация: В основном вы будете работать с `yii\db\Query` вместо `yii\db\QueryBuilder`. Последний вызывается неявно при вызове одного из методов запроса. `yii\db\QueryBuilder` это класс, отвечающий за генерацию зависимого от СУБД SQL выражения (такие как экранирование имён таблиц/столбцов) из независимых от СУБД объектов `yii\db\Query`.

6.2.1 Построение запросов

Создав объект `yii\db\Query`, вы можете вызвать различные методы для создания различных частей SQL выражения. Имена методов напоминают ключевые слова SQL, используемые в соответствующих частях SQL запроса. Например, чтобы указать **FROM** часть запроса, вам нужно вызвать метод `from()`. Все методы построителя запросов возвращают свой объект, который позволяет объединять несколько вызовов в цепочку.

Далее будет описание каждого метода построителя запросов.

`select()`

Метод `select()` определяет фрагмент **SELECT** SQL запроса. Вы можете указать столбцы, которые должны быть выбраны, они должны быть указаны в виде массива или строки. Имена столбцов автоматически экранируются при создании SQL-запроса при его генерации из объекта `yii\db\Query`.

```
$query->select(['id', 'email']);  
  
// эквивалентно:  
  
$query->select('id, email');
```

Имена столбцов могут быть выбраны вместе с префиксами таблиц и/или алиасами столбцов, также как при записи обычного SQL выражения. Например,

```
$query->select(['user.id AS user_id', 'email']);  
  
// эквивалентно:  
  
$query->select('user.id AS user_id, email');
```

Если вы используете формат массива для указания столбцов, вы можете также указать ключи массива для указания алиасов столбцов. Например, приведённый выше код может быть переписан:

```
$query->select(['user_id' => 'user.id', 'email']);
```

Если вы не вызываете метод `select()` при создании запроса, будет использована `*`, что означает выбрать все столбцы.

Кроме имён столбцов, вы можете также использовать SQL выражения. Вы должны использовать формат массива для использования выражений, которые содержат запятые для предотвращения некорректного автоматического экранирования. Например,

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name", 'email'])
;
```

Начиная с версии 2.0.1, вы также можете использовать подзапросы. Вы должны указывать каждый подзапрос в выражении как объект `yii\db\Query`. Например,

```
$subQuery = (new Query())->select('COUNT(*)')->from('user');

// SELECT 'id', (SELECT COUNT(*) FROM 'user') AS 'count' FROM 'post'
$query = (new Query())->select(['id', 'count' => $subQuery])->from('post');
```

Чтоб выбрать конкретные строки, вы можете вызвать метод `distinct()`:

```
// SELECT DISTINCT 'user_id' ...
$query->select('user_id')->distinct();
```

Вы можете вызвать `addSelect()` для добавления полей. Например,

```
$query->select(['id', 'username'])
->addSelect(['email']);
```

`from()`

Метод `from()` указывает фрагмент `FROM` SQL запроса. Например,

```
// SELECT * FROM 'user'
$query->from('user');
```

Вы можете указать имена таблиц в виде строки или массива. Имена таблиц могут содержать префикс схемы и/или алиасы таблиц, как при написании обычного SQL выражения. Например,

```
$query->from(['public.user u', 'public.post p']);

// эквивалентно:

$query->from('public.user u, public.post p');
```

Если вы используете формат массива, вы можете использовать ключи массива для указания алиасов:

```
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

Кроме имён таблиц, вы можете, также, как и в `select`, указывать подзапросы в виде объекта `yii\db\Query`.

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');

// SELECT * FROM (SELECT 'id' FROM 'user' WHERE status=1) u
$query->from(['u' => $subQuery]);
```


`where()`

Метод `where()` определяет фрагмент `WHERE` SQL выражения. Вы можете использовать один из трёх форматов:

- строковый формат, Например, `'status=1'`
- формат массива, Например, `['status' => 1, 'type' => 2]`
- формат операторов, Например, `['like', 'name', 'test']`

Строковый формат Строковый формат - это лучший выбор для простых условий. Он работает так, будто вы просто пишете SQL запрос. Например,

```
$query->where('status=1');

// или используя привязку параметров
$query->where('status=:status', [':status' => $status]);
```

Не встраивайте переменные непосредственно в условие, особенно если значение переменной получено от пользователя, потому что это делает ваше приложение подверженным атакам через SQL инъекции.

```
// Опасность! Не делайте так если вы не уверены, что $status это
// действительно число.
$query->where("status=$status");
```

При использовании привязки параметров, вы можете вызывать `params()` или `addParams()` для отдельного указания параметров.

```
$query->where('status=:status')
->addParams([':status' => $status]);
```

Формат массива Формат массива лучше всего использовать для указания нескольких объединяемых через `AND` условий, каждое из которых является простым равенством. Он описывается в виде массива, ключами которого являются имена столбцов, а значения соответствуют значениям столбцов.

```
// ...WHERE ('status' = 10) AND ('type' IS NULL) AND ('id' IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

Как вы можете видеть, построитель запросов достаточно умен, чтобы правильно обрабатывать значения `null` или массивов.

Вы также можете использовать подзапросы:

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE 'id' IN (SELECT 'id' FROM 'user')
$query->where(['id' => $userQuery]);
```

Формат операторов Формат оператора позволяет задавать произвольные условия в программном стиле. Он имеет следующий вид:

```
[operator, operand1, operand2, ...]
```

Операнды могут быть заданы в виде строкового формата, формата массива или формата операторов рекурсивно, в то время как оператор может быть одним из следующих:

- **and**: операнды должны быть объединены с помощью оператора AND. Например, `['and', 'id=1', 'id=2']` сгенерирует `id=1 AND id=2`. Если операнд массив, он будет сконвертирован в строку по правилам описанным ниже. Например, `['and', 'type=1', ['or', 'id=1', 'id=2']]` сгенерирует `type=1 AND (id=1 OR id=2)`. Этот метод не производит никакого экранирования.
- **or**: похож на оператор **and** за исключением того, что будет использоваться оператор OR.
- **between**: первый операнд должен быть именем столбца, а второй и третий оператор должны быть начальным и конечным значением диапазона. Например, `['between', 'id', 1, 10]` сгенерирует `id BETWEEN 1 AND 10`.
- **not between**: похож на **between** за исключением того, что BETWEEN заменяется на NOT BETWEEN в сгенерированном условии.
- **in**: первый операнд должен быть столбцом или выражением БД. Второй операнд может быть либо массивом, либо объектом `Query`. Будет сгенерировано условие IN. Если второй операнд массив, он будет представлять набор значений, которым может быть равен столбец или выражение БД; Если второй операнд объект `Query`, будет сформирован подзапрос, который будет использован как диапазон для столбца или выражения БД. Например, `['in', 'id', [1, 2, 3]]` сформирует `id IN (1, 2, 3)`. Метод будет правильно экранировать имя столбца и значения диапазона. Оператор **in** также поддерживает составные столбцы. В этом случае, первый операнд должен быть массивом названий столбцов, в то время как операнд 2 должен быть массивом массивов или объектом `Query` представляющим диапазоны для столбцов.
- **not in**: похож на оператор **in**, кроме того что IN будет заменён на NOT IN в сформированном условии.
- **like**: первый операнд должен быть столбцом или выражением БД, а второй операнд будет строкой или массивом представляющим значения, на которые должны быть похожи столбцы или выражения БД. Например, `['like', 'name', 'tester']` сформирует `name LIKE 'tester'`. Когда диапазон значений задан в виде массива, несколько LIKE утверждений будут сформированы и соединены с помощью AND. Например, `['like', 'name', ['test', 'sample']]` сформирует `name LIKE 'test%' AND name LIKE 'sample%'`. Вы также може-

те передать третий необязательный операнд, для указания способа экранирования специальных символов в значениях. Операнд должен быть представлен массивом соответствия специальных символов их экранированным аналогам. Если этот операнд не задан, то будет использовано соответствие по умолчанию. Вы можете также использовать значение `false` или пустой массив, чтоб указать что значения уже экранированы. Обратите внимание, что при использовании массива соответствия экранирования (или если третий операнд не передан), значения будут автоматически заключены в символы процентов.

Примечание: При использовании PostgreSQL вы можете использовать также `ilike`¹³ вместо `like` для регистронезависимого поиска.

- `or like`: похож на оператор `like`, только утверждения `LIKE` будут объединяться с помощью оператора `OR`, если второй операнд будет представлен массивом.
- `not like`: похож на оператор `like`, только `LIKE` будет заменён на `NOT LIKE` в сгенерированном условии.
- `or not like`: похож на оператор `not like`, только утверждения `NOT LIKE` будут объединены с помощью `OR`.
- `exists`: требует один операнд, который должен быть экземпляром `yii\db\Query` представляющим подзапрос. Будет сгенерировано выражение `EXISTS (sub-query)`.
- `not exists`: похож на оператор `exists` и сформирует выражение `NOT EXISTS (sub-query)`.
- `>`, `<=`, или другие валидные операторы БД, которые требуют двух операндов: первый операнд должен быть именем столбца, второй операнд это значение. Например, `['>', 'age', 10]` сформирует `age > 10`.

Добавление условий Вы можете использовать `andWhere()` или `orWhere()` для добавления дополнительных условий. Вы можете использовать эти вызовы несколько раз для добавления нескольких условий. Например,

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

Если `$search` не пустое, то будет сформировано следующее условие `WHERE`:

¹³<http://www.postgresql.org/docs/8.3/static/functions-matching.html#FUNCTIONS-LIKE>

```
WHERE ('status' = 10) AND ('title' LIKE '%yii%')
```

Условия для фильтров Когда условие `WHERE` формируется на основе пользовательского ввода, обычно, хочется проигнорировать не заданные значения. Например, в форме поиска, которая позволяет осуществлять поиск по имени пользователя или email, вы хотели бы игнорировать `username/email` условие, если пользователь ничего не ввёл в поле ввода. Вы можете достичь этого используя метод `filterWhere()`.

```
// $username и $email вводит пользователь
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

Единственное отличие между `filterWhere()` и `where()` заключается в игнорировании пустых значений, переданных в условие в форме массива. Таким образом если `$email` будет пустым, а `$username` нет, то приведённый выше код сформирует условие `WHERE username=:username`.

Информация: значение признаётся пустым, если это `null`, пустой массив, пустая строка или строка состоящая из одних пробельных символов.

Также вместо `andWhere()` и `orWhere()`, вы можете использовать `andFilterWhere()` и `orWhereWhere()` для добавления дополнительных условий фильтрации.

`orderBy()`

Метод <code>orderBy()</code> определяет фрагмент <code>ORDER BY</code> SQL выражения. Например,
<code>'php</code>
<code>// ... ORDER BY id ASC, name DESC</code>
<code>\$query->orderBy([</code>
<code>'id' => SORT_ASC,</code>
<code>'name' => SORT_DESC,</code>
<code>]);</code>
<code>'</code>

В данном коде, ключи массива - это имена столбцов, а значения массива - это соответствующее направление сортировки. PHP константа `SORT_ASC` определяет сортировку по возрастанию и `SORT_DESC` сортировка по умолчанию.

Если `ORDER BY` содержит только простые имена столбцов, вы можете определить их с помощью столбцов, также как и при написании обычного SQL. Например,

```
$query->orderBy('id ASC, name DESC');
```

Примечание: Вы должны использовать массив для указания ORDER BY содержащих выражения БД.

Вы можете вызывать `addOrderBy()` для добавления столбцов в фрагмент ORDER BY.

```
$query->orderBy('id ASC')
    ->addOrderBy('name DESC');
```

groupBy()

Метод `groupBy()` определяет фрагмент GROUP BY SQL запроса.

```
// ... GROUP BY 'id', 'status'
$query->groupBy(['id', 'status']);
```

Если фрагмент GROUP BY содержит только простые имена столбцов, вы можете указать их используя строку, также как в обычном SQL выражении.

```
$query->groupBy('id, status');
```

Примечание: Вы должны использовать массив для указания GROUP BY содержащих выражения БД.

Вы можете вызывать `addGroupBy()` для добавления имён столбцов в фрагмент GROUP BY. For example,

```
$query->groupBy(['id', 'status'])
    ->addGroupBy('age');
```

having()

Метод `having()` определяет фрагмент HAVING SQL запроса. Он принимает условия, которое может быть определено тем же способом, что и для `where()`.

```
// ... HAVING 'status' = 1
$query->having(['status' => 1]);
```

Пожалуйста, обратитесь к документации для `where()` для более подробной информации о определении условий.

Вы можете вызывать `andHaving()` или `orHaving()` для добавления дополнительных условий в фрагмент HAVING.

```
// ... HAVING ('status' = 1) AND ('age' > 30)
$query->having(['status' => 1])
    ->andHaving(['>', 'age', 30]);
```

limit() и offset()

Методы `limit()` и `offset()` определяют фрагменты `LIMIT` и `OFFSET` SQL запроса.

```
// ... LIMIT 10 OFFSET 20
$query->limit(10)->offset(20);
```

Если вы определяете неправильный `limit` или `offset` (например отрицательное значение), они будут проигнорированы.

Информация: Для СУБД, которые не поддерживают `LIMIT` и `OFFSET` (такие как MSSQL), построитель запросов будет генерировать SQL выражения, которые эмулируют поведение `LIMIT/OFFSET`.

join()

Метод <code>join()</code> определяет фрагмент <code>JOIN</code> SQL запроса.
<code>'php</code>
<code>// ... LEFT JOIN post ON post.user_id = user.id</code>
<code>\$query->join('LEFT JOIN', 'post', 'post.user_id = user.id');</code>
<code>'</code>
Метод <code>join()</code> принимает четыре параметра:
- <code>\$type</code> : тип объединения, например, <code>'INNER JOIN'</code> , <code>'LEFT JOIN'</code> .
- <code>\$table</code> : имя таблицы, которая должна быть присоединена.
- <code>\$on</code> : необязательное условие объединения, то есть фрагмент <code>ON</code> . Пожалуйста, обратитесь к документации для <code>where()</code> для более подробной информации о определении условий. Отметим, что синтаксис массивов не работает
для задания условий для столбцов, то есть <code>['user.id' => 'comment.userId']</code> будет означать условие, где ID пользователя
должен быть равен строке <code>'comment.userId'</code> . Вместо этого стоит указывать условие в виде строки <code>'user.id = comment.userId'</code> .
- <code>\$params</code> : необязательные параметры присоединяемые к условию объединения.

Вы можете использовать следующие сокращающие методы для указания `INNER JOIN`, `LEFT JOIN` и `RIGHT JOIN`, в указанном порядке.

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

Например,

```
$query->leftJoin('post', 'post.user_id = user.id');
```

Для соединения с несколькими таблицами, вызовите вышеуказанные методы несколько раз.

Кроме соединения с таблицами, вы можете также присоединять подзапросы. Чтобы это сделать, укажите объединяемый подзапрос как объект `yii\db\Query`.

```
$subQuery = (new \yii\db\Query())->from('post');
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

В этом случае, вы должны передать подзапросы в массиве и использовать ключи для определения алиасов.

`union()`

Метод `union()` определяет фрагмент UNION SQL запроса.

```
$query1 = (new \yii\db\Query())
    ->select("id, category_id AS type, name")
    ->from('post')
    ->limit(10);

$query2 = (new \yii\db\Query())
    ->select('id, type, name')
    ->from('user')
    ->limit(10);

$query1->union($query2);
```

Вы можете вызвать `union()` несколько раз для присоединения фрагментов UNION.

6.2.2 Методы выборки

`yii\db\Query` предоставляет целый набор методов для разных вариантов выборки:

- `all()`: возвращает массив строк, каждая из которых это ассоциативный массив пар ключ-значение.
- `one()`: возвращает первую строку запроса.
- `column()`: возвращает первый столбец результата.
- `scalar()`: возвращает скалярное значение первого столбца первой строки результата.
- `exists()`: возвращает значение указывающее, что выборка содержит результат.
- `count()`: возвращает результат COUNT запроса.
- Другие методы агрегирования запросов, включая `sum($q)`, `average($q)`, `max($q)`, `min($q)`. Параметр `$q` обязателен для этих методов и могут содержать либо имя столбца, либо выражение БД.

Например,

```
// SELECT 'id', 'email' FROM 'user'
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
```

```
->from('user')
->all();

// SELECT * FROM 'user' WHERE 'username' LIKE '%test%'
$row = (new \yii\db\Query())
->from('user')
->where(['like', 'username', 'test'])
->one();
```

Примечание: метод `one()` вернёт только первую строку результата запроса. Он НЕ добавляет `LIMIT 1` в генерируемый SQL. Это хорошо и предпочтительно если вы знаете, что запрос вернёт только одну или несколько строк данных (например, при запросе по первичному ключу). Однако, если запрос потенциально может вернуть много строк данных, вы должны вызвать `limit(1)` для повышения производительности. Например, `(new \yii\db\Query())->from('user')->limit(1)->one()`.

Все методы выборки могут получать необязательный параметр `$db`, представляющий **соединение** с БД, которое должно использоваться, чтобы выполнить запрос к БД. Если вы пропускаете этот параметр, будет использоваться **компонент приложения** `$db`. Ниже приведён ещё один пример использования метода `count()`:

```
// executes SQL: SELECT COUNT(*) FROM 'user' WHERE 'last_name'=:last_name
$count = (new \yii\db\Query())
->from('user')
->where(['last_name' => 'Smith'])
->count();
```

При вызове методов выборки `yii\db\Query`, внутри на самом деле проводится следующая работа:

- Вызывается `yii\db\QueryBuilder` для генерации SQL запроса на основе текущего `yii\db\Query`;
- Создаёт объект `yii\db\Command` с сгенерированным SQL запросом;
- Вызывается выбирающий метод (например `queryAll()`) из `yii\db\Command` для выполнения SQL запроса и извлечения данных.

Иногда вы можете захотеть увидеть или использовать SQL запрос построенный из объекта `yii\db\Query`. Этой цели можно добиться с помощью следующего кода:

```
$command = (new \yii\db\Query())
->select(['id', 'email'])
->from('user')
->where(['last_name' => 'Smith'])
->limit(10)
->createCommand();

// показать SQL запрос
```



```
echo $command->sql;
// показать привязываемые параметры
print_r($command->params);

// возвращает все строки запроса
$rows = $command->queryAll();
```

Индексация результатов запроса

При вызове `all()` возвращается массив строк индексированный последовательными целыми числами. Иногда вам может потребоваться индексировать его по-другому, например, сделать индекс по указанному столбцу или значением выражения. Вы можете реализовать такое поведение через вызов `indexBy()` перед вызовом `all()`.

```
// возвращает [100 => ['id' => 100, 'username' => '...', ...], 101 => [...],
               103 => [...], ...]
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

Для индексации по значению выражения, передайте анонимную функцию в метод `indexBy()`:

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

Анонимная функция должна принимать параметр `$row`, который содержит текущую строку запроса и должна вернуть скалярное значение, которое будет использоваться как значение индекса для текущей строки.

Пакетная выборка

При работе с большими объемами данных, методы наподобие `yii\db\Query::all()` не подходят, потому что они требуют загрузки всех данных в память. Чтобы сохранить требования к памяти минимальными, Yii предоставляет поддержку так называемых пакетных выборок. Пакетная выборка делает возможным курсоры данных и выборку данных пакетами.

Пакетная выборка может использоваться следующим образом:

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
```

```
->orderBy('id');

foreach ($query->batch() as $users) {
    // $users это массив из 100 или менее строк из таблицы пользователей
}

// или если вы хотите перебрать все строки по одной
foreach ($query->each() as $user) {
    // $user представляет одну строку из выборки
}
```

Метод `yii\db\Query::batch()` и `yii\db\Query::each()` возвращает объект `yii\db\BatchQueryResult`, который реализует интерфейс `Iterator` и может использоваться в конструкции `foreach`. Во время первой итерации будет выполнен SQL запрос к базе данных. Данные будут выбираться пакетами в следующих итерациях. По умолчанию, размер пакета имеет размер 100, то есть при каждой выборке будет выбираться по 100 строк. Вы можете изменить размер пакета, передав первый параметр в метод `batch()` или `each()`.

По сравнению с `yii\db\Query::all()`, пакетная выборка загружает только по 100 строк данных за раз в память. Если вы обрабатываете данные и затем сразу выбрасываете их, пакетная выборка может помочь уменьшить использование памяти.

Если указать индексный столбец через `yii\db\Query::indexBy()`, в пакетной выборке индекс будет сохраняться. Например,

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users индексируется по столбцу "username"
}

foreach ($query->each() as $username => $user) {
    // ...
}
```

6.3 Active Record

Active Record¹⁴ обеспечивает объектно-ориентированный интерфейс для доступа и манипулирования данными, хранящимися в базах данных. Класс `Active Record` соответствует таблице в базе данных, объект `Active Record` соответствует строке этой таблицы, а *атрибут* объекта `Active Record` представляет собой значение отдельного столбца строки. Вместо непосредственного написания SQL-выражений вы сможете получать доступ к атрибутам `Active Record` и вызывать методы `Active Record` для

¹⁴<http://ru.wikipedia.org/wiki/ActiveRecord>

доступа и манипулирования данными, хранящимися в таблицах базы данных.

Для примера предположим, что `Customer` - это класс Active Record, который сопоставлен с таблицей `customer`, а `name` - столбец в таблице `customer`. Тогда вы можете написать следующий код для вставки новой строки в таблицу `customer`:

```
$customer = new Customer();  
$customer->name = 'Qiang';  
$customer->save();
```

Вышеприведённый код аналогичен использованию следующего SQL-выражения в MySQL, которое менее интуитивно, потенциально может вызывать ошибки и даже проблемы совместимости, если вы используете различные виды баз данных:

```
$db->createCommand('INSERT INTO 'customer' ('name') VALUES (:name)', [  
    ':name' => 'Qiang',  
)->execute();
```

Yii поддерживает работу с Active Record для следующих реляционных баз данных:

- MySQL 4.1 и выше: посредством `yii\db\ActiveRecord`
- PostgreSQL 7.3 и выше: посредством `yii\db\ActiveRecord`
- SQLite 2 и 3: посредством `yii\db\ActiveRecord`
- Microsoft SQL Server 2008 и выше: посредством `yii\db\ActiveRecord`
- Oracle: посредством `yii\db\ActiveRecord`
- CUBRID 9.3 и выше: посредством `yii\db\ActiveRecord` (Имейте ввиду, что вследствие бага¹⁵ в PDO-расширении для CUBRID, заключение значений в кавычки не работает, поэтому необходимо использовать CUBRID версии 9.3 как на клиентской стороне, так и на сервере)
- Sphinx: посредством `yii\sphinx\ActiveRecord`, потребуется расширение `yii2-sphinx`
- Elasticsearch: посредством `yii\elasticsearch\ActiveRecord`, потребуется расширение `yii2-elasticsearch`

Кроме того Yii поддерживает использование Active Record со следующими NoSQL базами данных:

- Redis 2.6.12 и выше: посредством `yii\redis\ActiveRecord`, потребуется расширение `yii2-redis`
- MongoDB 1.3.0 и выше: посредством `yii\mongodb\ActiveRecord`, потребуется расширение `yii2-mongodb`

В этом руководстве мы в основном будем описывать использование Active Record для реляционных баз данных. Однако большая часть этого материала также применима при использовании Active Record с NoSQL базами данных.

¹⁵<http://jira.cubrid.org/browse/API5-658>

6.3.1 Объявление классов Active Record

Для начала объявите свой собственный класс, унаследовав класс `yii\db\ActiveRecord`. Поскольку каждый класс Active Record сопоставлен с таблицей в базе данных, в своём классе вы должны переопределить метод `tableName()`, чтобы указать с какой именно таблицей связан ваш класс.

В нижеследующем примере мы объявляем класс Active Record с названием `Customer` для таблицы `customer`.

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string название таблицы, сопоставленной с этим ActiveRecord-
     * классом.
     */
    public static function tableName()
    {
        return 'customer';
    }
}
```

Объекты Active Record являются *моделями*. Именно поэтому мы обычно задаём классам Active Record пространство имён `app\models` (или другое пространство имён, предназначенное для моделей).

Т.к. класс `yii\db\ActiveRecord` наследует класс `yii\base\Model`, он обладает *всеми* возможностями *моделей*, такими как атрибуты, правила валидации, способы сериализации данных и т.д.

6.3.2 Подключение к базам данных

По умолчанию Active Record для доступа и манипулирования данными БД использует *компонент приложения* `db` в качестве компонента `DB connection`. Как сказано в разделе *Объекты доступа к данным (DAO)*, вы можете настраивать компонент `db` на уровне конфигурации приложения как показано ниже:

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
```

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

Если вы хотите использовать для подключения к базе данных другой компонент подключения, отличный от `db`, вам нужно переопределить метод `getDb()`:

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // использовать компонент приложения "db2"
        return \Yii::$app->db2;
    }
}
```

6.3.3 Получение данных

После объявления класса `Active Record` вы можете использовать его для получения данных из соответствующей таблицы базы данных. Этот процесс, как правило, состоит из следующих трёх шагов:

1. Создать новый объект запроса вызовом метода `yii\db\ActiveRecord::find()`;
2. Настроить объект запроса вызовом [методов построения запросов](#);
3. Вызвать один из [методов получения данных](#) для извлечения данных в виде объектов `Active Record`.

Как вы могли заметить, эти шаги очень похожи на работу с [построителем запросов](#). Различие лишь в том, что для создания объекта запроса вместо оператора `new` используется метод `yii\db\ActiveRecord::find()`, возвращающий новый объект запроса, являющийся представителем класса `yii\db\ActiveQuery`.

Ниже приведено несколько примеров использования `Active Query` для получения данных:

```
// возвращает покупателя с идентификатором 123
// SELECT * FROM 'customer' WHERE 'id' = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// возвращает всех активных покупателей, сортируя их по идентификаторам
// SELECT * FROM 'customer' WHERE 'status' = 1 ORDER BY 'id'
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
```

```
->orderBy('id')
->all();

// возвращает количество активных покупателей
// SELECT COUNT(*) FROM 'customer' WHERE 'status' = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// возвращает всех покупателей массивом, индексированным их идентификаторами
// SELECT * FROM 'customer'
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

В примерах выше `$customer` - это объект класса `Customer`, в то время как `$customers` - это массив таких объектов. Все эти объекты заполнены данными таблицы `customer`.

Информация: Т.к. класс `yii\db\ActiveQuery` наследует `yii\db\Query`, вы можете использовать в нём *все* методы построения запросов и все методы класса `Query` как описано в разделе [Построитель запросов](#).

Т.к. извлечение данных по первичному ключу или значениям отдельных столбцов достаточно распространённая задача, Yii предоставляет два коротких метода для её решения:

- `yii\db\ActiveRecord::findOne()`: возвращает один объект `ActiveRecord`, заполненный первой строкой результата запроса.
- `yii\db\ActiveRecord::findAll()`: возвращает массив объектов `ActiveRecord`, заполненных *всеми* полученными результатами запроса.

Оба метода могут принимать параметры в одном из следующих форматов:

- скалярное значение: значение интерпретируется как первичный ключ, по которому следует искать. Yii прочитает информацию о структуре базы данных и автоматически определит, какой столбец таблицы содержит первичные ключи.
- массив скалярных значений: массив интерпретируется как набор первичных ключей, по которым следует искать.
- ассоциативный массив: ключи массива интерпретируются как названия столбцов, а значения - как содержимое столбцов, которое следует искать. За подробностями вы можете обратиться к разделу [Hash Format](#)

Нижеследующий код демонстрирует, каким образом эти методы могут быть использованы:

```
// возвращает покупателя с идентификатором 123
// SELECT * FROM 'customer' WHERE 'id' = 123
```

```

$customer = Customer::findOne(123);

// возвращает покупателей с идентификаторами 100, 101, 123 и 124
// SELECT * FROM 'customer' WHERE 'id' IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// возвращает активного покупателя с идентификатором 123
// SELECT * FROM 'customer' WHERE 'id' = 123 AND 'status' = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// возвращает всех неактивных покупателей
// SELECT * FROM 'customer' WHERE 'status' = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);

```

Примечание: Ни метод `yii\db\ActiveRecord::findOne()`, ни `yii\db\ActiveQuery::one()` не добавляет условие `LIMIT 1` к генерируемым SQL-запросам. Если ваш запрос может вернуть много строк данных, вы должны вызвать метод `limit(1)` явно в целях улучшения производительности, например: `Customer::find()->limit(1)->one()`.

Помимо использования методов построения запросов вы можете также писать запросы на “чистом” SQL для получения данных и заполнения ими объектов Active Record. Вы можете делать это посредством метода `yii\db\ActiveRecord::findBySql()`:

```

// возвращает всех неактивных покупателей
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, ['status' => Customer::
    STATUS_INACTIVE])->all();

```

Не используйте дополнительные методы построения запросов после вызова метода `findBySql()`, т.к. они будут проигнорированы.

6.3.4 Доступ к данным

Как сказано выше, получаемые из базы данные заполняют объекты Active Record и каждая строка результата запроса соответствует одному объекту Active Record. Вы можете получить доступ к значениям столбцов с помощью атрибутов этих объектов. Например так:

```

// "id" и "email" - названия столбцов в таблице "customer"
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;

```

Примечание: Атрибуты объекта Active Record названы в соответствии с названиями столбцов связанной таблицы с учётом регистра. Yii автоматически объявляет для каждого столбца связанной таблицы атрибут в Active Record. Вы НЕ должны переопределять какие-либо из этих атрибутов.

Атрибуты Active Record названы в соответствии с именами столбцов таблицы. Если столбцы вашей таблицы именуются через нижнее подчёркивание, то может оказаться, что вам придётся писать РНР-код вроде этого: `$customer->first_name` - в нём будет использоваться нижнее подчёркивание для разделения слов в названиях атрибутов. Если вы обеспокоены единообразием стиля кодирования, вам придётся переименовать столбцы вашей таблицы соответствующим образом (например, назвать столбцы в стиле camelCase).

Преобразование данных

Часто бывает так, что данные вводятся и/или отображаются в формате, который отличается от формата их хранения в базе данных. Например, в базе данных вы храните дни рождения покупателей в формате UNIX timestamp (что, кстати говоря, не является хорошим дизайном), в то время как во многих случаях вы хотите манипулировать днями рождения в виде строк формата `'ддммгггг..'`. Для достижения этой цели, вы можете объявить методы *преобразования данных* в ActiveRecord-классе `Customer` как показано ниже:

```
class Customer extends ActiveRecord
{
    // ...

    public function getBirthdayText()
    {
        return date('d.m.Y', $this->birthday);
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

Теперь в своём РНР коде вместо доступа к `$customer->birthday`, вы сможете получить доступ к `$customer->birthdayText`, что позволит вам вводить и отображать дни рождения покупателей в формате `'ддммгггг..'`.

Подсказка: Вышеприведённый пример демонстрирует общий способ преобразования данных в различные форматы.

Если вы работаете с датами и временем, вы можете использовать `DateValidator` и `yii\jui\DatePicker`, которые проще в использовании и являются более мощными инструментами.

Получение данных в виде массива

Несмотря на то, что получение данных в виде Active Record объектов является удобным и гибким, этот способ не всегда подходит при получении большого количества данных из-за больших накладных расходов памяти. В этом случае вы можете получить данные в виде PHP-массива, используя перед выполнением запроса метод `asArray()`:

```
// возвращает всех покупателей
// каждый покупатель будет представлен в виде ассоциативного массива
$customers = Customer::find()
    ->asArray()
    ->all();
```

Примечание: В то время как этот способ бережёт память и улучшает производительность, он ближе к низкому слою абстракции базы данных и вы потеряете многие возможности Active Record. Важное отличие заключается в типах данных значений столбцов. Когда вы получаете данные в виде объектов Active Record, значения столбцов автоматически приводятся к типам, соответствующим типам столбцов; с другой стороны, когда вы получаете данные в массивах, значения столбцов будут строковыми (до тех пор, пока они являются результатом работы PDO-слоя без какой-либо обработки), несмотря на настоящие типы данных соответствующих столбцов.

Пакетное получение данных

В главе [Построитель запросов](#) мы объясняли, что вы можете использовать *пакетную выборку* для снижения расходов памяти при получении большого количества данных из базы. Вы можете использовать такой же подход при работе с Active Record. Например:

```
// получить 10 покупателей одновременно
foreach (Customer::find()->batch(10) as $customers) {
    // $customers - это массив, в котором находится 10 или меньше объектов
    // класса Customer
}

// получить одновременно десять покупателей и перебрать их одного за другим
foreach (Customer::find()->each(10) as $customer) {
    // $customer - это объект класса Customer
}
```

```
// пакетная выборка с жадной загрузкой
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer - это объекта класса Customer
}
```

6.3.5 Сохранение данных

Используя Active Record, вы легко можете сохранить данные в базу данных, осуществив следующие шаги:

1. Подготовьте объект Active Record;
2. Присвойте новые значения атрибутам Active Record;
3. Вызовите метод `yii\db\ActiveRecord::save()` для сохранения данных в базу данных.

Например:

```
// вставить новую строку данных
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// обновить имеющуюся строку данных
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

Метод `save()` может вставить или обновить строку данных в зависимости от состояния Active Record объекта. Если объект создан с помощью оператора `new`, вызов метода `save()` приведёт к вставке новой строки данных; если же объект был получен с помощью запроса на получение данных, вызов `save()` обновит строку таблицы, соответствующую объекту Active Record.

Вы можете различать два состояния Active Record объекта с помощью проверки значения его свойства `isNewRecord`. Это свойство также используется внутри метода `save()` как показано ниже:

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

Подсказка: Вы можете вызвать `insert()` или `update()` непосредственно, чтобы вставить или обновить строку данных в таблице.

Валидация данных

Т.к. класс `yii\db\ActiveRecord` наследует класс `yii\base\Model`, он обладает такими же возможностями [валидации данных](#). Вы можете объявить правила валидации переопределив метод `rules()` и осуществлять валидацию данных посредством вызовов метода `validate()`.

Когда вы вызываете метод `save()`, по умолчанию он автоматически вызывает метод `validate()`. Только после успешного прохождения валидации происходит сохранение данных; в ином случае метод `save()` просто возвращает `false`, и вы можете проверить свойство `errors` для получения сообщений об ошибках валидации.

Подсказка: Если вы уверены, что ваши данные не требуют валидации (например, данные пришли из доверенного источника), вы можете вызвать `save(false)`, чтобы пропустить валидацию.

Массовое присваивание

Как и обычные [модели](#), объекты Active Record тоже обладают [возможностью массового присваивания](#). Как будет показано ниже, используя эту возможность, вы можете одним PHP выражением присвоить значения множества атрибутов Active Record объекту. Запомните однако, что только [безопасные атрибуты](#) могут быть массово присвоены.

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

Обновление счётчиков

Распространённой задачей является инкремент или декремент столбца в таблице базы данных. Назовём такие столбцы столбцами-счётчиками. Вы можете использовать метод `updateCounters()` для обновления одного или нескольких столбцов-счётчиков. Например:

```
$post = Post::findOne(100);
```

```
// UPDATE 'post' SET 'view_count' = 'view_count' + 1 WHERE 'id' = 100  
$post->updateCounters(['view_count' => 1]);
```

Примечание: Если вы используете метод `yii\db\ActiveRecord::save()` для обновления столбца-счётчика, вы можете прийти к некорректному результату, т.к. вполне вероятно, что этот же счётчик был сохранён сразу несколькими запросами, которые читают и записывают этот же столбец-счётчик.

Dirty-атрибуты

Когда вы вызываете `save()` для сохранения Active Record объекта, сохраняются только *dirty-атрибуты*. Атрибут считается *dirty-атрибутом*, если его значение было изменено после чтения из базы данных или же он был сохранён в базу данных совсем недавно. Заметьте, что валидация данных осуществляется независимо от того, имеются ли dirty-атрибуты в объекте Active Record или нет.

Active Record автоматически поддерживает список dirty-атрибутов. Это достигается за счёт хранения старых значений атрибутов и сравнения их с новыми. Вы можете вызвать метод `yii\db\ActiveRecord::getDirtyAttributes()` для получения текущего списка dirty-атрибутов. Вы также можете вызвать `yii\db\ActiveRecord::markAttributeDirty()`, чтобы явно пометить атрибут в качестве dirty-атрибута.

Если вам нужны значения атрибутов, какими они были до их изменения, вы можете вызвать `getOldAttributes()` или `getOldAttribute()`.

Примечание: Сравнение старых и новых значений будет осуществлено с помощью оператора `===`, так что значение будет считаться dirty-значением даже в том случае, если оно осталось таким же, но изменило свой тип. Это часто происходит, когда модель получает пользовательский ввод из HTML-форм, где каждое значение представлено строкой. Чтобы убедиться в корректности типа данных, например для целых значений, вы можете применить *фильтрацию данных*: `['attributeName', 'filter', 'filter' => 'intval']`.

Значения атрибутов по умолчанию

Некоторые столбцы ваших таблиц могут иметь значения по умолчанию, объявленные в базе данных. Иногда вы можете захотеть предварительно заполнить этими значениями вашу веб-форму, которая соответствует Active Record объекту. Чтобы избежать повторного указания этих значений, вы можете вызвать метод `loadDefaultValues()` для заполнения соответствующих Active Record атрибутов значениями по умолчанию, объявленными в базе данных:

```
$customer = new Customer();  
$customer->loadDefaultValues();  
// $customer->xyz получит значение по умолчанию, которое было указано при  
// объявлении столбца "xyz"
```

Приведение типов атрибутов

При заполнении результатами запроса `yii\db\ActiveRecord` производит автоматическое приведение типов для значений атрибутов на основе информации из [схемы базы данных](#). Это позволяет данным, полученным из колонки таблицы объявленной как целое, заноситься в экземпляр `ActiveRecord` как значение целого типа PHP, булево как булево и т.д. Однако, механизм приведения типов имеет несколько ограничений:

- Числа с плавающей точкой не будут обработаны, а будут представлены как строки, в противном случае они могут потерять точность.
- Ковертация целых чисел зависит от разрядности используемой операционной системы. В частности: значения колонок, объявленных как 'unsigned integer' или 'big integer' будут приведены к целому типу PHP только на 64-х разрядных системах, в то время как на 32-х разрядных - они будут представлены как строки.

Имейте в виду, что преобразование типов производится только в момент заполнения экземпляра `ActiveRecord` данными из результата запроса. При заполнении данных из HTTP запроса или непосредственно через механизм доступа к полям - автоматическая конвертация не производится. Схема таблицы базы данных также используется при построении SQL запроса для сохранения данных `ActiveRecord`, обеспечивая соответствие типов связываемых параметров в запросе. Однако, над атрибутами объекта `ActiveRecord` не будет производиться приведение типов в процессе сохранения.

Совет: вы можете использовать поведение `yii\behaviors\AttributeTypecastBehavior` для того, чтобы производить приведение типов для `ActiveRecord` во время валидации или сохранения.

Обновление нескольких строк данных

Методы, представленные выше, работают с отдельными `Active Record` объектами, иницилируя вставку или обновление данных для отдельной строки таблицы. Вместо них для обновления нескольких строк одновременно можно использовать метод `updateAll()`, который является статическим.

```
// UPDATE 'customer' SET 'status' = 1 WHERE 'email' LIKE '%@example.com%'  
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email',  
    '@example.com']);
```

Подобным образом можно использовать метод `updateAllCounters()` для обновления значений столбцов-счётчиков в нескольких строках одновременно.

```
// UPDATE 'customer' SET 'age' = 'age' + 1
Customer::updateAllCounters(['age' => 1]);
```

6.3.6 Удаление данных

Для удаления одной отдельной строки данных сначала получите `Active Record` объект, соответствующий этой строке, а затем вызовите метод `yii\db\ActiveRecord::delete()`.

```
$customer = Customer::findOne(123);
$customer->delete();
```

Вы можете вызвать `yii\db\ActiveRecord::deleteAll()` для удаления всех или нескольких строк данных одновременно. Например:

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

Примечание: будьте очень осторожны, используя метод `deleteAll()`, потому что он может полностью удалить все данные из вашей таблицы, если вы сделаете ошибку при указании условий удаления.

6.3.7 Жизненные циклы Active Record

Важно понимать как устроены жизненные циклы `Active Record` при использовании `Active Record` для различных целей. В течение каждого жизненного цикла вызывается определённая последовательность методов, которые вы можете переопределять, чтобы получить возможность тонкой настройки жизненного цикла. Для встраивания своего кода вы также можете отвечать на конкретные события `Active Record`, которые срабатывают в течение жизненного цикла. Эти события особенно полезны, когда вы разрабатываете **поведения**, которые требуют тонкой настройки жизненных циклов `Active Record`.

Ниже мы подробно опишем различные жизненные циклы `Active Record` и методы/события, которые участвуют в жизненных циклах.

Жизненный цикл создания нового объекта

Когда создаётся новый объект `Active Record` с помощью оператора `new`, следующий жизненный цикл имеет место:

1. Вызывается конструктор класса;
2. Вызывается `init()`: иницируется событие `EVENT_INIT`.

Жизненный цикл получения данных

Когда происходит получение данных посредством одного из методов получения данных, каждый вновь создаваемый объект Active Record при заполнении данными проходит следующий жизненный цикл:

1. Вызывается конструктор класса.
2. Вызывается `init()`: инициируется событие `EVENT_INIT`.
3. Вызывается `afterFind()`: инициируется событие `EVENT_AFTER_FIND`.

Жизненный цикл сохранения данных

Когда вызывается метод `save()` для вставки или обновления объекта Active Record, следующий жизненный цикл имеет место:

1. Вызывается `beforeValidate()`: инициируется событие `EVENT_BEFORE_VALIDATE`. Если метод возвращает `false` или свойство события `yii\base\ModelEvent::$isValid` равно `false`, оставшиеся шаги не выполняются.
2. Осуществляется валидация данных. Если валидация закончилась неудачей, после 3-го шага остальные шаги не выполняются.
3. Вызывается `afterValidate()`: инициируется событие `EVENT_AFTER_VALIDATE`.
4. Вызывается `beforeSave()`: инициируется событие `EVENT_BEFORE_INSERT` или событие `EVENT_BEFORE_UPDATE`. Если метод возвращает `false` или свойство события `yii\base\ModelEvent::$isValid` равно `false`, оставшиеся шаги не выполняются.
5. Осуществляется фактическая вставка или обновление данных в базу данных;
6. Вызывается `afterSave()`: инициируется событие `EVENT_AFTER_INSERT` или событие `EVENT_AFTER_UPDATE`.

Жизненный цикл удаления данных

Когда вызывается метод `delete()` для удаления объекта Active Record, следующий жизненный цикл имеет место:

1. Вызывается `beforeDelete()`: инициируется событие `EVENT_BEFORE_DELETE`. Если метод возвращает `false` или свойство события `yii\base\ModelEvent::$isValid` равно `false`, остальные шаги не выполняются.
2. Осуществляется фактическое удаление данных из базы данных.
3. Вызывается `afterDelete()`: инициируется событие `EVENT_AFTER_DELETE`.

Примечание: Вызов следующих методов НЕ инициирует ни один из вышеприведённых жизненных циклов:

- yii\db\ActiveRecord::updateAll()
- yii\db\ActiveRecord::deleteAll()
- yii\db\ActiveRecord::updateCounters()
- yii\db\ActiveRecord::updateAllCounters()

6.3.8 Работа с транзакциями

Есть два способа использования [транзакций](#) при работе с ActiveRecord.

Первый способ заключается в том, чтобы явно заключить все вызовы методов ActiveRecord в блок транзакции как показано ниже:

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // другие... операции с базой данных...
});

// или по-другому

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // другие... операции с базой данных...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

Второй способ заключается в том, чтобы перечислить операции с базой данных, которые требуют транзакционного выполнения, в методе yii\db\ActiveRecord::transactions(). Например:

```
class Customer extends ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // вышеприведённая строка эквивалентна следующей:
            // 'api' => self::OP_ALL,
        ];
    }
}
```


Метод `yii\db\ActiveRecord::transactions()` должен возвращать массив, ключи которого являются именами [сценариев](#), а значения соответствуют операциям, которые должны быть выполнены с помощью транзакций. Вы должны использовать следующие константы для обозначения различных операций базы данных:

- `OP_INSERT`: операция вставки, осуществляемая с помощью метода `insert()`;
- `OP_UPDATE`: операция обновления, осуществляемая с помощью метода `update()`;
- `OP_DELETE`: операция удаления, осуществляемая с помощью метода `delete()`.

Используйте операторы `|` для объединения вышеприведённых констант при обозначении множества операций. Вы можете также использовать вспомогательную константу `OP_ALL`, чтобы обозначить одной константой все три вышеприведённые операции.

6.3.9 Оптимистическая блокировка

Оптимистическая блокировка - это способ предотвращения конфликтов, которые могут возникать, когда одна и та же строка данных обновляется несколькими пользователями. Например, пользователь А и пользователь В одновременно редактируют одну и ту же wiki-статью. После того, как пользователь А сохранит свои изменения, пользователь В нажимает на кнопку “Сохранить” в попытке также сохранить свои изменения. Т.к. пользователь В работал с фактически-устаревшей версией статьи, было бы неплохо иметь способ предотвратить сохранение его варианта статьи и показать ему некоторое сообщение с подсказкой о том, что произошло.

Оптимистическая блокировка решает вышеприведённую проблему за счёт использования отдельного столбца для сохранения номера версии каждой строки данных. Когда строка данных сохраняется с использованием устаревшего номера версии, выбрасывается исключение `yii\db\StaleObjectException`, которое предохраняет строку от сохранения. Оптимистическая блокировка поддерживается только тогда, когда вы обновляете или удаляете существующую строку данных, используя методы `yii\db\ActiveRecord::update()` или `yii\db\ActiveRecord::delete()` соответственно.

Для использования оптимистической блокировки:

1. Создайте столбец в таблице базы данных, ассоциированной с классом `Active Record`, для сохранения номера версии каждой строки данных. Столбец должен быть типа `big integer` (в `Mysql` это будет `BIGINT DEFAULT 0`).
2. Переопределите метод `yii\db\ActiveRecord::optimisticLock()` таким образом, чтобы он возвращал название этого столбца.

3. В веб-форме, которая принимает пользовательский ввод, добавьте скрытое поле для сохранения текущей версии обновляемой строки. Убедитесь, что для вашего атрибута с версией объявлены правила валидации, и валидация проходит успешно.
4. В действии контроллера, которое занимается обновлением строки данных с использованием Active Record, оберните в блок `try...catch` код и перехватывайте исключение `yii\db\StaleObjectException`. Реализуйте необходимую бизнес-логику (например, возможность слияния изменений, подсказку о том, что данные устарели) для разрешения возникшего конфликта.

Например, предположим, что столбец с версией называется `version`. Вы можете реализовать оптимистическую блокировку с помощью подобного кода:

```
// ----- код представления -----  
  
use yii\helpers\Html;  
  
// другие... поля ввода  
echo Html::activeHiddenInput($model, 'version');  
  
// ----- код контроллера -----  
  
use yii\db\StaleObjectException;  
  
public function actionUpdate($id)  
{  
    $model = $this->findModel($id);  
  
    try {  
        if ($model->load(Yii::$app->request->post()) && $model->save()) {  
            return $this->redirect(['view', 'id' => $model->id]);  
        } else {  
            return $this->render('update', [  
                'model' => $model,  
            ]);  
        }  
    } catch (StaleObjectException $e) {  
        // логика разрешения конфликта версий  
    }  
}
```

6.3.10 Работа со связными данными

Помимо работы с отдельными таблицами баз данных, Active Record также имеет возможность объединять связные данные, что делает их легкодоступными для получения через основные объекты данных. Например,

данные покупателя связаны с данными заказов, потому что один покупатель может осуществить один или несколько заказов. С помощью объявления этой связи вы можете получить возможность доступа к информации о заказе покупателя с помощью выражения `$customer->orders`, которое возвращает информацию о заказе покупателя в виде массива объектов класса `Order`, которые являются Active Record объектами.

Объявление связей

Для работы со связными данными посредством Active Record вы прежде всего должны объявить связи в классе Active Record. Эта задача решается простым объявлением *методов получения связных данных* для каждой интересующей вас связи как показано ниже:

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

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
        ;
    }
}
```

В вышеприведённом коде мы объявили связь `orders` для класса `Customer` и связь `customer` для класса `Order`.

Каждый метод получения связных данных должен быть назван в формате `getXyz`. Мы называем `xyz` (первая буква в нижнем регистре) *именем связи*. Помните, что имена связей чувствительны к регистру.

При объявлении связи, вы должны указать следующую информацию:

- кратность связи: указывается с помощью вызова метода `hasMany()` или метода `hasOne()`. В вышеприведённом примере вы можете легко увидеть в объявлениях связей, что покупатель может иметь много заказов в то время, как заказ может быть сделан лишь одним покупателем.
- название связного Active Record класса: указывается в качестве первого параметра для метода `hasMany()` или для метода `hasOne()`. Рекомендуется использовать код `Xyz::className()`, чтобы получить строку с именем класса, при этом вы сможете воспользоваться возможностями авто-дополнения кода, встроенного в IDE, а также по-

лучите обработку ошибок на этапе компиляции.

- связь между двумя типами данных: указываются столбцы с помощью которых два типа данных связаны. Значения массива - это столбцы основного объекта данных (представлен классом `Active Record`, в котором объявляется связь), в то время как ключи массива - столбцы связанных данных.

Есть простой способ запомнить это правило: как вы можете увидеть в примере выше, столбец связанной `Active Record` указывается сразу же после указания самого класса `Active Record`. Вы видите, что `customer_id` - это свойство класса `Order`, а `id` - свойство класса `Customer`.

Доступ к связным данным

После объявления связей вы можете получать доступ к связным данным с помощью имён связей. Это происходит таким же образом, каким осуществляется доступ к свойству объекта объявленному с помощью метода получения связных данных. По этой причине, мы называем его *свойством связи*. Например:

```
// SELECT * FROM 'customer' WHERE 'id' = 123
$customer = Customer::findOne(123);

// SELECT * FROM 'order' WHERE 'customer_id' = 123
// $orders - это массив объектов Order
$orders = $customer->orders;
```

Информация: когда вы объявляете связь с названием `xyz` посредством геттера `getXYZ()`, у вас появляется возможность доступа к свойству `xyz` подобно свойству объекта. Помните, что название связи чувствительно к регистру.

Если связь объявлена с помощью метода `hasMany()`, доступ к свойству связи вернёт массив связных объектов `Active Record`; если связь объявлена с помощью метода `hasOne()`, доступ к свойству связи вернёт связный `Active Record` объект или `null`, если связные данные не найдены.

Когда вы запрашиваете свойство связи в первый раз, выполняется SQL-выражение как показано в примере выше. Если то же самое свойство запрашивается вновь, будет возвращён результат предыдущего SQL-запроса без повторного выполнения SQL-выражения. Для принудительного повторного выполнения SQL-запроса, вы можете удалить свойство связи с помощью операции: `unset($customer->orders)`.

Примечание: Несмотря на то, что эта концепция выглядит похожей на концепцию свойств объектов, между ними есть важное различие. Для обычных свойств объектов значения

свойств имеют тот же тип, который возвращает геттер. Однако метод получения связанных данных возвращает объект `yii\db\ActiveQuery`, в то время как доступ к свойству связи возвращает объект `yii\db\ActiveRecord` или массив таких объектов. `'php $customer->orders; // массив объектов Order $customer->getOrders(); // объект ActiveQuery'` Это полезно при тонкой настройке запросов к связным данным, что будет описано в следующем разделе.

Динамические запросы связанных данных

Т.к. метод получения связанных данных возвращает объект запроса `yii\db\ActiveQuery`, вы можете в дальнейшем перед его отправкой в базу данных настроить этот запрос, используя методы построения запросов. Например:

```
$customer = Customer::findOne(123);

// SELECT * FROM 'order' WHERE 'customer_id' = 123 AND 'subtotal' > 200
// ORDER BY 'id'
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

В отличие от доступа к данным с помощью свойства связи, каждый раз при выполнении такого динамического запроса посредством метода получения связанных данных будет выполняться SQL-запрос, даже если тот же самый динамический запрос был отправлен ранее.

Иногда вы можете даже захотеть настроить объявление связи таким образом, чтобы вы могли более просто осуществлять динамические запросы связанных данных. Например, вы можете объявить связь `bigOrders` как показано ниже:

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

После этого вы сможете выполнять следующие запросы связанных данных:

```
// SELECT * FROM 'order' WHERE 'customer_id' = 123 AND 'subtotal' > 200
// ORDER BY 'id'
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM 'order' WHERE 'customer_id' = 123 AND 'subtotal' > 100
// ORDER BY 'id'
```

```
$orders = $customer->bigOrders;
```

Связывание посредством промежуточной таблицы

При проектировании баз данных, когда между двумя таблицами имеется кратность связи many-to-many, обычно вводится промежуточная таблица¹⁶. Например, таблицы `order` и `item` могут быть связаны посредством промежуточной таблицы с названием `order_item`. Один заказ будет соотноситься с несколькими товарами, в то время как один товар будет также соотноситься с несколькими заказами.

При объявлении подобных связей вы можете пользоваться методом `via()` или методом `viaTable()` для указания промежуточной таблицы. Разница между методами `via()` и `viaTable()` заключается в том, что первый метод указывает промежуточную таблицу с помощью названия связи, в то время как второй метод непосредственно указывает промежуточную таблицу. Например:

```
class Order extends ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

или по-другому:

```
class Order extends ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::className(), ['order_id' => 'id']);
    }

    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

Использовать связи, объявленные с помощью промежуточных таблиц, можно точно также, как и обычные связи. Например:

```
// SELECT * FROM 'order' WHERE 'id' = 100
$order = Order::findOne(100);

// SELECT * FROM 'order_item' WHERE 'order_id' = 100
// SELECT * FROM 'item' WHERE 'item_id' IN (...)
```

¹⁶http://en.wikipedia.org/wiki/Junction_table

```
// возвращает массив объектов Item
$item = $order->items;
```

Отложенная и жадная загрузка

В разделе Доступ к связным данным, мы показывали, что вы можете получать доступ к свойству связи объекта Active Record точно также, как получаете доступ к свойству обычного объекта. SQL-запрос будет выполнен только во время первого доступа к свойству связи. Мы называем подобный способ получения связных данных *отложенной загрузкой*. Например:

```
// SELECT * FROM 'customer' WHERE 'id' = 123
$customer = Customer::findOne(123);

// SELECT * FROM 'order' WHERE 'customer_id' = 123
$orders = $customer->orders;

// SQL-запрос не выполняется
$orders2 = $customer->orders;
```

Отложенная загрузка очень удобна в использовании. Однако этот метод может вызвать проблемы производительности, когда вам понадобится получить доступ к тем же самым свойствам связей для нескольких объектов Active Record. Рассмотрите следующий пример кода. Сколько SQL-запросов будет выполнено?

```
// SELECT * FROM 'customer' LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM 'order' WHERE 'customer_id' = ...
    $orders = $customer->orders;
}
```

Как вы могли заметить по вышеприведённым комментариям кода, будет выполнен 101 SQL-запрос! Это произойдёт из-за того, что каждый раз внутри цикла будет выполняться SQL-запрос при получении доступа к свойству связи `orders` каждого отдельного объекта `Customer`.

Для решения этой проблемы производительности вы можете, как показано ниже, использовать подход, который называется *жадной загрузкой*:

```
// SELECT * FROM 'customer' LIMIT 100;
// SELECT * FROM 'orders' WHERE 'customer_id' IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
```

```
// SQL-запрос не выполняется
$orders = $customer->orders;
}
```

Посредством вызова метода `yii\db\ActiveQuery::with()`, вы указываете объекту Active Record вернуть заказы первых 100 покупателей с помощью одного SQL-запроса. В результате снижаете количество выполняемых SQL-запросов от 101 до 2!

Вы можете жадно загружать одну или несколько связей. Вы можете даже жадно загружать *вложенные связи*. Вложенная связь - это связь, которая объявлена внутри связанного Active Record класса. Например, `Customer` связан с `Order` посредством связи `orders`, а `Order` связан с `Item` посредством связи `items`. При формировании запроса для `Customer`, вы можете жадно загрузить `items`, используя нотацию вложенной связи `orders.items`.

Ниже представлен код, который показывает различные способы использования метода `with()`. Мы полагаем, что класс `Customer` имеет две связи: `orders` и `country` - в то время как класс `Order` имеет лишь одну связь `items`.

```
// жадная загрузка "orders" и "country" одновременно
$customers = Customer::find()->with('orders', 'country')->all();
// аналог с использованием синтаксиса массива
$customers = Customer::find()->with(['orders', 'country'])->all();
// SQL-запрос не выполняется
$orders= $customers[0]->orders;
// SQL-запрос не выполняется
$country = $customers[0]->country;

// жадная загрузка связи "orders" и вложенной связи "orders.items"
$customers = Customer::find()->with('orders.items')->all();
// доступ к деталям первого заказа первого покупателя
// SQL-запрос не выполняется
$items = $customers[0]->orders[0]->items;
```

Вы можете жадно загрузить более глубокие вложенные связи, такие как `a.b.c.d`. Все родительские связи будут жадно загружены. Таким образом, когда вы вызываете метод `with()` с параметром `a.b.c.d`, вы жадно загрузите связи `a`, `a.b`, `a.b.c` и `a.b.c.d`.

Информация: В целом, когда жадно загружается N связей, среди которых M связей объявлено с помощью промежуточной таблицы, суммарное количество выполняемых SQL-запросов будет равно $N+M+1$. Заметьте, что вложенная связь `a.b.c.d` насчитывает 4 связи.

Когда связь жадно загружается, вы можете настроить соответствующий запрос получения связанных данных с использованием анонимной функции. Например:


```
// найти покупателей и получить их вместе с их странами и активными заказами
// SELECT * FROM 'customer'
// SELECT * FROM 'country' WHERE 'id' IN (...)
// SELECT * FROM 'order' WHERE 'customer_id' IN (...) AND 'status' = 1
$customers = Customer::find()->with([
    'country',
    'orders' => function ($query) {
        $query->andWhere(['status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

Когда настраивается запрос на получение связанных данных для какой-либо связи, вы можете указать название связи в виде ключа массива и использовать анонимную функцию в качестве соответствующего значения этого массива. Анонимная функция получит параметр `$query`, который представляет собой объект `yii\db\ActiveQuery`, используемый для выполнения запроса на получение связанных данных для данной связи. В вышеприведённом примере кода мы изменили запрос на получение связанных данных, наложив на него дополнительное условие выборки статуса заказов.

Примечание: Если вы вызываете метод `select()` в процессе жадной загрузки связей, вы должны убедиться, что будут выбраны столбцы, участвующие в объявлении связей. Иначе связанные модели будут загружены неправильно. Например:

```
$orders = Order::find()->select(['id', 'amount'])->with('customer')->all();
// $orders[0]->customer всегда равно null. Для исправления
// проблемы вы должны сделать следующее:
$orders = Order::find()->select(['id', 'amount', 'customer_id'])->with('customer')->all();
```

Использование JOIN со связями

Примечание: Материал этого раздела применим только к реляционным базам данных, таким как MySQL, PostgreSQL, и т.д.

Запросы на получение связанных данных, которые мы рассмотрели выше, ссылаются только на столбцы основной таблицы при извлечении основной информации. На самом же деле нам часто нужно сослаться в запросах на столбцы связанных таблиц. Например, мы можем захотеть получить покупателей, для которых имеется хотя бы один активный заказ. Для решения этой проблемы мы можем построить запрос с использованием JOIN как показано ниже:

```
// SELECT 'customer'.* FROM 'customer'
// LEFT JOIN 'order' ON 'order'.'customer_id' = 'customer'.'id'
```

```
// WHERE 'order'.'status' = 1
//
// SELECT * FROM 'order' WHERE 'customer_id' IN (...)
$customers = Customer::find()
    ->select('customer.*')
    ->leftJoin('order', 'order'.'customer_id' = 'customer'.'id')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->with('orders')
    ->all();
```

Примечание: Важно однозначно указывать в SQL-выражениях имена столбцов при построении запросов на получение связанных данных с участием оператора JOIN. Наиболее распространённая практика - предварять названия столбцов с помощью имён соответствующих им таблиц.

Однако лучшим подходом является использование имеющихся объявлений связей с помощью вызова метода `yii\db\ActiveQuery::joinWith()`:

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

Оба подхода выполняют одинаковый набор SQL-запросов. Однако второй подход более прозрачен и прост.

По умолчанию, метод `joinWith()` будет использовать конструкцию `LEFT JOIN` для объединения основной таблицы со связанной. Вы можете указать другой тип операции JOIN (например, `RIGHT JOIN`) с помощью третьего параметра этого метода - `$joinType`. Если же вам нужен `INNER JOIN`, вы можете вместо этого просто вызвать метод `innerJoinWith()`.

Вызов метода `joinWith()` будет жадно загружать связанные данные по умолчанию. Если вы не хотите получать связанные данные, вы можете передать во втором параметре `$eagerLoading` значение `false`.

Подобно методу `with()` вы можете объединять данные с одной или несколькими связями; вы можете настроить запрос на получение связанных данных “на лету”; вы можете объединять данные с вложенными связями; вы можете смешивать использование метода `with()` и метода `joinWith()`. Например:

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

Иногда во время объединения двух таблиц вам может потребоваться указать некоторые дополнительные условия рядом с оператором `ON` во

время выполнения JOIN-запроса. Это можно сделать с помощью вызова метода `yii\db\ActiveQuery::onCondition()` как показано ниже:

```
// SELECT 'customer'.* FROM 'customer'
// LEFT JOIN 'order' ON 'order'.'customer_id' = 'customer'.'id' AND 'order'
//      'status' = 1
//
// SELECT * FROM 'order' WHERE 'customer_id' IN (...)
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->onCondition(['order.status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

Вышеприведённый запрос вернёт *всех* покупателей и для каждого покупателя вернёт все активные заказы. Заметьте, что это поведение отличается от нашего предыдущего примера, в котором возвращались только покупатели, у которых был как минимум один активный заказ.

Информация: Когда в объекте `yii\db\ActiveQuery` указано условие выборки с помощью метода `onCondition()`, это условие будет размещено в конструкции `ON`, если запрос содержит оператор `JOIN`. Если же запрос не содержит оператор `JOIN`, такое условие будет автоматически размещено в конструкции `WHERE`.

Обратные связи

Объявления связей часто взаимны между двумя Active Record классами. Например, `Customer` связан с `Order` посредством связи `orders`, а `Order` взаимно связан с `Customer` посредством связи `customer`.

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

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
        ;
    }
}
```

Теперь рассмотрим следующий участок кода:

```
// SELECT * FROM 'customer' WHERE 'id' = 123
```

```
$customer = Customer::findOne(123);

// SELECT * FROM 'order' WHERE 'customer_id' = 123
$order = $customer->orders[0];

// SELECT * FROM 'customer' WHERE 'id' = 123
$customer2 = $order->customer;

// выведет "not the same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

Мы думали, что `$customer` и `$customer2` эквивалентны, но оказалось, что нет! Фактически они содержат одинаковые данные, но являются разными объектами. Когда мы получаем доступ к данным посредством `$order->customer`, выполняется дополнительный SQL-запрос для заполнения нового объекта `$customer2`.

Чтобы избежать избыточного выполнения последнего SQL-запроса в вышеприведённом примере, мы должны подсказать Yii, что `customer` - обратная связь относительно `orders`, и сделаем это с помощью вызова метода `inverseOf()` как показано ниже:

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']->
            inverseOf('customer'));
    }
}
```

Теперь, после этих изменений в объявлении связи, получим:

```
// SELECT * FROM 'customer' WHERE 'id' = 123
$customer = Customer::findOne(123);

// SELECT * FROM 'order' WHERE 'customer_id' = 123
$order = $customer->orders[0];

// SQL-запрос не выполняется
$customer2 = $order->customer;

// выведет "same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

Примечание: обратные связи не могут быть объявлены для связей, использующих промежуточную таблицу. То есть, если связь объявлена с помощью методов `via()` или `viaTable()`, вы не должны вызывать после этого метод `inverseOf()`.

6.3.11 Сохранение связанных данных

Во время работы со связными данными вам часто требуется установить связи между двумя разными видами данных или удалить существующие связи. Это требует установки правильных значений для столбцов, с помощью которых заданы связи. При использовании Active Record вам может понадобиться завершить участок кода следующим образом:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// установка атрибута, которой задаёт связь "customer" в объекте Order
$order->customer_id = $customer->id;
$order->save();
```

Active Record предоставляет метод `link()`, который позволяет выполнить эту задачу более красивым способом:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

Метод `link()` требует указать название связи и целевой объект Active Record, с которым должна быть установлена связь. Метод изменит значения атрибутов, которые связывают два объекта Active Record, и сохранит их в базу данных. В вышеприведённом примере, метод присвоит атрибуту `customer_id` объекта `Order` значение атрибута `id` объекта `Customer` и затем сохранит его в базу данных.

Примечание: Невозможно связать два свежесозданных объекта Active Record.

Преимущество метода `link()` становится ещё более очевидным, когда связь объявлена посредством промежуточной таблицы. Например, вы можете использовать следующий код, чтобы связать объект `Order` с объектом `Item`:

```
$order->link('items', $item);
```

Вышеприведённый код автоматически вставит строку данных в промежуточную таблицу `order_item`, чтобы связать объект `order` с объектом `item`.

Информация: Метод `link()` не осуществляет какую-либо валидацию данных во время сохранения целевого объекта Active Record. На вас лежит ответственность за валидацию любых введённых данных перед вызовом этого метода.

Существует противоположная операция для `link()` - это операция `unlink()`, она снимает существующую связь с двух объектов Active Record. Например:

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();  
$customer->unlink('orders', $customer->orders[0]);
```

По умолчанию метод `unlink()` задаст вторичному ключу (или ключам), который определяет существующую связь, значение `null`. Однако вы можете запросить удаление строки таблицы, которая содержит значение вторичного ключа, передав значение `true` в параметре `$delete` для этого метода.

Если связь построена на основе промежуточной таблицы, вызов метода `unlink()` иницирует очистку вторичных ключей в промежуточной таблице, или же удаление соответствующей строки данных в промежуточной таблице, если параметр `$delete` равен `true`.

6.3.12 Связывание объектов из разных баз данных

Active Record позволяет вам объявить связи между классами Active Record, которые относятся к разным базам данных. Базы данных могут быть разных типов (например, MySQL и PostgreSQL или MS SQL и MongoDB), и они могут быть запущены на разных серверах. Вы можете использовать тот же самый синтаксис для осуществления запросов выборки связанных данных. Например:

```
// Объект Customer соответствует таблице "customer" в реляционной базе  
данных например( MySQL)  
class Customer extends \yii\db\ActiveRecord  
{  
    public static function tableName()  
    {  
        return 'customer';  
    }  
  
    public function getComments()  
    {  
        // у покупателя может быть много комментариев  
        return $this->hasMany(Comment::className(), ['customer_id' => 'id']);  
    }  
}  
  
// Объект Comment соответствует коллекции "comment" в базе данных MongoDB  
class Comment extends \yii\mongodb\ActiveRecord  
{  
    public static function collectionName()  
    {  
        return 'comment';  
    }  
}
```

```
public function getCustomer()
{
    // комментарий принадлежит одному покупателю
    return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
;
}
}

$customers = Customer::find()->with('comments')->all();
```

Вы можете использовать большую часть возможностей запросов получения связанных данных, которые были описаны в этой главе.

Примечание: Применимость метода `joinWith()` ограничена базами данных, которые позволяют выполнять запросы между разными базами с использованием оператора JOIN. По этой причине вы не можете использовать этот метод в вышеприведённом примере, т.к. MongoDB не поддерживает операцию JOIN.

6.3.13 Тонкая настройка классов Query

По умолчанию все запросы данных для Active Record поддерживаются с помощью класса `yii\db\ActiveQuery`. Для использования собственного класса запроса вам необходимо переопределить метод `yii\db\ActiveRecord::find()` и возвращать из него объект вашего собственного класса запроса. Например:

```
namespace app\models;

use yii\db\ActiveRecord;
use yii\db\ActiveQuery;

class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}

class CommentQuery extends ActiveQuery
{
    // ...
}
```

Теперь, когда вы будете осуществлять получение данных (например, выполните `find()`, `findOne()`) или объявите связь (например, `hasOne()`) с объектом `Comment`, вы будете работать с объектом класса `CommentQuery` вместо `ActiveQuery`.

Подсказка: В больших проектах рекомендуется использовать собственные классы запросов, которые будут содержать в себе большую часть кода, связанного с настройкой запросов, таким образом классы Active Record удастся сохранить более чистыми.

Вы можете настроить класс запроса большим количеством различных способов для улучшения методик построения запросов. Например, можете объявить новые методы построения запросов в собственном классе запросов:

```
class CommentQuery extends ActiveQuery
{
    public function active($state = true)
    {
        return $this->andWhere(['active' => $state]);
    }
}
```

Примечание: Вместо вызова метода `where()` старайтесь во время объявления новых методов построения запросов использовать `andWhere()` или `orWhere()` для добавления дополнительных условий, в этом случае уже заданные условия выборок не будут перезаписаны.

Это позволит вам писать код построения запросов как показано ниже:

```
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

Вы также можете использовать новые методы построения запросов, когда объявляете связи для класса `Comment` или осуществляете запрос для выборки связанных данных:

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

$customers = Customer::find()->with('activeComments')->all();

// или по-другому:

$customers = Customer::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```


Информация: В Yii версии 1.1 была концепция с названием *score*. Она больше не поддерживается в Yii версии 2.0, и вы можете использовать собственные классы запросов и собственные методы построения запросов, чтобы добиться той же самой цели.

6.3.14 Получение дополнительных атрибутов

Когда объект Active Record заполнен результатами запроса, его атрибуты заполнены значениями соответствующих столбцов из полученного набора данных.

Вы можете получить дополнительные столбцы или значения с помощью запроса и сохранить их внутри объекта Active Record. Например, предположим, что у нас есть таблица 'room', которая содержит информацию о доступных в отеле комнатах. Каждая комната хранит информацию о её геометрических размерах с помощью атрибутов 'length', 'width', 'height'. Представьте, что вам требуется получить список всех доступных комнат, отсортированных по их объёму в порядке убывания. В этом случае вы не можете вычислять объём с помощью PHP, потому что нам требуется сортировать записи по объёму, но вы также хотите отображать объём в списке. Для достижения этой цели, вам необходимо объявить дополнительный атрибут в вашем Active Record классе 'Room', который будет хранить значение 'volume':

```
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

Далее вам необходимо составить запрос, который вычисляет объём комнаты и выполняет сортировку:

```
$rooms = Room::find()
    ->select([
        '{{room}}.*', // получить все столбцы
        '([[length]] * [[width]] * [[height]]) AS volume', // вычислить
        объём
    ])
    ->orderBy('volume DESC') // отсортировать
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // содержит значение, вычисленное с помощью SQL-
    запроса
}
```

Возможность выбирать дополнительные атрибуты может быть особенно полезной для агрегирующих запросов. Представьте, что вам необходимо

отображать список покупателей с количеством их заказов. Прежде всего вам потребуется объявить класс `Customer` со связью `'orders'` и дополнительным атрибутом для хранения расчётов:

```
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

После этого вы сможете составить запрос, который объединяет заказы и вычисляет их количество:

```
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // получить все атрибуты покупателя
        'COUNT('{{order}}.id) AS ordersCount' // вычислить количество заказов
    ])
    ->joinWith('orders') // обеспечить построение промежуточной таблицы
    ->groupBy('{{customer}}.id') // сгруппировать результаты, чтобы
        заставить агрегацию работать
    ->all();
```

Недостаток этого подхода заключается в том, что если данные для поля не загружены по результатам SQL запроса, то они должны быть вычисленны отдельно. Это означает, что запись, полученная посредством обычного запроса без дополнительных полей в разделе `'select'`, не может вернуть реальные значения для дополнительного поля. Это же касается и только что сохраненной записи.

```
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // значение будет равно 'null', тк.. поле не было заполнено
```

Использование магических методов `__get()` и `__set()` позволяет эмулировать поведение обычного поля:

```
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }
}
```

```

public function getVolume()
{
    if (empty($this->length) || empty($this->width) || empty($this->
height)) {
        return null;
    }

    if ($this->_volume === null) {
        $this->setVolume(
            $this->length * $this->width * $this->height
        );
    }

    return $this->_volume;
}

// ...
}

```

Если результат запроса на выборку данных не содержит поле ‘volume’, то модель сможет рассчитать его автоматически используя имеющиеся атрибуты.

Вы также можете вычислять агрегируемые поля используя объявленные отношения:

```

class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }

    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // нет смысла выполнять запрос на поиск по пустым
            ключам
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount($this->getOrders()->count()); //
            вычисляем агрегацию по требованию из отношения
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {

```

```

        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

```

При такой реализации, в случае когда 'ordersCount' присутствует в разделе 'select' - значение 'Customer::ordersCount' будет заполнено из результатов запроса, в противном случае - оно будет вычислено по первому требованию на основании отношения Customer::orders.

Этот подход также можно использовать для быстрого доступа к некоторым данным отношений, в особенности для агрегации. Например:

```

class Customer extends \yii\db\ActiveRecord
{
    /**
     * Объявляет виртуальное свойство для агрегируемых данных, доступное
     * только на чтение.
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // нет смысла выполнять запрос на поиск по пустым
            ключам
        }

        return $this->ordersAggregation[0]['counted'];
    }

    /**
     * Объявляет обычное отношение 'orders'.
     */
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }

    /**
     * Объявляет новое отношение, основанное на 'orders', которое
     * предоставляет агрегацию.
     */
    public function getOrdersAggregation()
    {
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // выводит агрегируемые данные из
    отношения без дополнительного запроса благодаря жадной загрузке
}

```

```
$customer = Customer::findOne($pk);  
$customer->ordersCount; // выводит агрегируемые данные отношения через  
    ленивую загрузку
```

6.4 Миграции баз данных

В ходе разработки и ведения баз данных приложений, которые управляют данными, структуры используемых баз данных развиваются, как и исходный код приложений. Например, при разработке приложения, в будущем может оказаться необходимой новая таблица; уже после того, как приложение будет развернуто в рабочем режиме (продакшене), также может быть обнаружено, что для повышения производительности запросов должен быть создан определённый индекс; и так далее. В связи с тем, что изменение структуры базы данных часто требует изменение исходного кода, yii поддерживает так называемую возможность *миграции баз данных*, которая позволяет отслеживать изменения в базах данных при помощи терминов *миграции баз данных*, которые являются системой контроля версий вместе с исходным кодом.

Следующие шаги показывают, как миграции базы данных могут быть использованы командой разработчиков в процессе разработки:

1. Илья создает новую миграцию (например, создается новая таблица или изменяется определение столбца и т.п.).
2. Илья фиксирует новую миграцию в системе управления версиями (например, в Git, Mercurial).
3. Алексей обновляет свой репозиторий из системы контроля версий и получает новую миграцию.
4. Алексей применяет миграцию к своей локальной базе данных, тем самым синхронизируя свою базу данных, для того чтобы отразить изменения, которые сделал Илья.

А следующие шаги показывают, как развернуть новый релиз с миграциями баз данных в рабочем режиме (продакшене):

1. Сергей создаёт новую версию проекта репозитория, которая содержит некоторые новые миграции баз данных.
2. Сергей обновляет исходный код на рабочем сервере до новой версии.
3. Сергей применяет любую из накопленных миграций баз данных в рабочую базу данных.

Yii предоставляет набор инструментов для миграций из командной строки, которые позволяют:

- создавать новые миграции;
- применять миграции;
- отменять миграции;
- применять миграции повторно;
- показывать историю и статус миграций;

Все эти инструменты доступны через команду `yii migrate`. В этом разделе мы опишем подробно, как выполнять различные задачи, используя эти инструменты. Вы также можете сами посмотреть как использовать каждый отдельный инструмент при помощи команды `yii help migrate`.

Подсказка: Миграции могут не только изменять схему базы данных, но и приводить данные в соответствие с новой схемой, создавать иерархию RBAC или очищать кеш.

6.4.1 Создание миграций

Чтобы создать новую миграцию, выполните следующую команду:

```
yii migrate/create <name>
```

Требуемый аргумент `name` даёт краткое описание новой миграции. Например, если миграция о создании новой таблицы с именем *news*, Вы можете использовать имя `create_news_table` и выполнить следующую команду:

```
yii migrate/create create_news_table
```

Примечание: Поскольку аргумент `name` будет использован как часть имени класса создаваемой миграции, он должен содержать только буквы, цифры и/или символы подчеркивания.

Приведенная выше команда создаст новый PHP класс с именем файла `m150101_185401_create_news_table.php` в директории `@app/migrations`. Файл содержит следующий код, который главным образом декларирует класс миграции `m150101_185401_create_news_table` с следующим каркасом кода:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }
}
```

```

public function down()
{
    echo "m101129_185401_create_news_table cannot be reverted.\n";

    return false;
}

/*
// Use safeUp/safeDown to run migration code within a transaction
public function safeUp()
{
}

public function safeDown()
{
}
*/
}

```

Каждая миграция базы данных определяется как PHP класс расширяющийся от `yii\db\Migration`. Имя класса миграции автоматически создается в формате `m<YMMDD_HHMMSS>_<Name>` (`mГодМесяцДень<ЧасыМинутыСекунды>_Имя<>`), где

- `<YMMDD_HHMMSS>` относится к UTC дате-времени при котором команда создания миграции была выполнена.
- `<Name>` это тоже самое значение аргумента `name` которое вы прописываете в команду.

В классе миграции, Вы должны прописать код в методе `up()` когда делаете изменения в структуре базы данных. Вы также можете написать код в методе `down()`, чтобы отменить сделанные `up()` изменения. Метод `up` вызывается для обновления базы данных с помощью данной миграции, а метод `down()` вызывается для отката изменений базы данных. Следующий код показывает как можно реализовать класс миграции, чтобы создать таблицу `news`:

```

<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }
}

```

```
public function down()
{
    $this->dropTable('news');
}
```

Информация: Не все миграции являются обратимыми. Например, если метод `up()` удаляет строку из таблицы, возможно что у вас уже не будет возможности вернуть эту строку методом `down()`. Иногда Вам может быть просто слишком лень реализовывать метод `down()`, в связи с тем, что это не очень распространено - откатывать миграции базы данных. В этом случае вы должны в методе `down()` вернуть `false`, чтобы указать, что миграция не является обратимой.

Базовый класс миграций `yii\db\Migration` предоставляет подключение к базе данных через свойство `db`. Вы можете использовать его для манипулирования схемой базы данных используя методы описанные в [работе со схемой базы данных](#).

Вместо использования физических типов данных, при создании таблицы или столбца, следует использовать *абстрактные типы* для того, чтобы ваша миграция являлась независимой от конкретной СУБД. Класс `yii\db\Schema` определяет набор констант для предоставления поддержки абстрактных типов. Эти константы называются в следующем формате `TYPE_<Name>`. Например, `TYPE_PK` относится к типу автоинкремента (`AUTO_INCREMENT`) первичного ключа; `TYPE_STRING` относится к строковому типу. Когда миграция применяется к конкретной базе данных, абстрактные типы будут переведены в соответствующие физические типы. В случае с MySQL, `TYPE_PK` будет превращено в `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, а `TYPE_STRING` станет `varchar(255)`.

Вы можете добавить дополнительные ограничения при использовании абстрактных типов. В приведенном выше примере, `NOT NULL` добавляется к `Schema::TYPE_STRING` чтобы указать, что столбец не может быть `NULL`.

Информация: Сопоставление абстрактных типов и физических типов определяется свойством `$typeMap` в каждом конкретном `QueryBuilder` классе.

Начиная с версии 2.0.6, появился новый построитель схем, который является более удобным инструментом для описания структуры столбцов. Теперь, при написании миграций, можно использовать такой код:

```
<?php
```



```
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

Весь список методов описания типов столбцов доступен в API документации `yii\db\SchemaBuilderTrait`.

6.4.2 Генерация миграций

Начиная с версии 2.0.7 появился удобный способ создания миграций из консоли.

В том случае, если миграция названа особым образом, таким как, например, `create_xxx_table` или `drop_xxx_table` сгенерированный файл миграции будет содержать дополнительный код.

Создание таблицы

```
yii migrate/create create_post_table
```

сгенерирует

```
class m150811_220037_create_post_table extends Migration
{
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    public function down()
    {
        $this->dropTable('post');
    }
}
```

Чтобы сразу создать поля таблицы, укажите их через опцию `--fields`.

```
yii migrate/create create_post_table --fields=title:string,body:text
```

сгенерирует

```
class m150811_220037_create_post_table extends Migration
{
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text()
        ]);
    }

    public function down()
    {
        $this->dropTable('post');
    }
}
```

Можно указать дополнительные параметры.

```
yii migrate/create create_post_table --fields=title:string(12):notNull:
unique,body:text
```

сгенерирует

```
class m150811_220037_create_post_table extends Migration
{
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    public function down()
    {
        $this->dropTable('post');
    }
}
```

Примечание: первичный ключ добавляется автоматически и по умолчанию называется `id`. Если вам необходимо другое имя, указать его можно через опцию `--fields=name:primaryKey`.

Удаление таблицы

```
yii migrate/create drop_post_table --fields=title:string(12):notNull:unique,
body:text
```

сгенерирует

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

Добавление столбца

Если имя миграции задано как `add_xxx_column_to_yyy_table`, файл будет содержать необходимые методы `addColumn` и `dropColumn`.

Для добавления столбца:

```
yii migrate/create add_position_column_to_post_table --fields=position:
integer
```

сгенерирует

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

Удаление столбца

Если имя миграции задано как `drop_xxx_column_from_yyy_table`, файл будет содержать необходимые методы `addColumn` и `dropColumn`.

```
yii migrate/create drop_position_column_from_post_table --fields=position:
integer
```

сгенерирует

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
    public function up()
    {
```

```

        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}

```

Добавление промежуточной таблицы

Если имя миграции задано как `create_junction_table_for_xxx_and_yyy_tables`, файл будет содержать код для создания промежуточной таблицы.

```
yii migrate/create create_junction_table_for_post_and_tag_tables
```

сгенерирует

```

class m150811_220037_create_junction_post_and_tag extends Migration
{
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'PRIMARY KEY(post_id, tag_id)'
        ]);

        $this->createIndex('idx-post_tag-post_id', 'post_tag', 'post_id');
        $this->createIndex('idx-post_tag-tag_id', 'post_tag', 'tag_id');

        $this->addForeignKey('fk-post_tag-post_id', 'post_tag', 'post_id', 'post', 'id', 'CASCADE');
        $this->addForeignKey('fk-post_tag-tag_id', 'post_tag', 'tag_id', 'tag', 'id', 'CASCADE');
    }

    public function down()
    {
        $this->dropTable('post_tag');
    }
}

```

Транзакции Миграций

При выполнении сложных миграций баз данных, важно обеспечить каждую миграцию либо успехом, либо ошибкой, в целом так, чтобы база данных могла поддерживать целостность и непротиворечивость. Для достижения данной цели рекомендуется, заключить операции каждой миграции базы данных в [транзакции](#).

Самый простой способ реализации транзакций миграций это прописать код миграций в методы `safeUp()` и `safeDown()`. Эти два метода отличаются от методов `up()` и `down()` тем, что они неявно заключены в транзакции. В результате, если какая-либо операция в этих методах не удастся, все предыдущие операции будут отменены автоматически.

В следующем примере, помимо создания таблицы `news` мы также вставляем в этой таблице начальную строку.

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

Обратите внимание, что обычно при выполнении нескольких операций в базе данных при помощи метода `safeUp()`, вы должны реализовать обратный порядок исполнения в методе `safeDown()`. В приведенном выше примере мы сначала создали таблицу, а затем вставили строку в `safeUp()`; а в `safeDown()` мы сначала удаляем строку и затем удаляем таблицу.

Примечание: Не все СУБД поддерживают транзакции. И некоторые запросы к базам данных не могут быть введены в транзакции. Для различных примеров, пожалуйста, обратитесь к негласным обязательствам¹⁷. В этом случае вместо этих методов вы должны реализовать методы `up()` и `down()`.

¹⁷<http://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html>

Методы доступа к базе данных

Базовый класс миграции `yii\db\Migration` предоставляет набор методов, которые позволяют Вам получить доступ и управлять базами данных. Вы можете найти эти методы, их названия аналогичны [методам DAO](#), предоставленным в классе `yii\db\Command`. Например, метод `yii\db\Migration::createTable()` позволяет создать новую таблицу, подобно методу `yii\db\Command::createTable()`.

Преимущество методов, описанных при помощи `yii\db\Migration` заключается в том, что Вам не нужно явно создавать экземпляр/копию `yii\db\Command` и исполнение каждого метода будет автоматически отображать полезные сообщения говорящие вам, что операции с базой данных выполняются и сколько они идут.

Ниже представлен список всех этих методов доступа к базам данных:

- `execute()`: выполнение SQL инструкции
- `insert()`: вставка одной строки
- `batchInsert()`: вставка нескольких строк
- `update()`: обновление строк
- `delete()`: удаление строк
- `createTable()`: создание таблицы
- `renameTable()`: переименование таблицы
- `dropTable()`: удаление таблицы
- `truncateTable()`: удаление всех строк в таблице
- `addColumn()`: добавление столбца
- `renameColumn()`: переименование столбца
- `dropColumn()`: удаление столбца
- `alterColumn()`: изменения столбца
- `addPrimaryKey()`: добавление первичного ключа
- `dropPrimaryKey()`: удаление первичного ключа
- `addForeignKey()`: добавление внешнего ключа
- `dropForeignKey()`: удаление внешнего ключа
- `createIndex()`: создание индекса
- `dropIndex()`: удаление индекса

Информация: `yii\db\Migration` не предоставляет методы запросов к базе данных. Это потому, что обычно не требуется отображать дополнительные сообщения об извлечении данных из базы данных. Это также, потому, что можно использовать более мощный [Построитель Запросов](#) для построения и выполнения сложных запросов.

Примечание: при обработке данных внутри миграции, может показаться, что использование существующих классов [Active Record](#), со всей их готовой бизнес логикой, будет разумным решением и упростит код миграции. Однако, следует

помнить, что код миграций не должен меняться, по определению. В отличие от миграций, бизнес логика приложений часто изменяется. Это может привести к нарушению работы миграции при определённых изменениях на уровне Active Record. Поэтому рекомендуется делать миграции независимыми от других частей приложения, таких как классы Active Record.

6.4.3 Применение Миграций

Для обновления базы данных до последней структуры, Вы должны применить все новые миграции, используя следующую команду:

```
yii migrate
```

Эта команда выведет список всех миграций, которые не применялись до сих пор. Если Вы подтвердите, что Вы хотите применить эти миграции, то этим самым запустите метод `up()` или `safeUp()` в каждом новом классе миграции, один за другим, в порядке их временного значения `timestamp`.

Для каждой миграции которая была успешно проведена, эта команда будет вставлять строку в таблицу базы данных с именем `migration` записав успешное проведение миграции. Это позволяет инструменту миграции выявлять какие миграции были применены, а какие - нет.

Примечание: Инструмент миграции автоматически создаст таблицу `migration` в базе данных указанной в параметре `db`. По умолчанию база данных определяется как [компонент приложения db](#).

Иногда, необходимо применить одну или несколько новых миграций, вместо всех доступных миграций. Это возможно сделать, указав, при выполнении команды, количество миграций, которые необходимо применить. Например, следующая команда будет пытаться применить следующие три доступные миграции:

```
yii migrate 3
```

Также можно явно указать конкретную миграцию, которая должна быть применена к базе данных, это можно сделать при помощи команды `migrate/to` в одном из следующих форматов:

```
yii migrate/to 150101_185401          # используя временную
метку определяющую миграцию
yii migrate/to "2015-01-01 18:54:01"  # используя строку,
которая может быть получена путем использования функции strtotime()
yii migrate/to m150101_185401_create_news_table # используя полное имя
yii migrate/to 1392853618             # используя временную
метку UNIX
```

Если раньше имелись какие-либо не применённые миграции, до указанной конкретной миграции, то все они будут применены до данной миграции. А если указанная миграция уже применялась ранее, то любые более поздние версии данной прикладной миграции будут отменены.

6.4.4 Отмена Миграций

Чтобы отменить (откатить) одну или несколько миграций, которые применялись ранее, нужно запустить следующую команду:

```
yii migrate/down      # отменяет самую последнюю применённую миграцию
yii migrate/down 3     # отменяет 3 последних применённых миграции
```

Примечание: Не все миграции являются обратимыми. При попытке отката таких миграций произойдёт ошибка и остановится весь процесс отката.

6.4.5 Перезагрузка Миграций

Под перезагрузкой миграций подразумевается, сначала последовательный откат определённых миграций, а потом применение их снова. Это может быть сделано следующим образом:

```
yii migrate/redo      # перезагрузить последнюю применённую миграцию
yii migrate/redo 3     # перезагрузить 3 последние применённые миграции
```

Примечание: Если миграция не является обратимой, Вы не сможете её перезагрузить.

6.4.6 Список Миграций

Чтобы посмотреть какие миграции были применены, а какие нет, используйте следующие команды:

```
yii migrate/history   # показать последних 10 применённых миграций
yii migrate/history 5  # показать последних 5 применённых миграций
yii migrate/history all # показать все применённые миграции

yii migrate/new       # показать первых 10 новых миграций
yii migrate/new 5     # показать первых 5 новых миграций
yii migrate/new all   # показать все новые миграции
```

6.4.7 Изменение Истории Миграций

Вместо применения или отката миграций, есть возможность просто **отметить**, что база данных была обновлена до определенной миграции. Это часто используется при ручном изменении базы данных в конкретное состояние и Вам не нужно применять миграции для того, чтобы это изменение

было повторно применено позже. Этой цели можно добиться с помощью следующей команды:

```
yii migrate/mark 150101_185401          # используя временную
метку определённой миграции
yii migrate/mark "2015-01-01 18:54:01"  # используя строку,
которая может быть получена путем использования функции strtotime()
yii migrate/mark m150101_185401_create_news_table # используя полное имя
yii migrate/mark 1392853618            # используя временную
метку UNIX
```

Эта команда изменит таблицу `migration` добавив или удалив определенные строки, тем самым указав, что к базе данных была применена указанная миграция. Никаких миграций не будет применяться или отменяться по этой команде.

6.4.8 Настройка Миграций

Есть несколько способов настроить команду миграции.

Используя параметры командной строки

В команду миграций входит несколько параметров командной строки, которые могут использоваться, для того, чтобы настроить поведение миграции:

- **interactive**: логический тип - `boolean` (по умолчанию `true`). Указывает, следует ли выполнять миграцию в интерактивном режиме. Если это значение является - `true`, то пользователю будет выдан запрос, перед выполнением командой определенных действий. Вы можете установить это значение в `false` если команда используется в фоновом режиме.
- **migrationPath**: строка - `string` (по умолчанию `@app/migrations`). Указывает каталог для хранения всех файлов классов миграций. Этот параметр может быть определён либо как путь до директории, либо как [псевдоним](#) пути. Обратите внимание, что данный каталог должен существовать, иначе команда будет выдавать ошибку.
- **migrationTable**: строка - `string` (по умолчанию `migration`). Определяет имя таблицы в базе данных в которой хранится информация о истории миграций. Эта таблица будет автоматически создана командой миграции, если её не существует. Вы также можете создать её вручную, используя структуру `version varchar(255) primary key, apply_time integer`.
- **db**: строка - `string` (по умолчанию `db`). Определяет ID базы данных [компонента приложения](#). Этот параметр представляет собой базу данных, которая подвергается миграциям при помощи команды миграций.

- **templateFile**: строка - string (по умолчанию @yii/views/migration.php). Указывает путь до файла шаблона, который используется для формирования скелета класса файлов миграции. Этот параметр может быть определён либо как путь до файла, либо как **псевдоним** пути. Файл шаблона - это PHP скрипт, в котором можно использовать предопределённую переменную с именем `$className` для того, чтобы получить имя класса миграции.
- **generatorTemplateFiles**: массив (по умолчанию `[`

```
'create_table' => '@yii/views/createTableMigration.php',
'drop_table' => '@yii/views/dropTableMigration.php',
'add_column' => '@yii/views/addColumnMigration.php',
'drop_column' => '@yii/views/dropColumnMigration.php',
'create_junction' => '@yii/views/createTableMigration.php'
```


`]`), в котором указаны файлы шаблонов для генерации миграций. Подробнее в разделе «Генерация миграций».
- **fields**: массив конфигураций столбцов, который используется для генерации кода миграции. По умолчанию пуст. Формат каждой конфигурации **имястолбца_тип:столбца_декоратор:столбца_**. Например, `--fields=name:string(12):notNull` даст нам столбец типа строка размера 12 с ограничением `not null`.

В следующем примере показано, как можно использовать эти параметры.

Например, если мы хотим перенести модуль `forum`, чьи файлы миграций расположены в каталоге `migrations` данного модуля, для этого мы можем использовать следующую команду:

```
# не интерактивная миграция модуля форума
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Глобальная настройка команд

Вместо того, чтобы каждый раз вводить одинаковые значения параметров миграции, когда вы запускаете команду миграции, можно настроить её раз и навсегда в конфигурации приложения, как показано ниже:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

С приведённой выше конфигурацией, каждый раз при запуске команды миграции, таблица `backend_migration` будет использована для записи истории миграций. И Вам больше не нужно указывать её через параметр `migrationTable` в командной строке.

6.4.9 Миграции в Несколько Баз Данных

По умолчанию, миграции применяются для базы данных, указанной в `db` компоненте приложения. Если Вы хотите применить миграцию к другой базе данных, Вы можете определить параметр `db` в командной строке как показано ниже,

```
yii migrate --db=db2
```

Приведенная выше команда применит миграции к базе данных `db2`.

Иногда может случиться так, что Вы захотите применить *некоторые* из миграций к одной базе данных, а некоторые другие к другой базе данных. Для достижения этой цели, при реализации класса миграции, необходимо явно указать идентификатор ID компонента базы данных, который миграция будет использовать, следующим образом:

```
<?php
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

Вышеуказанная миграция будет применена к `db2` даже если указать другую базу данных через параметр `db` командной строки. Обратите внимание, что история миграций в этом случае будет записана в базу данных, указанную в параметре `db` командной строки.

Если у вас есть несколько миграций, которые используют ту же другую базу данных, то рекомендуется создать базовый класс миграций выше кода `init()`. Затем каждый класс миграции может расширяться от этого базового класса.

Совет: Кроме установки свойства `db`, Вы также можете работать с разными базами данных путем создания нового соединения с конкретной базой данных в классе Вашей миграции. Можно использовать [DAO методы](#) с этими соединениями для манипулирования различными базами данных.

Другая стратегия, которую Вы можете выбрать, чтобы перенести (мигрировать) несколько баз данных - это сохранить миграции различных баз данных в разные директории. Затем вы можете перенести эти базы данных в нужные базы следующими командами:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
```

```
yii migrate --migrationPath=@app/migrations/db2 --db=db2  
...
```

Первая команда применит миграции в директории `@app/migrations/db1` к базе данных `db1`, а вторая команда применит миграции в директории `@app/migrations/db2` к базе данных `db2` и так далее.

Error: not existing file: <https://github.com/yiisoft/yii2-sphinx/blob/master/docs/guide/ru>

Error: not existing file: <https://github.com/yiisoft/yii2-redis/blob/master/docs/g>

Error: not existing file: <https://github.com/yiisoft/yii2-mongodb/blob/master/docs/guide>

Error: not existing file: <https://github.com/yiisoft/yii2-elasticsearch/blob/master>

Глава 7

Получение данных от пользователя

7.1 Создание форм

Основной способ использования форм в Yii является использование `yii\widgets\ActiveForm`. Этот подход должен быть применён, когда форма основана на модели. Кроме того, имеются дополнительные методы в `yii\helpers\Html`, которые используются для добавления кнопок и текстовых подсказок к любой форме.

Форма, которая отображается на стороне клиента, в большинстве случаев, соответствует [модели](#). Модель, в свою очередь, проверяет данные из элементов формы на сервере (смотрите раздел [Валидация](#) для более подробных сведений). Когда создаётся форма, основанная на модели, необходимо определить, что же является моделью. Модель может основываться на классе [Active Record](#), который описывает некоторые данные из базы данных, или модель может основываться на базовом классе `Model` (происходит от `yii\base\Model`), который позволяет использовать произвольный набор элементов формы, например, форма входа.

В следующем примере показано, как создать модель формы, основанной на базовом классе `Model`:

```
<?php

class LoginForm extends \yii\base\Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // тут определяются правила валидации
        ];
    }
}
```

```
}
}
```

В контроллере будем передавать экземпляр этой модели в представление для виджета `ActiveForm`, который генерирует форму.

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

$form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
]);
<?=$form->field($model, 'username') ?>
<?=$form->field($model, 'password')->passwordInput() ?>

<div class="form-group">
    <div class="col-lg-offset-1 col-lg-11">
        <?=$form->field($model, 'password')->passwordInput() ?>
    </div>
</div>
<?php ActiveForm::end() ?>
```

В вышеизложенном коде, `ActiveForm::begin()` не только создаёт экземпляр формы, но также и знаменует её начало. Весь контент, расположенный между `ActiveForm::begin()` и `ActiveForm::end()`, будет завернут в HTML `<form>` тег. Вы можете настроить некоторые настройки виджета через передачу массива в его `begin` метод, также как и в любом другом виджете. В этом случае, дополнительный CSS класс и идентификатор ID будет прикреплен к открывающемуся тегу `<form>`. Для просмотра всех доступных настроек, пожалуйста обратитесь к API документации `yii\widgets\ActiveForm`.

Для создания в форме элемента с меткой и любой применимой Javascript валидацией, вызывается `ActiveForm::field()`, который возвращает экземпляр `yii\widgets\ActiveField`. Когда этот метод вызывается непосредственно, то результатом будет текстовый элемент (`input type="text"`). Для того, чтобы настроить элемент, можно вызвать один за одним дополнительные методы `ActiveField`:

```
// элемент формы password
<?=$form->field($model, 'password')->passwordInput() ?>
// добавлена подсказка hint и настроена метка label
<?=$form->field($model, 'username')->textInput()->hint('Пожалуйста,
    введите имя')->label('Имя') ?>
// создание HTML5 email элемента
<?=$form->field($model, 'email')->input('email') ?>
```

Впоследствии будут созданы `<label>`, `<input>` и другие теги в соответствии с `template`, который определен в элементе. Имя элемента формы

определяется автоматически из моделей `form name` и их атрибутов. Например, имя элемента для атрибута `username` в коде, приведённом выше, будет `LoginForm[username]`. Это правило именования будет учитываться на стороне сервера при получении массива результатов `$_POST['LoginForm']` для всех элементов формы входа (Login Form).

Подсказка: Если в форме только одна модель и вы хотите упростить имена полей ввода, то можете сделать это перекрыв метод `formName()` модели так, чтобы он возвращал пустую строку. Это может пригодиться для получения более красивых URL при фильтрации моделей в `GridView`.

Специфический атрибут модели может быть задан через более сложный способ. Например, при загрузке файлов или выборе нескольких значений из списка, в качестве значений атрибуту модели нужно передать массив, для этого к имени можно добавить []:

```
// поддерживает загрузку нескольких файлов:
echo $form->field($model, 'uploadFile[]')->fileInput(['multiple'=>'multiple'
]);

// поддерживает выбор нескольких значений:
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' =>
'Item B', 'c' => 'Item C']);
```

Имена элементов форм следует выбирать, учитывая, что могут возникнуть конфликты. Подробнее об этом в документации jQuery¹:

Имена и идентификаторы форм и их элементов не должны совпадать с элементами форм, такими как `submit`, `length`, или `method`. Конфликты имен могут вызывать трудно диагностируемые ошибки. Подробнее о способах избегания подобных проблем смотрите DOMLint².

Дополнительные HTML элементы могут быть добавлены к форме используя обычный HTML или методы из класса помощника `Html`, как это было сделано с помощью `Html::submitButton()` в примере, что выше.

Подсказка: Если вы используете Twitter Bootstrap CSS в своём приложении, то воспользуйтесь `yii\bootstrap\ActiveForm` вместо `yii\widgets\ActiveForm`. Он добавит к `ActiveForm` дополнительные стили, которые сработают в рамках bootstrap CSS.

Подсказка: для добавления “звёздочки” к обязательным элементам формы воспользуйтесь следующим CSS:

¹<https://api.jquery.com/submit/>

²<http://kangax.github.io/domlint/>

```
div.required label.control-label:after {  
    content: " *";  
    color: red;  
}
```

7.1.1 Создание выпадающего списка

Для создания выпадающего списка можно использовать метод ActiveForm `dropDownList()`:

```
use app\models\ProductCategory;  
  
/* @var $this yii\web\View */  
/* @var $form yii\widgets\ActiveForm */  
/* @var $model app\models\Product */  
  
echo $form->field($model, 'product_category')->dropDownList(  
    ProductCategory::find()->select(['category_name', 'id'])->indexBy('id')->  
    column(),  
    ['prompt'=>'Select Category']  
);
```

Текущее значение поля модели будет автоматически выбрано в списке.

7.1.2 Работа с Pjax

Виджет Pjax позволяет обновлять определённую область страницы вместо перезагрузки всей страницы. Вы можете использовать его для обновления формы после её отсылки.

Для того, чтобы задать, какая из форм будет работать через PJAX, можно воспользоваться опцией `$formSelector`. Если значение не задано, все формы с атрибутом `data-pjax` внутри PJAX-контента будут работать через PJAX.

```
use yii\widgets\Pjax;  
use yii\widgets\ActiveForm;  
  
Pjax::begin([  
    // Pjax options  
]);  
  
$form = ActiveForm::begin([  
    'options' => ['data' => ['pjax' => true]],  
    // остальные опции ActiveForm  
]);  
  
    // Содержимое ActiveForm  
  
ActiveForm::end();  
Pjax::end();
```

Подсказка: Будьте осторожны с ссылками внутри виджета Pjax так как ответ будет также отображаться внутри виджета. Чтобы ссылка работала без PjAX, добавьте к ней HTML-атрибут `data-pjax="0"`.

Значения кнопок submit и загрузка файлов В `jQuery.serializeArray()` имеются определённые проблемы при работе с файлами³ и значениями кнопом типа submit⁴. Они не будут исправлены и признаны устаревшими в пользу класса `FormData` из HTML5.

Это означает, что поддержка файлов и значений submit-кнопок через AJAX или виджет Pjax зависит от поддержки в браузере⁵ класса `FormData`.

7.1.3 Еще по теме

Следующая глава [Валидация](#) описывает валидацию отправленной формы как на стороне сервера, так и на стороне клиента.

Если вы хотите более подробно изучить информацию по использованию форм, то обратитесь к главам:

- [Табличный ввод](#) - получение данных нескольких моделей одного вида.
- [Работа с несколькими моделями](#) - обработка нескольких разных моделей в рамках одной формы.
- [Загрузка файлов](#) - использование форм для загрузки файлов.

7.2 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;  
  
// заполняем модель пользовательскими данными  
$model->load(\Yii::$app->request->post());
```

³<https://github.com/jquery/jquery/issues/2321>

⁴<https://github.com/jquery/jquery/issues/2321>

⁵https://developer.mozilla.org/en-US/docs/Web/API/FormData#Browser_compatibility

```
// аналогично следующей строке:  
// $model->attributes = \Yii::$app->request->post('ContactForm');  
  
if ($model->validate()) {  
    // все данные корректны  
} else {  
    // данные не корректны: $errors - массив содержащий сообщения об ошибках  
    $errors = $model->errors;  
}
```

7.2.1 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 `rules()` должен 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 каком 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 `scenario`. Эти атрибуты называются - *активными атрибутами*.
2. Определяет, какие правила проверки должны использоваться, получив список правил от `yii\base\Model::rules()` используя текущий `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, объявленным в `scenarios()` и связан с одним или несколькими активными правилами, объявленными в `rules()`.

Примечание: Правилам валидации полезно давать имена. Например:

```
public function rules()
{
    return [
        // ...
        'password' => [['password'], 'string', 'max' => 60],
    ];
}
```

В случае наследования предыдущей модели, именованные правила можно модифицировать или удалить:

```
public function rules()
{
    $rules = parent::rules();
    unset($rules['password']);
    return $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, **required** валидатор добавил к модели сообщение об ошибке “Имя пользователя не может быть пустым.” когда атрибут **username** не удовлетворил правилу 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, **number** валидатор поддерживает **tooBig** и **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 `when` свойство, чтобы определить такие условия. Например:

```
[ 'state', 'required', 'when' => function($model) {
    return $model->country == 'USA';
}],
```

Это свойство `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 `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:

```
return [
    [['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 преобразование 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:

```
return [
    // 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.

7.2.2 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 `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 `hasErrors()` и затем получить ошибки проверки вызвав метод `errors` как это делают нормальные модели. Вы можете также получить доступ к динамическим атрибутам, определенным через экземпляр модели, например, `$model->name` и `$model->email`.

7.2.3 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 получают пустые входные данные, или если они уже не смогли пройти некоторые правила валидации. Если вы хотите, чтобы, это правило применялось всегда, вы можете настроить свойства `skipOnEmpty` и/или `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()`.

7.2.4 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 клиента 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 predefined 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, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
```



```
}

```

Подсказка: 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 objects⁶. 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` предусмотренная 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 выполняется асинхронный вызов к серверу. В следующем примере показано, как проверить размеры загружаемого файла изображения на стороне клиента.

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Изображение слишком широкое!');
            }
            def.resolve();
        }
    JS;
}
```

⁶<http://api.jquery.com/category/deferred-object/>

```

    }
    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 проверки 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 АЈАХ в этом случае. Это вызовет АЈАХ-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 одного поля, Вы должны свойство `enableAjaxValidation` выбрать как `true` и указать уникальный `id` формы:

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

Чтобы включить AJAX-валидацию для всей формы, Вы должны свойство `enableAjaxValidation` выбрать как `true` для формы:

```
$form = yii\widgets\ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

Примечание: В случае, если свойство `enableAjaxValidation` указано и у поля и у формы, первый вариант будет иметь приоритет.

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, описанная здесь более интегрированная и требует меньше усилий к написанию кода.

7.3 Загрузка файлов

Загрузка файлов в Yii, обычно, выполняется при помощи класса `yii\web\UploadedFile`, который представляет каждый загруженный файл в виде объекта `UploadedFile`. Используя `yii\widgets\ActiveForm` и модели можно легко создать безопасный механизм загрузки файлов.

7.3.1 Создание моделей

Как и в случае с обработкой текстового ввода, для загрузки файла можно создать класс модели и использовать его атрибут для хранения экземпляра объекта `UploadedFile`, содержащего параметры загруженного файла. Так же, возможно использование правил валидации модели для проверки загруженного файла. Например,

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
                'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
                . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

В примере выше атрибут `imageFile` используется для хранения экземпляра загруженного файла. Правило валидации `file`, которое, при помощи валидатора `yii\validators\FileValidator`, проверяет расширение загруженного файла на соответствие с `png` или `jpg`. Метод `upload()` выпол-

няет валидацию и сохраняет загруженный файл на сервере.

Валидатор `file` позволяет проверять расширение, размер, тип MIME и другие параметры загруженного файла. Подробности в разделе [Встроенные валидаторы](#).

Подсказка: При загрузке изображений лучше использовать соответствующий валидатор `image`. Данный валидатор реализован классом `yii\validators\ImageValidator` и позволяет проверить корректность загруженного изображения при помощи расширения `Imagine`⁷.

7.3.2 Представление

Теперь можно создать представление, отображающее поле загрузки файла:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data']]) ?>

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

    <button>Submit</button>

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

Важно помнить, что для корректной загрузки файла, необходим параметр формы `enctype`. Метод `fileInput()` выведет тег `<input type="file">`, позволяющий пользователю выбрать файл для загрузки.

Подсказка: начиная с версии 2.0.8, `yii\web\widgets\ActiveField::fileInput` автоматически добавляет к форме свойство `enctype`, если в ней есть поле для загрузки файла.

7.3.3 Загрузка

Теперь напишем код действия контроллера, который объединит модель и представление.

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;
```

⁷<https://github.com/yiisoft/yii2-imagine>

```
class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model, 'imageFile');
            if ($model->upload()) {
                // file is uploaded successfully
                return;
            }
        }

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

При получении данных, отправленных из формы, для создания из загруженного файла экземпляра объекта `UploadedFile`, вызывается метод `yii\web\UploadedFile::getInstance()`. Далее всю работу по валидации и сохранению загруженного файла на сервере берет на себя модель.

7.3.4 Загрузка нескольких файлов

Для загрузки нескольких файлов достаточно внести в предыдущий код несколько небольших изменений.

Сначала нужно добавить в правило валидации `file` параметр `maxFiles` для ограничения максимального количества загружаемых одновременно файлов. Установка `maxFiles` равным 0 означает снятие ограничений на количество файлов, которые могут быть загружены одновременно. Максимально разрешенное количество одновременно закачиваемых файлов также ограничивается директивой PHP `max_file_uploads`⁸, и по умолчанию равно 20. Метод `upload()` нужно изменить для сохранения загруженных файлов по одному.

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;
```

⁸<http://php.net/manual/ru/ini.core.php#ini.max-file-uploads>

```

public function rules()
{
    return [
        [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
        'png, jpg', 'maxFiles' => 4],
    ];
}

public function upload()
{
    if ($this->validate()) {
        foreach ($this->imageFiles as $file) {
            $file->saveAs('uploads/' . $file->baseName . '.' . $file->
            extension);
        }
        return true;
    } else {
        return false;
    }
}
}

```

В представлении, в вызов метода `fileInput()`, нужно добавить параметр `multiple` для того, чтобы поле *input* позволяло выбирать несколько файлов одновременно:

```

<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-
data']]) ?>

<?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' => true,
'accept' => 'image/*']) ?>

<button>Submit</button>

<?php ActiveForm::end() ?>

```

В действии контроллера нужно заменить вызов `UploadedFile::getInstance()` на `UploadedFile::getInstances()` для присвоения атрибуту модели `imageFiles` массива объектов `UploadedFile`.

```

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()

```

```

{
    $model = new UploadForm();

    if (Yii::$app->request->isPost) {
        $model->imageFiles = UploadedFile::getInstances($model, '
imageFiles');
        if ($model->upload()) {
            // file is uploaded successfully
            return;
        }
    }

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

```

7.4 Табличный ввод

Иногда возникает необходимость обработки нескольких моделей одного вида в одной форме. Например, несколько параметров, каждый из которых сохраняется как пара имя-значение и представляется моделью `Setting` [active record](#). Такой тип форм часто называют “табличным вводом”. Обработка данных нескольких моделей разных видов в одной форме описана в разделе [Работа с несколькими моделями](#).

Дальше будет рассмотрен вариант реализации табличного ввода при помощи Yii.

Выделим три сценария, которые потребуют немного разных подходов:

- Изменение фиксированного набора записей из базы данных;
- Создание произвольного набора записей;
- Изменение, создание и удаление записей на одной странице.

В отличие от форм с одной моделью, рассмотренных ранее, теперь будем иметь дело с массивом моделей. Этот массив передается в представление и для каждой модели отображаются поля ввода в табличном виде. Для загрузки и валидации нескольких моделей сразу будем использовать вспомогательные методы класса `yii\base\Model`:

- `Model::loadMultiple()` загружает данные post в массив моделей;
- `Model::validateMultiple()` валидирует массив моделей.

Изменение фиксированного набора записей

Начнем с действия контроллера:

```

<?php

namespace app\controllers;

```



```
use Yii;
use yii\base\Model;
use yii\web\Controller;
use app\models\Setting;

class SettingsController extends Controller
{
    // ...

    public function actionUpdate()
    {
        $settings = Setting::find()->indexBy('id')->all();

        if (Model::loadMultiple($settings, Yii::$app->request->post()) &&
            Model::validateMultiple($settings)) {
            foreach ($settings as $setting) {
                $setting->save(false);
            }
            return $this->redirect('index');
        }

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

В коде выше, для получения из базы данных массива моделей, индексированного по главному ключу, использован метод `indexBy()`. В дальнейшем будем использовать это для идентификации полей формы. Метод `Model::loadMultiple()` загружает данные запроса POST в массив моделей, а метод `Model::validateMultiple()` проводит валидацию всех моделей. Так, как модели уже прошли валидацию, мы передаем методу `save()` параметр `false` для отключения повторной валидации.

Теперь займемся формой в представлении `update`:

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

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo $form->field($setting, "[$index]value")->label($setting->name);
}

ActiveForm::end();
```

Для каждого элемента массива `$settings` генерируется имя и поле ввода значения. Важно указать правильный индекс в имени поля ввода значения, так как `Model::loadMultiple()` определяет модель по этому индексу.

Создание произвольного набора записей

Процесс создания новых записей похож на их изменение, за исключением части, где создаются новые модели:

```
public function actionCreate()
{
    $count = count(Yii::$app->request->post('Setting', []));
    $settings = [new Setting()];
    for($i = 1; $i < $count; $i++) {
        $settings[] = new Setting();
    }

    // ...
}
```

Сначала создается массив `$settings`, содержащий один экземпляр модели, так что, по умолчанию в представлении всегда будет отображено хотя бы одно поле. Дополнительно, добавляются модели для каждой полученной строки ввода.

В представлении возможно использовать javascript для добавления новых полей динамически.

Изменение, создание и удаление записей на одной странице

Примечание: Раздел находится в разработке

TBD

7.5 Работа с несколькими моделями

Когда имеешь дело со сложными данными, иногда может потребоваться использовать несколько разных моделей для обработки данных, введенных пользователем. Для примера, предположим, что информация пользователя для входа хранится в таблице `user`, а данные профиля хранятся в таблице `profile`, и вы можете захотеть обрабатывать входные данные о пользователе через модели `User` и `Profile`. Учитывая поддержку Yii моделей и форм, вы можете решить данную задачу способом, не сильно отличающимся от обработки одинарной модели.

Далее мы покажем, как можно создать форму, которая позволила бы вам собирать данные для обеих моделей `User` и `Profile`.

Действие контроллера для обработки данных пользователя и данных профиля может быть написано следующим образом,

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
```

```

use yii\web\NotFoundException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        $profile = Profile::findOne($id);

        if (!isset($user, $profile)) {
            throw new NotFoundException("The user was not found.");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) && $profile->load(Yii::$app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}

```

В действии `update`, мы сначала загружаем из базы модели `$user` и `$profile`. Затем мы вызываем метод `yii\base\Model::load()` для заполнения этих двух моделей данными, введенными пользователем. В случае успеха мы проверяем модели и сохраняем их. В противном случае мы рендерим представление `update`, которое содержит следующий контент:

```

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

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>

<?= $form->field($user, 'username') ?>

...other input fields...

```

```
<?= $form->field($profile, 'website') ?>

<?= Html::submitButton('Update', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

Как вы можете видеть, в представлении `update` рендерятся поля ввода для двух моделей `$user` и `$profile`.

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 на подобии `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 в качестве локали будет использован **язык приложения**. Подробнее смотрите в разделе «**интернационализация**». Компонент форматирования будет выбирать корректный формат для даты и чисел в соответствии с локалью, включая имена месяцев и дней недели, переведённые на текущий язык. Форматирование дат также зависит от **часового пояса**, 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. January 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 следующие подсекции.

```
'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 текущим временем в человеко понятном формате, например: `1 час назад`.
- `duration`: значение будет отформатировано как продолжительность в человеко понятном формате, например `1 день, 2 минуты`.

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.

²http://en.wikipedia.org/wiki/Unix_time

Дополнительно вы можете задать специальный формат, используя синтаксис, заданный ICU Project³, который описан в руководстве ICU по следующему адресу: <http://userguide.icu-project.org/formatparse/datetime>. Также вы можете использовать синтаксис, который распознаётся PHP-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
```

Часовые пояса

Для форматирования значений даты и времени Yii будет преобразовывать их в соответствии с настроенным часовым поясом. 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 в часовом поясе 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 часовой пояс для предполагаемых timestamp, которые не включают в себя часовой пояс, как во втором примере в коде выше. Вы можете задать `yii\i18n\Formatter::$defaultTimeZone` часовым поясом, который вы используете для хранения данных.

Примечание: Поскольку часовые пояса являются субъектом ответственности правительств по всему миру и могут часто меняться, это значит, что вы, вероятно, не имеете самую свежую информацию в базе данных часовых поясов, установленной на вашем сервере. Вы можете обратиться к ICU руководству⁵ для получения подробностей об обновлении базы данных часовых поясов. См. также: [Настройка вашего PHP окружения для интернационализации](#).

³<http://site.icu-project.org/>

⁴<http://php.net/manual/ru/function.date.php>

⁵<http://userguide.icu-project.org/datetime/timezone#TOC-Updating-the-Time-Zone-Data>

8.1.3 Форматирование чисел

Для форматирования числовых значений класс форматирования предоставляет следующие методы:

- **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 даёт никакого эффекта, кроме значений `null`, которые будут отформатированы в соответствии с `nullDisplay`.

⁶<http://php.net/manual/ru/class.numberformatter.php>

- **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.
- **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 **провайдер 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 постраничного разделение 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 [провайдер данных](#) с одним из [виджетов данных](#), сортировка будет применена автоматически. В противном случае вы 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` и `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`.

8.4 Провайдеры данных

В разделах [Постраничное разделение данных](#) и [Сортировка](#) было описано, как сделать возможность для конечных пользователей, чтобы они могли выбирать определённую страницу для вывода данных и сортировку их по некоторым колонкам.

Провайдер данных это класс, который реализует `yii\data\DataProviderInterface`. Такая реализация поддерживает в основном разбивку на страницы и сортировку. Они обычно используются для работы [виджетов данных](#), что позволяет конечным пользователям интерактивно использовать сортировку данных и их разбивку на страницы.

В Yii реализованы следующие классы провайдеров данных:

- `yii\data\ActiveDataProvider`: использует `yii\db\Query` или `yii\db\ActiveQuery` для запроса данных из базы данных, возвращая их в виде массива или экземпляров [Active Record](#).

- `yii\data\SqlDataProvider`: выполняет запрос SQL к базе данных и возвращает результат в виде массива.
- `yii\data\ArrayDataProvider`: принимает большой массив и возвращает выборку из него с возможностью сортировки и разбивки на страницы.

Использование всех этих провайдеров данных имеет общую закономерность:

```
// создание провайдера данных с конфигурацией для сортировки и постраничной
// разбивки
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// Получение данных с разбивкой на страницы и сортировкой.
$models = $provider->getModels();

// получение количества данных на текущей странице
$count = $provider->getCount();

// получение общего количества данных на всех страницах
$totalCount = $provider->getTotalCount();
```

Определение поведений сортировки и разбивки для провайдера данных устанавливается через его свойства `pagination` и `sort`, которые соответствуют настройкам `yii\data\Pagination` and `yii\data\Sort`. Вы можете отключить сортировку и разбивку на страницы путём выставления их настроек в `false`.

Виджеты данных, такие как `yii\grid\GridView`, имеют свойство `dataProvider`, которое может принимать экземпляр провайдера данных для отображения его данных. Например:

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

Эти провайдеры данных в некоторой степени различаются по использовании, в зависимости от источника данных. Далее опишем более подробно использование каждого провайдера данных.

8.4.1 ActiveRecord

Для использования `yii\data\ActiveDataProvider`, необходимо настроить его свойство `query`. Оно принимает любой `yii\db\Query` или `yii\db\ActiveQuery` объект. Если использовать первый, то данные будут возвращены в виде массивов, если второй - данные также могут быть возвращены в виде массивов, а также в виде экземпляров [Active Record](#). Например:

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// возвращает массив Post объектов
$posts = $provider->getModels();
```

Если изменить `$query` в этом примере на следующий код, то будут возвращены сырые массивы.

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['status' => 1]);
```

Примечание: Если `query` содержит условия сортировки в `orderBy`, то новые условия, полученные от конечных пользователей (через настройки `sort`) будут добавлены к существующим условиям в `orderBy`. Любые условия в `limit` и `offset` будут переписаны запросом конечного пользователя к различным страницам (через конфигурацию `pagination`).

По умолчанию, `yii\data\ActiveDataProvider` использует компонент приложения `db` для подключения к базе данных. Можно использовать разные базы данных, настроив подключение через конфигурацию свойства `yii\data\ActiveDataProvider::$db`.

8.4.2 SqlDataProvider

`yii\data\SqlDataProvider` работает с сырыми запросами SQL, которые используются для извлечения необходимых данных. Основываясь на спецификации из `sort` и `pagination`, провайдер данных будет добавлять `ORDER BY` и `LIMIT` конструкции к SQL запросу, для возврата только запрошенной страницы данных с учётом определённой сортировки.

Для использования `yii\data\SqlDataProvider`, необходимо настроить свойства `sql` и `totalCount`. Например:

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// возвращает массив данных
$models = $provider->getModels();
```

Совет: Свойство `totalCount` обязательно только тогда, когда вам нужна разбивка на страницы. Всё потому, что запрос SQL `sql` будет изменяться провайдером данных для возврата только текущей запрошенной страницы. Провайдеру необходимо знать общее количество данных в запросе для корректного вычисления разбивки на доступные страницы.

8.4.3 ArrayDataProvider

`yii\data\ArrayDataProvider` лучше использовать для работы с большим массивом. Этот провайдер помогает вернуть выборку из большого массива с сортировкой по одному или нескольким колонкам. Для использования `yii\data\ArrayDataProvider` необходимо определить свойство `allModels`, как большой массив. Элементы в большом массиве могут быть ассоциативными массивами (например результаты выборки из DAO) или объекты ([Active Record](#) экземпляры). Например:

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];
```

```
$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
    ],
]);

// получает строки для текущей запрошенной странице
$rows = $provider->getModels();
```

Примечание: Сравнивая с Active Data Provider и SQL Data Provider, ArrayDataProvider менее эффективный потому, что требует загрузки *всех* данных в память.

8.4.4 Принципы работы с ключами данных

При возврате данных с помощью провайдера, часто требуется идентификация каждого элемента по уникальному ключу. Например, если данные - это какая-то информация по клиенту, то возможно понадобится использовать ID клиента, как ключ для данных по каждому клиенту. Провайдер данных через `yii\data\DataProviderInterface::getModels()` может вернуть список из ключей и соответствующего набора данных. Например,

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
]);

// возвращает массив объектов Post
$posts = $provider->getModels();

// возвращает значения первичного ключа в соответствии с $posts
$ids = $provider->getKeys();
```

В вышеописанном примере, так как `yii\data\ActiveDataProvider` предоставляется один `yii\db\ActiveQuery` объект, то в этом случае провайдер достаточно умен, чтобы вернуть значения первичных ключей в качестве идентификатора. Также есть возможность настроить способ вычисления значения идентификатора, через настройку `yii\data\ActiveDataProvider::$key`, как имя колонки или функцию вычисления значений ключа. Например:

```
// в качестве ключа используется столбец "slug"
```



```
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// в качестве ключа используется md5(id)
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.4.5 Создание своего провайдера данных

Для создания своих классов провайдера данных, необходимо реализовать `yii\data\DataProviderInterface`. Простой способ сделать это - наследовать `yii\data\BaseDataProvider`, который помогает сфокусироваться на логике ядра провайдера данных. В основном необходимо реализовать следующие методы:

- `prepareModels()`: подготавливает модели данных, которые будут доступны в текущей странице и возвращает их в виде массива.
- `prepareKeys()`: принимает массив имеющихся в настоящее время моделей данных и возвращает ключи, связанные с ними.
- `prepareTotalCount`: возвращает значение, указывающее общее количество моделей данных в провайдере данных.

Ниже приведён пример провайдера данных, который эффективно считывает данные из CSV:

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string name of the CSV file to read
     */
    public $filename;

    /**
     * @var string|callable name of the key column or a callable returning it
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject; // SplFileObject is very convenient for seeking
    to particular line in a file
}
```

```

/**
 * @inheritdoc
 */
public function init()
{
    parent::init();

    // open file
    $this->fileObject = new SplFileObject($this->filename);
}

/**
 * @inheritdoc
 */
protected function prepareModels()
{
    $models = [];
    $pagination = $this->getPagination();

    if ($pagination === false) {
        // in case there's no pagination, read all lines
        while (!$this->fileObject->eof()) {
            $models[] = $this->fileObject->fgetcsv();
            $this->fileObject->next();
        }
    } else {
        // in case there's pagination, read only a single page
        $pagination->totalCount = $this->getTotalCount();
        $this->fileObject->seek($pagination->getOffset());
        $limit = $pagination->getLimit();

        for ($count = 0; $count < $limit; ++$count) {
            $models[] = $this->fileObject->fgetcsv();
            $this->fileObject->next();
        }
    }

    return $models;
}

/**
 * @inheritdoc
 */
protected function prepareKeys($models)
{
    if ($this->key !== null) {
        $keys = [];

        foreach ($models as $model) {
            if (is_string($this->key)) {
                $keys[] = $model[$this->key];
            } else {

```

```

        $keys[] = call_user_func($this->key, $model);
    }

    return $keys;
} else {
    return array_keys($models);
}
}

/**
 * @inheritdoc
 */
protected function prepareTotalCount()
{
    $count = 0;

    while (!$this->fileObject->eof()) {
        $this->fileObject->next();
        ++$count;
    }

    return $count;
}
}

```

8.5 Виджеты для данных

Yii предоставляет набор **виджетов**, которые могут быть использованы для отображения данных. В то время как виджет `DetailView` может быть использован для отображения данных по одной записи, то виджеты `ListView` и `GridView` могут быть использованы для показа данных в виде списка или таблицы с возможностью сортировки, фильтрации и разбивки данных постранично.

8.5.1 DetailView

Виджет `DetailView` отображает детали по данным для одной `model`.

Этот виджет лучше использовать для отображения данных модели в обычном формате (т.е. каждый атрибут модели будет представлен в виде строки в таблице). Модель может быть либо объектом класса `yii\base\Model` или его наследником, таких как `active record`, либо ассоциативным массивом.

`DetailView` использует свойство `$attributes` для определений, какие атрибуты модели должны быть показаны и в каком формате. Обратитесь к разделу [Форматирование данных](#) за возможными настройками форматирования.

Обычное использование `DetailView` сводится к следующему коду:

```

echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title', // title свойство обычный( текст)
        'description:html', // description свойство, как HTML
        [ // name свойство зависимой модели owner
            'label' => 'Owner',
            'value' => $model->owner->name,
        ],
        'created_at:datetime', // дата создания в формате datetime
    ],
]);

```

8.5.2 ListView

Виджет `ListView` использует для отображения информации [провайдер данных](#). Каждая модель отображается, используя определённый вид. Поскольку провайдер включает в себя разбивку на страницы, сортировку и фильтрацию, то его использование удобно для отображения информации конечному пользователю и создания интерфейса управления данными.

Обычное использование сводится к следующему коду:

```

use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);

```

`_post` файл вид, который может содержать следующее:

```

<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="post">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>

```

В вышеописанном коде текущая модель доступна как `$model`. Кроме этого доступны дополнительные переменные:

- `$key`: mixed, значение ключа в соответствии с данными.

- `$index`: integer, индекс элемента данных в массиве элементов, возвращенных поставщику данных, который начинается с 0.
- `$widget`: `ListView`, это экземпляр виджета.

Если необходимо послать дополнительные данные в каждый вид, то можно использовать свойство `$viewParams` как ключ-значение, например:

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

Они также станут доступны в виде в качестве переменных.

8.5.3 GridView

Таблица данных или `GridView` - это один из сверхмощных Yii виджетов. Он может быть полезен, если необходимо быстро создать административный раздел системы. `GridView` использует данные, как [провайдер данных](#) и отображает каждую строку используя `columns` для предоставления данных в таблице.

Каждая строка из таблицы представлена данными из одиночной записи и колонка, как правило, представляет собой атрибут записи (некоторые столбцы могут соответствовать сложным выражениям атрибутов или статическому тексту).

Минимальный код, который необходим для использования `GridView`:

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

В вышеприведённом коде сначала создаётся провайдер данных и затем используется `GridView` для отображения атрибутов для каждого элемента из провайдера данных. Отображенная таблица оснащена функционалом сортировки и разбивки на страницы из коробки.

Колонки таблицы

Колонки таблицы настраиваются с помощью определённых `yii\grid\Column` классов, которые настраиваются в свойстве `columns` виджета `GridView`. В зависимости от типа колонки и их настроек, данные отображаются по разному. По умолчанию это класс `yii\grid\DataColumn`, который представляет атрибут модели с возможностью сортировки и фильтрации по нему.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // Обычные поля определенные данными содержащимися в $dataProvider.
        // Будут использованы данные из полей модели.
        'id',
        'username',
        // Более сложный пример.
        [
            'class' => 'yii\grid\DataColumn', // может быть опущено,
            // поскольку является значением по умолчанию
            'value' => function ($data) {
                return $data->name; // $data['name'] для массивов, например,
                // при использовании SqlDataProvider.
            },
        ],
    ],
]);
```

Учтите, что если `columns` не сконфигурирована, то Yii попытается отобразить все возможные колонки из провайдера данных.

Классы колонок

Колонки таблицы могут быть настроены, используя различные классы колонок:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- тут
            // тут можно настроить дополнительные свойства
        ],
    ],
]);
```

В дополнение к классам колонок от Yii, вы можете самостоятельно создать свой собственный класс.

Каждый класс колонки наследуется от `yii\grid\Column`, так что есть некоторые общие параметры, которые можно установить при настройке колонок.

- **header** позволяет установить содержание для строки заголовка.
- **footer** позволяет установить содержание для “подвала”.

- **visible** определяет, должен ли столбец быть видимым.
- **content** позволяет передавать действительный обратный вызов, который будет возвращать данные для строки. Формат следующий:

```
function ($model, $key, $index, $column) {
    return 'a string';
}
```

Вы можете задать различные параметры контейнера HTML через массивы:

- **headerOptions**
- **footerOptions**
- **filterOptions**
- **contentOptions**

DataColumn Data column используется для отображения и сортировки данных. По умолчанию этот тип используется для всех колонок.

Основная настройка этой колонки - это свойство **format**. Значение этого свойства посылается в методы **formatter** компонента, который по умолчанию **Formatter**

```
echo GridView::widget([
    'columns' => [
        [
            'attribute' => 'name',
            'format' => 'text'
        ],
        [
            'attribute' => 'birthday',
            'format' => ['date', 'php:Y-m-d']
        ],
    ],
]);
```

В вышеприведённом коде **text** соответствует `yii\i18n\Formatter::asText()`. В качестве первого аргумента для этого метода будет передаваться значение колонки. Во второй колонки описано **date**, которая соответствует `yii\i18n\Formatter::asDate()`. В качестве первого аргумента, опять же, будет передаваться значение колонки, в то время как второй аргумент будет `'php:Y-m-d'`.

Доступный список форматов смотрите в разделе [Форматирование данных](#).

Для конфигурации колонок данных также доступен короткий вид записи, который описан в API документации для **колонок**.

ActionColumn ActionColumn отображает кнопки действия, такие как изменение или удаление для каждой строки.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
```

```
'columns' => [
    [
        'class' => 'yii\grid\ActionColumn',
        // вы можете настроить дополнительные свойства здесь.
    ],
]
```

Доступные свойства для конфигурации:

- **controller** это идентификатор контроллера, который должен обрабатывать действия. Если не установлен, то будет использоваться текущий активный контроллер.
- **template** определяет шаблон для каждой ячейки в колонке действия. Маркеры заключённые в фигурные скобки являются ID действием контроллера (также называются *именами кнопок* в контексте колонки действия). Они могут быть заменены, через свойство **buttons**. Например, маркер {view} будет заменён результатом из функции, определённой в `buttons['view']`. Если такая функция не может быть найдена, то маркер заменяется на пустую строку. По умолчанию шаблон имеет вид {view} {update} {delete}.
- **buttons** массив из функций для отображения кнопок. Ключи массива представлены как имена кнопок (как описывалось выше), а значения представлены в качестве анонимных функций, которые выводят кнопки. Замыкания должны использоваться в следующем виде:

```
function ($url, $model, $key) {
    // возвращаем HTML код для кнопки
}
```

где, `$url` - это URL, который будет повешен на как ссылка на кнопку, `$model` - это объект модели для текущей строки и `$key` - это ключ для модели из провайдера данных.

- **urlCreator** замыкание, которое создаёт URL используя информацию из модели. Вид замыкания должен быть таким же как и в `yii\grid\ActionColumn::createUrl()`. Если свойство не задано, то URL для кнопки будет создана используя метод `yii\grid\ActionColumn::createUrl()`.
- **visibleButtons** это массив условий видимости каждой из кнопок. Ключи массива представлены как имена кнопок (как описывалось выше), а значения представлены как булево значение или анонимная функция. Если имя кнопки не описано в массиве, она будет отображена по умолчанию. Замыкания должны использоваться в следующем виде:

```
'php function ($model, $key, $index) { return $model->status === 'editable';
// отображать ли кнопку } '
```

Или вы можете передать булево значение:

```
'php [
```

```
'update' => \Yii::$app->user->can('update')
```


] ,

CheckboxColumn `Checkbox column` отображает колонку как флаг (checkbox).

Для добавления `CheckboxColumn` в виджет `GridView`, необходимо добавить его в `columns`:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        // ...
        [
            'class' => 'yii\grid\CheckboxColumn',
            // вы можете настроить дополнительные свойства здесь.
        ],
    ],
],
```

Пользователи могут нажимать на флаги для выделения строк в таблице. Отмеченные строки могут быть обработаны с помощью JavaScript кода:

```
var keys = $('#grid').yiiGridView('getSelectedRows');
// массив ключей для отмеченных строк
```

SerialColumn `Serial column` выводит в строках номера начиная с 1 и увеличивая их по мере вывода строк.

Использование очень простое :

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'], // <-- тут
        // ...
    ],
],
```

Сортировка данных

Примечание: Эта секция под разработкой

- <https://github.com/yiisoft/yii2/issues/1576>

Фильтрация данных

Для фильтрации данных в `GridView` необходима **модель**, которая описывает форму для фильтрации, внося условия в запрос поиска для провайдера данных. Общепринятой практикой считается использование **active records** и создание для неё класса модели для поиска, которая содержит необходимую функциональность (может быть сгенерирована через **Gii**). Класс модели для поиска должен описывать правила валидации и реализовать метод `search()`, который будет возвращать провайдер данных.

Для поиска возможных `Post` моделей, можно создать `PostSearch` наподобие следующего примера:

```
<?php

namespace app\models;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;

class PostSearch extends Post
{
    public function rules()
    {
        // только поля определенные в rules() будут доступны для поиска
        return [
            [['id'], 'integer'],
            [['title', 'creation_date'], 'safe'],
        ];
    }

    public function scenarios()
    {
        // bypass scenarios() implementation in the parent class
        return Model::scenarios();
    }

    public function search($params)
    {
        $query = Post::find();

        $dataProvider = new ActiveDataProvider([
            'query' => $query,
        ]);

        // загружаем данные формы поиска и производим валидацию
        if (!$this->load($params) && $this->validate()) {
            return $dataProvider;
        }

        // изменяем запрос добавляя в его фильтрацию
        $query->andWhere(['id' => $this->id]);
        $query->andWhere(['like', 'title', $this->title])
            ->andWhere(['like', 'creation_date', $this->
creation_date]);

        return $dataProvider;
    }
}
```

Теперь можно использовать этот метод в контроллере, чтобы получить провайдер данных для GridView:

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search(Yii::$app->request->get());
```

```
return $this->render('myview', [
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
]);
```

и в виде присвоить их `$dataProvider` и `$searchModel` в виджете `GridView`:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        // ...
    ],
]);
```

Отдельная форма фильтрации

Фильтров в шапке `GridView` достаточно для большинства задач, но добавление отдельной формы фильтрации не представляет особой сложности. Она бывает полезна в случае необходимости фильтрации по полям, которые не отображаются в `GridView` или особых условий фильтрации, например по диапазону дат.

Создайте частичное представление `_search.php` со следующим содержанием:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

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

<div class="post-search">
    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

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

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

    <div class="form-group">
        <?= Html::submitButton('Искать', ['class' => 'btn btn-primary']) ?>
        <?= Html::resetButton('Сбросить', ['class' => 'btn btn-default']) ?>
    </div>

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

и добавьте его отображение в `index.php` таким образом:

```
<?= $this->render('_search', ['model' => $searchModel]) ?>
```

Примечание: если вы используете Gii для генерации CRUD кода, отдельная форма фильтрации (`_search.php`) генерируется по умолчанию, но закомментирована в представлении `index.php`. Вам остается только раскомментировать эту строку и форма готова к использованию!

Для фильтра по диапазону дат мы можем добавить дополнительные атрибуты `createdFrom` и `createdTo` в поисковую модель (их нет в соответствующей таблице модели):

```
class PostSearch extends Post
{
    /**
     * @var string
     */
    public $createdFrom;

    /**
     * @var string
     */
    public $createdTo;
}
```

Расширим условия запроса в методе `search()`:

```
$query->andWhere(['>=', 'creation_date', $this->createdFrom])
->andWhere(['<=', 'creation_date', $this->createdTo]);
```

И добавим соответствующие поля в форму фильтрации:

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

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

Отображение зависимых моделей

Бывают случаи, когда необходимо в GridView вывести в колонке значения из зависимой модели для active records, например имя автора новости, вместо его `id`. Для этого необходимо задать `yii\grid\GridView::$columns` как `author.name`, если же модель `Post` содержит зависимость с именем `author` и имя автора хранится в атрибуте `name`. GridView отобразит имя автора, но вот сортировка и фильтрации по этому полю будет не доступна. Необходимо дополнить некоторый функционал в `PostSearch` модель, которая была упомянута в предыдущем разделе.

Для включения сортировки по зависимой колонки необходимо присоединить зависимую таблицу и добавить правило в компонент Sort для провайдера данных.:

```
$query = Post::find();
$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);

// присоединяем зависимость 'author' которая является связью с таблицей '
// и устанавливаем алиас таблицы в значение 'author'
$query->joinWith(['author' => function($query) { $query->from(['author' => '
    'users']); }]);
// добавляем сортировку по колонке из зависимости
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...
```

Фильтрации также необходим вызов `joinWith`, как описано выше. Также необходимо определить для поиска столбец в атрибутах и правилах:

```
public function attributes()
{
    // делаем поле зависимости доступным для поиска
    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],
        [['title', 'creation_date', 'author.name'], 'safe'],
    ];
}
```

В `search()` просто добавляется другое условие фильтрации:

```
$query->andWhere(['LIKE', 'author.name', $this->getAttribute('author.
name')]);
```

Информация: В коде, что выше, использует такая же строка, как и имя зависимости и псевдонима таблицы. Однако, когда ваш псевдоним и имя связи различаются, вы должны обратить внимание, где вы используете псевдоним, а где имя связи. Простым правилом для этого является использование псевдонима в каждом месте, которое используется для построения запроса к базе данных, и имя связи во всех других определениях, таких как `attributes()`, `rules()` и т.д.

Например, если вы используете псевдоним `au` для связи с таблицей автора, то `joinWith` будет выглядеть так:

```
$query->joinWith(['author' => function($query) { $query->from(['
    au' => 'users']); }]);
```

Это также возможно вызвать как `$query->joinWith(['author'])` ;, когда псевдоним определен в определении отношения.

Псевдоним должен быть использован в состоянии фильтра, но имя атрибута остается неизменным:

```
$query->andFilterWhere(['LIKE', 'au.name', $this->getAttribute('author.name')]);
```

То же самое верно и для определения сортировки:

```
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['au.name' => SORT_ASC],
    'desc' => ['au.name' => SORT_DESC],
];
```

Кроме того, при определении `defaultOrder` для сортировки необходимо использовать имя зависимости вместо псевдонима:

```
$dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
```

Информация: Для подробной информации по `joinWith` и запросам, выполняемым в фоновом режиме, обратитесь к [active record](#) документации.

Использование SQL видов для вывода данных, их сортировки и фильтрации. Существует и другой подход, который быстрее и более удобен - SQL виды. Например, если необходимо показать таблицу из пользователей и их профилей, то можно выбрать такой путь:

```
CREATE OR REPLACE VIEW vw_user_info AS
SELECT user.*, user_profile.lastname, user_profile.firstname
FROM user, user_profile
WHERE user.id = user_profile.user_id
```

Теперь необходимо создать `ActiveRecord`, которая будет отображение данных из этого вида:

```
namespace app\models\views\grid;

use yii\db\ActiveRecord;

class UserView extends ActiveRecord
{
    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return 'vw_user_info';
    }
}
```

```
}

public static function primaryKey()
{
    return ['id'];
}

/**
 * @inheritdoc
 */
public function rules()
{
    return [
        // здесь определяйте ваши правила
    ];
}

/**
 * @inheritdoc
 */
public static function attributeLabels()
{
    return [
        // здесь определяйте ваши метки атрибутов
    ];
}

}
```

После этого вы можете использовать `UIView` в модели поиска, без каких-либо дополнительных условий по сортировке и фильтрации. Все атрибуты будут работать из коробки. Но такая реализация имеет свои плюсы и минусы:

- вам не надо определять условия сортировок и фильтров. Всё работает из коробки;
- это намного быстрее данных, так как некоторые запросы уже выполнены (т.е. для каждой зависимости не нужно выполнять дополнительные запросы)
- поскольку это простое отображение данных из `sql` вида, то в модели будет отсутствовать некоторая доменная логика, например такие методы как `isActive`, `isDeleted`, необходимо продублировать в классе, который описывает вид.

Несколько `GridView` на одной странице

Вы можете использовать больше одной `GridView` на одной странице. Для этого нужно внести некоторые дополнительные настройки для того, чтобы они друг другу не мешали. При использовании нескольких экземпляров `GridView` вы должны настроить различные имена параметров для

сортировки и ссылки для разбиения на страницы так, чтобы каждый GridView имел свою индивидуальную сортировку и разбиение на страницы. Сделать это возможно через настройку `sortParam` и `pageParam` свойств провайдеров данных `sort` и `pagination`

Допустим мы хотим список моделей `Post` и `User`, для которых мы уже подготовили провайдеры данных `$userProvider` и `$postProvider`, тогда код будет выглядеть следующим образом:

```
use yii\grid\GridView;

$userProvider->pagination->pageParam = 'user-page';
$userProvider->sort->sortParam = 'user-sort';

$postProvider->pagination->pageParam = 'post-page';
$postProvider->sort->sortParam = 'post-sort';

echo '<h1>Users</h1>';
echo GridView::widget([
    'dataProvider' => $userProvider,
]);

echo '<h1>Posts</h1>';
echo GridView::widget([
    'dataProvider' => $postProvider,
]);
```

Использование GridView с Pjax

Примечание: Секция находится в стадии разработки

TBD

Error: not existing file: output-client-scripts.md

8.6 Темизация

Темизация — это способ заменить один набор представлений другим без переписывания кода, что замечательно подходит для изменения внешнего вида приложения.

Для того, чтобы начать использовать темизацию, настройте свойство `theme` компонента приложения `view`. Конфигурация настраивает объект `yii\base\Theme`, который отвечает за то, как именно заменяются файлы отображений. Главным образом, стоит настроить следующие свойства `yii\base\Theme`:

- `yii\base\Theme::$basePath`: базовая директория, в которой размещены темизированные ресурсы (CSS, JS, изображения, и так далее).
- `yii\base\Theme::$baseUrl`: базовый URL для доступа к темизированным ресурсам.
- `yii\base\Theme::$pathMap`: правила замены файлов представлений. Подробно описаны далее.

Например, если вы вызываете `$this->render('about')` в `SiteController`, то будет использоваться файл отображения `@app/views/site/about.php`. Тем не менее, если вы включите темизацию как показано ниже, то вместо него будет использоваться `@app/themes/basic/site/about.php`.

```
return [  
    'components' => [  
        'view' => [  
            'theme' => [  
                'basePath' => '@app/themes/basic',  
                'baseUrl' => '@web/themes/basic',  
                'pathMap' => [  
                    '@app/views' => '@app/themes/basic',  
                ],  
            ],  
        ],  
    ],  
];
```

Информация: При настройке тем поддерживаются псевдонимы пути. При замене отображений они преобразуются в реальные пути в файловой системе или URL.

Вы можете обратиться к объекту `yii\base\Theme` через свойство `yii\base\View::$theme`. Например, в файле отображения, это будет выглядеть следующим образом (объект `view` доступен как `$this`):

```
$theme = $this->theme;  
  
// returns: $theme->baseUrl . '/img/logo.gif'  
$url = $theme->getUrl('img/logo.gif');
```

```
// returns: $theme->basePath . '/img/logo.gif'
$file = $theme->getPath('img/logo.gif');
```

Свойство `yii\base\Theme::$pathMap` определяет то, как заменяются файлы представлений. Свойство принимает массив пар ключ-значение где ключи являются путями к оригинальным файлам, которые мы хотим заменить, а значения — соответствующими путями к файлам из темы. Замена основана на частичном совпадении: если путь к представлению начинается с любого из ключей массива `pathMap`, то соответствующая ему часть будет заменена значением из того же массива. Для приведённой выше конфигурации `@app/views/site/about.php` частично совпадает с ключом `@app/views` и будет заменён на `@app/themes/basic/site/about.php`.

8.6.1 Темизация модулей

Для того, чтобы темизировать модули, свойство `yii\base\Theme::$pathMap` может быть настроено следующим образом:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/modules' => '@app/themes/basic/modules', // <-- !!!
],
```

Это позволит вам темизировать `@app/modules/blog/views/comment/index.php` в `@app/themes/basic/modules/blog/views/comment/index.php`.

8.6.2 Темизация виджетов

Для того, чтобы темизировать виджеты вы можете настроить свойство `yii\base\Theme::$pathMap` следующим образом:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/widgets' => '@app/themes/basic/widgets', // <-- !!!
],
```

Это позволит вам темизировать `@app/widgets/currency/views/index.php` в `@app/themes/basic/widgets/currency/index.php`.

8.6.3 Наследование тем

Иногда требуется создать базовую тему, задающую общий вид приложения и далее изменять этот вид в зависимости, например, от сегодняшнего праздника. Добиться этого можно при помощи наследования тем. При этом один путь к файлу ставится в соответствие нескольким путям из темы:

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
```

```
        '@app/themes/basic',  
    ],  
]
```

В этом случае представление `@app/views/site/index.php` темизируется либо в `@app/themes/christmas/site/index.php`, либо в `@app/themes/basic/site/index.php` в зависимости от того, в какой из тем есть нужный файл. Если файлы присутствуют и там и там, используется первый из них. На практике большинство темизированных файлов будут расположены в `@app/themes/basic`, а их версии для праздников в `@app/themes/christmas`.

Глава 9

Безопасность

Error: not existing file: security-overview.md

9.1 Аутентификация

Аутентификация — это процесс проверки подлинности пользователя. Обычно используется идентификатор (например, `username` или адрес электронной почты) и секретный токен (например, пароль или ключ доступа), чтобы судить о том, что пользователь именно тот, за кого себя выдаёт. Аутентификация является основной функцией формы входа.

Yii предоставляет фреймворк авторизации с различными компонентами, обеспечивающими процесс входа. Для использования этого фреймворка вам нужно проделать следующее:

- Настроить компонент приложения `user`;
- Создать класс, реализующий интерфейс `yii\web\IdentityInterface`.

9.1.1 Настройка `yii\web\User`

Компонент `user` управляет статусом аутентификации пользователя. Он требует, чтобы вы указали `identity class`, который будет содержать текущую логику аутентификации. В следующей конфигурации приложения, `identity class` для `user` задан как `app\models\User`, реализация которого будет объяснена в следующем разделе:

```
return [  
    'components' => [  
        'user' => [  
            'identityClass' => 'app\models\User',  
        ],  
    ],  
];
```

9.1.2 Реализация `yii\web\IdentityInterface`

`identity class` должен реализовывать `yii\web\IdentityInterface`, который содержит следующие методы:

- `findIdentity()`: Этот метод находит экземпляр `identity class`, используя ID пользователя. Этот метод используется, когда необходимо поддерживать состояние аутентификации через сессии.
- `findIdentityByAccessToken()`: Этот метод находит экземпляр `identity class`, используя токен доступа. Метод используется, когда требуется аутентифицировать пользователя только по секретному токenu (например в RESTful приложениях, не сохраняющих состояние между запросами).
- `getId()`: Этот метод возвращает ID пользователя, представленного данным экземпляром `identity`.
- `getAuthKey()`: Этот метод возвращает ключ, используемый для основанной на `cookie` аутентификации. Ключ сохраняется в аутентификационной `cookie` и позже сравнивается с версией, находящейся

на сервере, чтобы удостовериться, что аутентификационная cookie верная.

- **validateAuthKey()**: Этот метод реализует логику проверки ключа для основанной на cookie аутентификации.

Если какой-то из методов не требуется, то можно реализовать его с пустым телом. Для примера, если у вас RESTful приложение, не сохраняющее состояние между запросами, вы можете реализовать только **findIdentityByAccessToken()** и **getId()**, тогда как остальные методы оставить пустыми.

В следующем примере, **identity class** реализован как класс **Active Record**, связанный с таблицей **user**.

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * Finds an identity by the given ID.
     *
     * @param string|integer $id the ID to be looked for
     * @return IdentityInterface|null the identity object that matches the
     * given ID.
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * Finds an identity by the given token.
     *
     * @param string $token the token to be looked for
     * @return IdentityInterface|null the identity object that matches the
     * given token.
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string current user ID
     */
    public function getId()
```



```

{
    return $this->id;
}

/**
 * @return string current user auth key
 */
public function getAuthKey()
{
    return $this->auth_key;
}

/**
 * @param string $authKey
 * @return boolean if auth key is valid for current user
 */
public function validateAuthKey($authKey)
{
    return $this->getAuthKey() === $authKey;
}
}

```

Как объяснялось ранее, вам нужно реализовать только `getAuthKey()` и `validateAuthKey()`, если ваше приложение использует только аутентификацию основанную на cookie. В этом случае вы можете использовать следующий код для генерации ключа аутентификации для каждого пользователя и хранения его в таблице `user`:

```

class User extends ActiveRecord implements IdentityInterface
{
    .....

    public function beforeSave($insert)
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
                $this->auth_key = \Yii::$app->security->generateRandomString
            );
            }
            return true;
        }
        return false;
    }
}

```

Примечание: Не путайте `identity` класс `User` с классом `yii\web\User`. Первый является классом, реализующим логику аутентификации пользователя. Он часто реализуется как класс [Active Record](#), связанный с некоторым постоянным хранилищем, где лежит информация о пользователях. Второй — это класс компонента приложения, отвечающий за управление состоянием аутентификации пользователя.

9.1.3 Использование yii\web\User

В основном класс `yii\web\User` используют как компонент приложения `user`.

Можно получить `identity` текущего пользователя, используя выражение `Yii::$app->user->identity`. Оно вернёт экземпляр `identity class`, представляющий текущего аутентифицированного пользователя, или `null`, если текущий пользователь не аутентифицирован (например, гость). Следующий код показывает, как получить другую связанную с аутентификацией информацию из `yii\web\User`:

```
// 'identity' текущего пользователя. 'Null', если пользователь не
// аутентифицирован.
$identity = Yii::$app->user->identity;

// ID текущего пользователя. 'Null', если пользователь не аутентифицирован.
$id = Yii::$app->user->id;

// проверка на то, что текущий пользователь гость не ( аутентифицирован)
$isGuest = Yii::$app->user->isGuest;
```

Для залогинивания пользователя вы можете использовать следующий код:

```
// найти identity с указанным username.
// замечание: также вы можете проверить и пароль, если это нужно
$identity = User::findOne(['username' => $username]);

// логиним пользователя
Yii::$app->user->login($identity);
```

Метод `yii\web\User::login()` устанавливает `identity` текущего пользователя в `yii\web\User`. Если сессии включены, то `identity` будет сохраняться в сессии, так что состояние статуса аутентификации будет поддерживаться на всём протяжении сессии. Если включен вход, основанный на cookie (так называемый “запомни меня” вход), то `identity` также будет сохранена в cookie так, чтобы статус аутентификации пользователя мог быть восстановлен на протяжении всего времени жизни cookie.

Для включения входа, основанного на cookie, вам нужно установить `yii\web\User::$enableAutoLogin` в `true` в конфигурации приложения. Вы также можете настроить время жизни, передав его при вызове метода `yii\web\User::login()`.

Для выхода пользователя, просто вызовите

```
Yii::$app->user->logout();
```

Обратите внимание: выход пользователя имеет смысл только если сессии включены. Метод сбрасывает статус аутентификации сразу и из памяти и из сессии. И по умолчанию, будут также уничтожены все сессионные данные пользователя. Если вы хотите сохранить сессионные данные, вы должны вместо этого вызвать `Yii::$app->user->logout(false)`.

9.1.4 События аутентификации

Класс `yii\web\User` вызывает несколько событий во время процессов входа и выхода.

- `EVENT_BEFORE_LOGIN`: вызывается перед вызовом `yii\web\User::login()`. Если обработчик устанавливает свойство `isValid` объекта в `false`, процесс входа будет прерван.
- `EVENT_AFTER_LOGIN`: вызывается после успешного входа.
- `EVENT_BEFORE_LOGOUT`: вызывается перед вызовом `yii\web\User::logout()`. Если обработчик устанавливает свойство `isValid` объекта в `false`, процесс выхода будет прерван.
- `EVENT_AFTER_LOGOUT`: вызывается после успешного выхода.

Вы можете использовать эти события для реализации функции аудита входа, сбора статистики онлайн пользователей. Например, в обработчике для `EVENT_AFTER_LOGIN` вы можете сделать запись о времени и IP адресе входа в таблицу `user`.

9.2 Авторизация

Примечание: этот раздел находится на стадии разработки.

Авторизация — это процесс проверки того, что пользователь имеет достаточно прав, чтобы выполнить какие-то действия. Yii предоставляет два метода авторизации: фильтры контроля доступа (ACF) и контроль доступа на основе ролей (RBAC).

9.2.1 Фильтры контроля доступа

Фильтры контроля доступа (ACF) являются простым методом, который лучше всего использовать в приложениях с простым контролем доступа. Как видно из их названия, ACF — это фильтры, которые могут присоединяться к контроллеру или модулю как поведение. ACF проверяет набор правил доступа, чтобы убедиться, что пользователь имеет доступ к запрошенному действию.

Код ниже показывает, как использовать ACF фильтр, реализованный в `yii\filters\AccessControl`:

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['login', 'logout', 'signup'],
            ],
        ];
    }
}
```

```

        'rules' => [
            [
                'allow' => true,
                'actions' => ['login', 'signup'],
                'roles' => ['?'],
            ],
            [
                'allow' => true,
                'actions' => ['logout'],
                'roles' => ['@'],
            ],
        ],
    ],
];
}
// ...
}

```

Код выше показывает ACF фильтр, связанный с контроллером `site` через поведение. Это типичный способ использования фильтров действий. Параметр `only` указывает, что фильтр ACF нужно применять только к действиям `login`, `logout` и `signup`. Параметр `rules` задаёт правила доступа, которые означают следующее:

- Разрешить всем гостям (ещё не прошедшим авторизацию) доступ к действиям `login` и `signup`. Опция `roles` содержит знак вопроса `?`, это специальный токен обозначающий “гостя”.
- Разрешить аутентифицированным пользователям доступ к действию `logout`. Символ `@` — это другой специальный токен, обозначающий аутентифицированного пользователя.

Когда фильтр ACF проводит проверку авторизации, он проверяет правила по одному сверху вниз, пока не найдёт совпадение. Значение опции `allow` выбранного правила указывает, авторизовывать пользователя или нет. Если ни одно из правил не совпало, то пользователь считается НЕавторизованным, и фильтр ACF останавливает дальнейшее выполнение действия.

По умолчанию, когда у пользователя отсутствует доступ к текущему действию, ACF делает следующее:

- Если пользователь гость, вызывается `yii\web\User::loginRequired()`, который перенаправляет браузер на страницу входа.
- Если пользователь авторизован, генерируется исключение `yii\web\ForbiddenHttpException`.

Вы можете переопределить это поведение, настроив свойство `yii\filters\AccessControl::$denyCallback`:

```

[
    'class' => AccessControl::className(),
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('У вас нет доступа к этой странице');
    }
]

```

]

Правила доступа поддерживают набор свойств. Ниже дано краткое описание поддерживаемых опций. Вы также можете расширить `yii\filters\AccessRule`, чтобы создать свой собственный класс правил доступа.

- **allow**: задаёт какое это правило, “allow” или “deny”.
- **actions**: задаёт действия, соответствующие этому правилу. Значение должно быть массивом идентификаторов действий. Сравнение — регистрозависимо. Если свойство пустое или не задано, то правило применяется ко всем действиям.
- **controllers**: задаёт контроллеры, которым соответствует правило. Значение должно быть массивом с идентификаторами контроллеров. Сравнение регистрозависимо. Если свойство пустое или не задано, то правило применяется ко всем контроллерам.
- **roles**: задаёт роли пользователей, соответствующих этому правилу. Распознаются две специальные роли, которые проверяются с помощью `yii\web\User::$isGuest`:
 - `?`: соответствует гостевому пользователю (не аутентифицирован),
 - `@`: соответствует аутентифицированному пользователю.

Использование других имён ролей будет приводить к вызову метода `yii\web\User::can()`, который требует включения RBAC (будет описано дальше). Если свойство пустое или не задано, то правило применяется ко всем ролям.

- **ips**: задаёт IP адреса пользователей, для которых применяется это правило. IP адрес может содержать * в конце, так чтобы он соответствовал IP адресу с таким же префиксом. Для примера, ‘192.168.*’ соответствует всем IP адресам в сегменте ‘192.168.’. Если свойство пустое или не задано, то правило применяется ко всем IP адресам.
- **verbs**: задаёт http метод (например, GET, POST), соответствующий правилу. Сравнение — регистронезависимо.
- **matchCallback**: задаёт PHP колбек, который вызывается для определения, что правило должно быть применено.
- **denyCallback**: задаёт PHP колбек, который будет вызван, если доступ будет запрещён при вызове этого правила.

Ниже показан пример, показывающий использование опции `matchCallback`, которая позволяет писать произвольную логику проверки доступа:

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
```

```

        'access' => [
            'class' => AccessControl::className(),
            'only' => ['special-callback'],
            'rules' => [
                [
                    'actions' => ['special-callback'],
                    'allow' => true,
                    'matchCallback' => function ($rule, $action) {
                        return date('d-m') === '31-10';
                    }
                ],
            ],
        ],
    ];
}

// Колбек сработал! Эта страница может быть отображена только 31-ого
// октября
public function actionSpecialCallback()
{
    return $this->render('happy-halloween');
}
}

```

9.2.2 Контроль доступа на основе ролей (RBAC)

Управление доступом на основе ролей (RBAC) обеспечивает простой, но мощный централизованный контроль доступа. Пожалуйста, обратитесь к Wikipedia¹ для получения информации о сравнении RBAC с другими, более традиционными, системами контроля доступа.

Yii реализует общую иерархическую RBAC, следуя NIST RBAC model². Обеспечивается функциональность RBAC через компонент приложения `authManager`.

Использование RBAC состоит из двух частей. Первая часть — это создание RBAC данных авторизации, и вторая часть — это использование данных авторизации для проверки доступа в том месте, где это нужно.

Для облегчения последующего описания, мы сначала введём некоторые основные понятия RBAC.

Основные концепции

Роль представляет собой набор разрешений (*permissions*) (например, создание сообщения, обновление сообщения). Роль может быть назначена

¹https://ru.wikipedia.org/wiki/%D0%A3%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B4%D0%BE%D1%81%D1%82%D1%83%D0%BF%D0%BE%D0%BC_%D0%BD%D0%B0_%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%B5_%D1%80%D0%BE%D0%BB%D0%B5%D0%B9

²<http://csrc.nist.gov/rbac/sandhu-ferraiolo-kuhn-00.pdf>

на одного или многих пользователей. Чтобы проверить, имеет ли пользователь указанные разрешения, мы должны проверить, назначена ли пользователю роль, которая содержит данное разрешение.

С каждой ролью или разрешением может быть связано правило (*rule*). Правило представляет собой кусок кода, который будет выполняться в ходе проверки доступа для определения может ли быть применена соответствующая роль или разрешение к текущему пользователю. Например, разрешение “обновление поста” может иметь правило, которое проверяет является ли текущий пользователь автором поста. Во время проверки доступа, если пользователь не является автором поста, он/она будет считаться не имеющими разрешения “обновление поста”.

И роли, и разрешения могут быть организованы в иерархию. В частности, роль может содержать другие роли или разрешения; и разрешения могут содержать другие разрешения. Yii реализует *частично упорядоченную* иерархию, которая включает в себя специальные *деревья* иерархии. Роль может содержать разрешение, но обратное не верно.

Настройка RBAC Manager

Перед определением авторизационных данных и проверкой прав доступа, мы должны настроить компонент приложения `authManager`. Yii предоставляет два типа менеджеров авторизации: `yii\rbac\PhpManager` и `yii\rbac\DbManager`. Первый использует файл с PHP скриптом для хранения данных авторизации, второй сохраняет данные в базе данных. Вы можете использовать первый, если ваше приложение не требует слишком динамичного управления ролями и разрешениями.

Настройка authManager с помощью PhpManager Следующий код показывает как настроить в конфигурации приложения `authManager` с использованием класса `yii\rbac\PhpManager`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

Теперь `authManager` может быть доступен через `\Yii::$app->authManager`.

Замечание: По умолчанию, `yii\rbac\PhpManager` сохраняет данные RBAC в файлах в директории `@app/rbac/`. Убедитесь что данная директория и файлы в них доступны для записи Web серверу, если иерархия разрешений должна меняться онлайн.

Настройка authManager с помощью DbManager Следующий код показывает как настроить в конфигурации приложения authManager с использованием класса yii\rbac\DbManager:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
        ],
        // ...
    ],
];
```

DbManager использует четыре таблицы для хранения данных:

- **itemTable**: таблица для хранения авторизационных элементов. По умолчанию “auth_item”.
- **itemChildTable**: таблица для хранения иерархии элементов. По умолчанию “auth_item_child”.
- **assignmentTable**: таблица для хранения назначений элементов авторизации. По умолчанию “auth_assignment”.
- **ruleTable**: таблица для хранения правил. По умолчанию “auth_rule”.

Прежде чем вы начнёте использовать этот менеджер, вам нужно создать таблицы в базе данных. Чтобы сделать это, вы можете использовать миграцию хранящуюся в файле @yii/rbac/migrations:

```
yii migrate --migrationPath=@yii/rbac/migrations
```

Теперь authManager может быть доступен через \Yii::\$app->authManager.

Создание данных авторизации

Для создания данных авторизации нужно выполнить следующие задачи:

- определение ролей и разрешений;
- установка отношений между ролями и правами доступа;
- определение правил;
- связывание правил с ролями и разрешениями;
- назначение ролей пользователям.

В зависимости от требований к гибкости авторизации перечисленные задачи могут быть выполнены разными путями.

Если иерархия прав не меняется, и количество пользователей зафиксировано, вы можете создать **консольную команду**, которая будет единожды инициализировать данные через APIs, предоставляемое authManager:

```
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
```



```
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;

        // добавляем разрешение "createPost"
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Create a post';
        $auth->add($createPost);

        // добавляем разрешение "updatePost"
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Update post';
        $auth->add($updatePost);

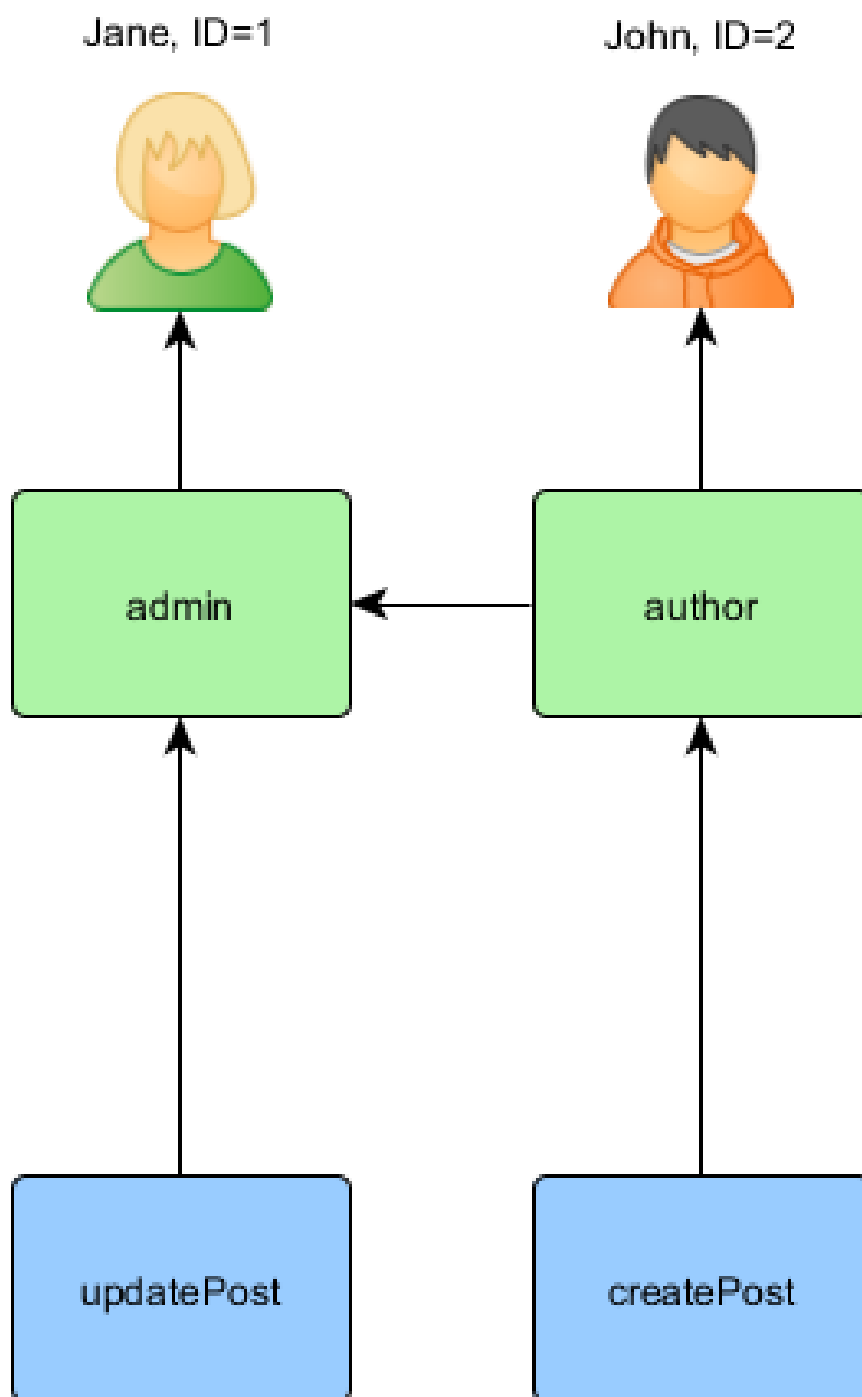
        // добавляем роль "author" и даём роли разрешение "createPost"
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // добавляем роль "admin" и даём роли разрешение "updatePost"
        // а также все разрешения роли "author"
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // Назначение ролей пользователям. 1 и 2 это IDs возвращаемые
        IdentityInterface::getId()
        // обычно реализуемый в модели User.
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }
}
```

Примечание: Если вы используете шаблон проекта advanced, RbacController необходимо создать в директории console/controllers и сменить пространство имён на console/controllers.

После выполнения команды `yii rbac/init` мы получим следующую иерархию:



Автор может создавать пост, администратор может обновлять пост и делать всё, что может делать автор.

Если ваше приложение позволяет регистрировать пользователей, то вам необходимо сразу назначать роли этим новым пользователям. Например, для того, чтобы все вошедшие пользователи могли стать авторами в расширенном шаблоне проекта, вы должны изменить `frontend\models\SignupForm::signup()` как показано ниже:

```
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save(false);

        // нужно добавить следующие три строки:
        $auth = Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

Для приложений, требующих комплексного контроля доступа с динамически обновляемыми данными авторизации, существуют специальные пользовательские интерфейсы (так называемые админ-панели), которые могут быть разработаны с использованием API, предлагаемого `authManager`.

Использование правил

Как упомянуто выше, правила добавляют дополнительные ограничения на роли и разрешения. Правила — это классы, расширяющие `yii\rbac\Rule`. Они должны реализовывать метод `execute()`. В иерархии, созданной нами ранее, автор не может редактировать свой пост. Давайте исправим это. Сначала мы должны создать правило, проверяющее что пользователь является автором поста:

```
namespace app\rbac;

use yii\rbac\Rule;

/**
 * Проверяем authorID на соответствие с пользователем, переданным через
 * параметры
 */
class AuthorRule extends Rule
{
    }
```

```

public $name = 'isAuthor';

/**
 * @param string|integer $user the user ID.
 * @param Item $item the role or permission that this rule is associated
 * width.
 * @param array $params parameters passed to ManagerInterface::
 * checkAccess().
 * @return boolean a value indicating whether the rule permits the role
 * or permission it is associated with.
 */
public function execute($user, $item, $params)
{
    return isset($params['post']) ? $params['post']->createdBy == $user
    : false;
}
}

```

Правило выше проверяет, что `post` был создан `$user`. Мы создадим специальное разрешение `updateOwnPost` в команде, которую мы использовали ранее:

```

$auth = Yii::$app->authManager;

// add the rule
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

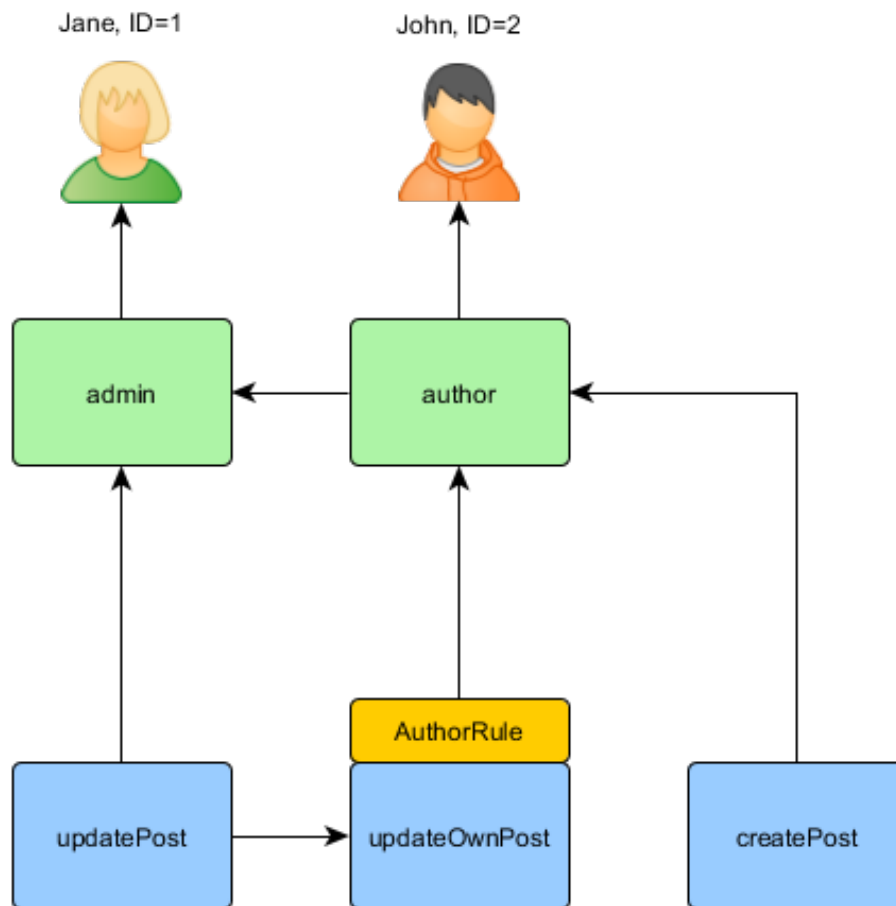
// добавляем разрешение "updateOwnPost" и привязываем к нему правило.
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" будет использоваться из "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// разрешаем автору "" обновлять его посты
$auth->addChild($author, $updateOwnPost);

```

Теперь мы имеем следующую иерархию:

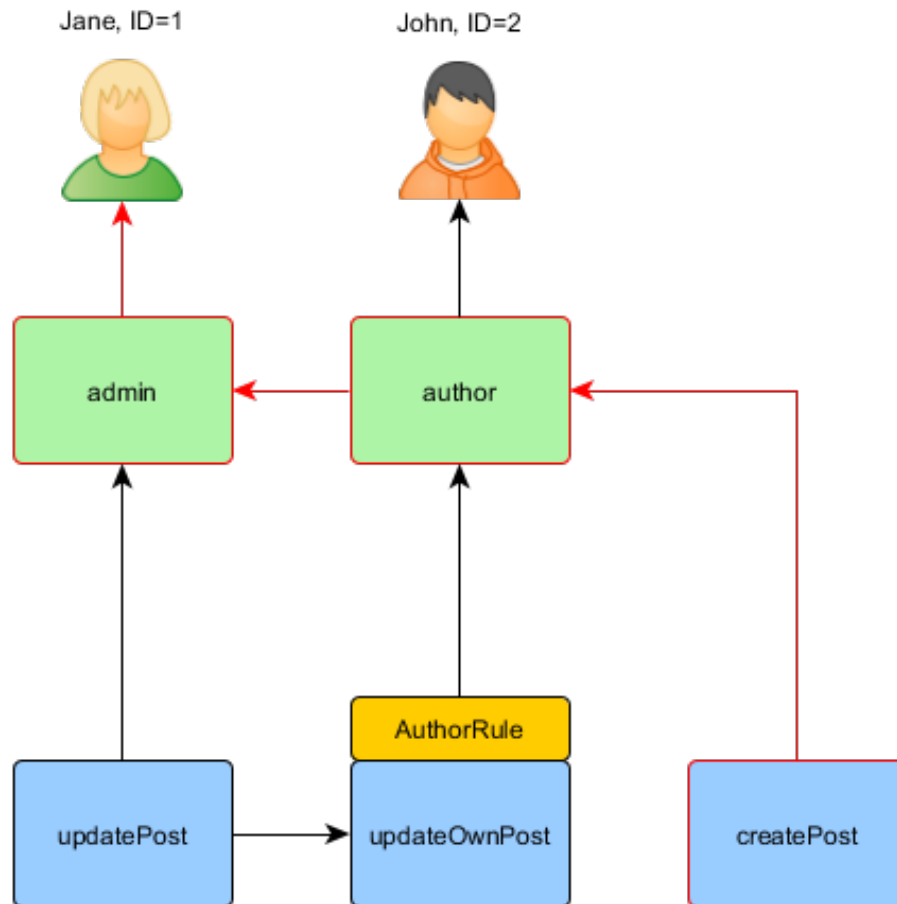


Проверка доступа

С готовыми авторизационными данными проверка доступа — это просто вызов метода `yii\rbac\ManagerInterface::checkAccess()`. Так как большинство проверок доступа относятся к текущему пользователю, для удобства Yii предоставляет сокращённый метод `yii\web\User::can()`, который можно использовать как показано ниже:

```
if (\Yii::$app->user->can('createPost')) {  
    // create post  
}
```

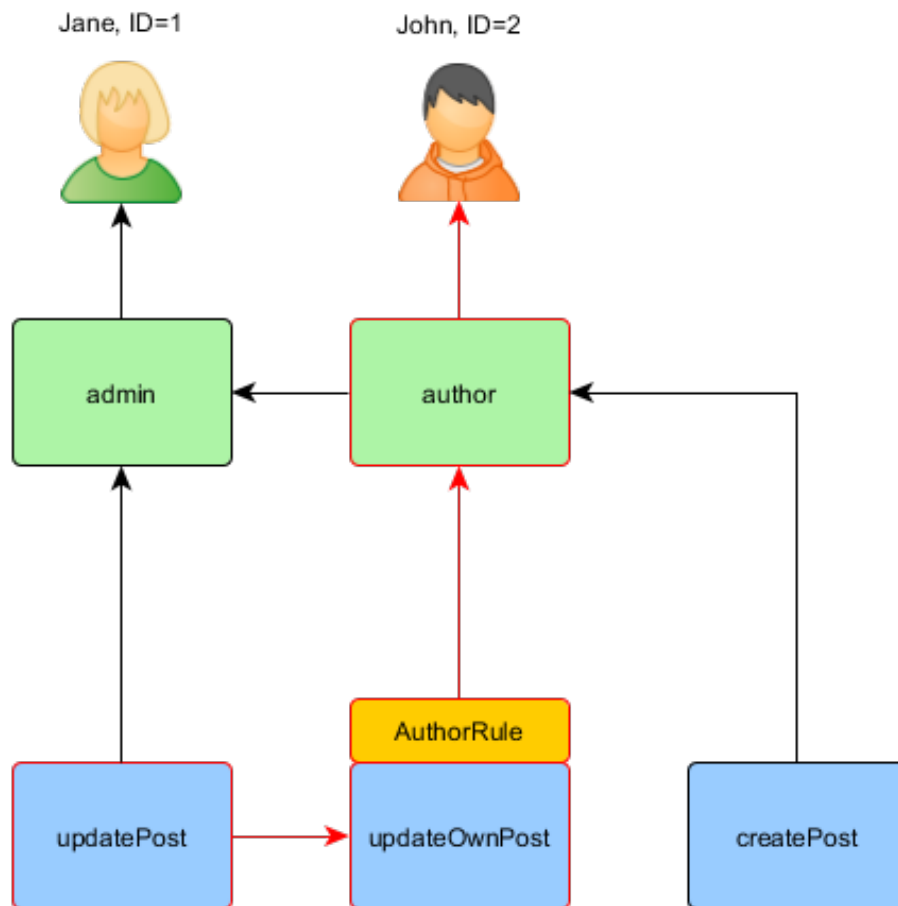
Если текущий пользователь Jane с ID=1, мы начнём с `createPost` и попробуем добраться до Jane:



Для того чтобы проверить, может ли пользователь обновить пост, нам надо передать дополнительный параметр, необходимый для правила `AuthorRule`, описанного ранее:

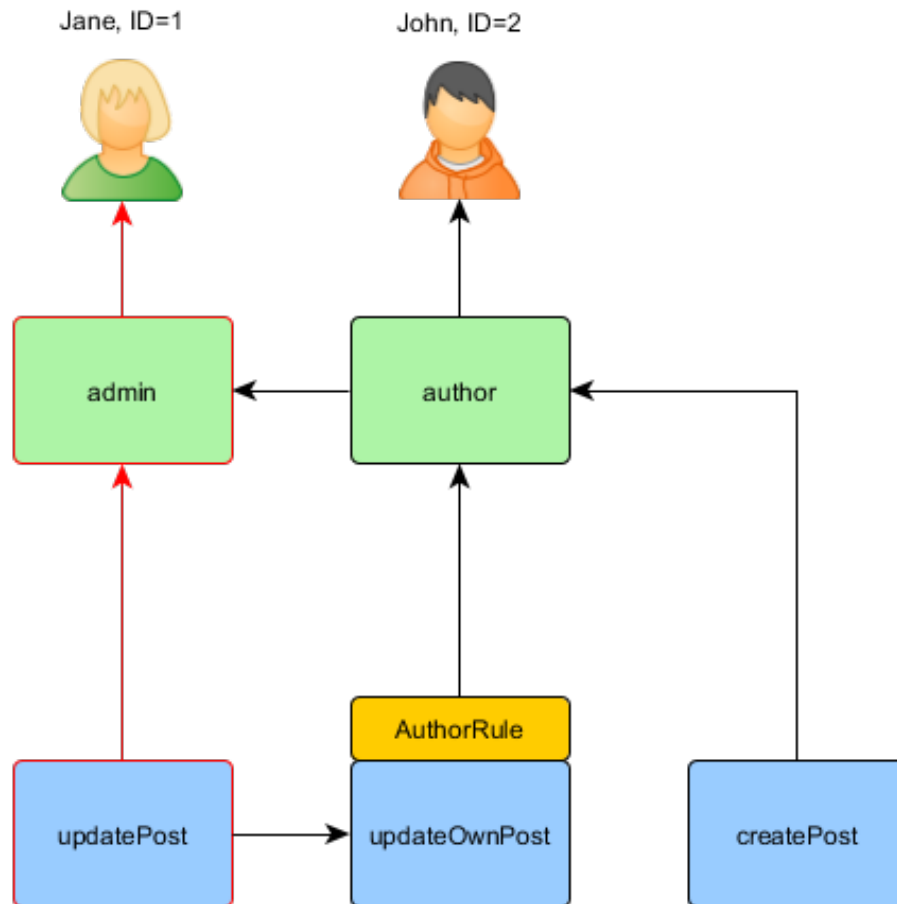
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {  
    // update post  
}
```

Вот что происходит если текущим пользователем является John:



Мы начинаем с `updatePost` и переходим к `updateOwnPost`. Для того чтобы это произошло, правило `AuthorRule` должно вернуть `true` при вызове метода `execute`. Метод получает `$params`, переданный при вызове метода `can`, значение которого равно `['post' => $post]`. Если всё правильно, мы увидим, что `author` привязан к John.

В случае Jane это немного проще, потому что она `admin`:



Есть несколько способов реализовать авторизацию в контроллере. Если вам необходимы отдельные права на добавление и удаление, то проверку стоит делать в каждом действии. Вы можете либо использовать условие выше в каждом методе действия, либо использовать `yii\filters\AccessControl`:

```

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['index'],
                    'roles' => ['managePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['view'],
                ],
            ],
        ],
    ];
}
  
```



```

        'roles' => ['viewPost'],
    ],
    [
        'allow' => true,
        'actions' => ['create'],
        'roles' => ['createPost'],
    ],
    [
        'allow' => true,
        'actions' => ['update'],
        'roles' => ['updatePost'],
    ],
    [
        'allow' => true,
        'actions' => ['delete'],
        'roles' => ['deletePost'],
    ],
],
],
];
}

```

Если права на все CRUD операции выдаются вместе, то лучшее решение в этом случае — завести одно разрешение вроде `managePost` и проверять его в `yii\web\Controller::beforeAction()`.

Использование роли по умолчанию

Роль по умолчанию — это роль, которая *неявно* присваивается *всем* пользователям. Вызов метода `yii\rbac\ManagerInterface::assign()` не требуется, и авторизационные данные не содержат информации о назначении.

Роль по умолчанию обычно связывают с правилом, определяющим к какой роли принадлежит каждый пользователь.

Роли по умолчанию обычно используются в приложениях, которые уже имеют какое-то описание ролей. Для примера, приложение может иметь столбец “group” в таблице пользователей, и каждый пользователь принадлежит к какой-то группе. Если каждая группа может быть сопоставлена роли в модели RBAC, вы можете использовать роль по умолчанию для автоматического назначения каждому пользователю роли RBAC. Давайте используем пример, чтобы понять как это можно сделать.

Предположим что в таблице пользователей у вас есть столбец `group`, в котором значение 1 представляет группу “администратор”, а 2 — группу “автор”. Вы планируете иметь две RBAC роли: `admin` и `author`, представляющие разрешения для двух соответствующих групп. Вы можете настроить данные роли как показано ниже.

```
namespace app\rbac;
```

```

use Yii;
use yii\rbac\Rule;

/**
 * Checks if user group matches
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;
            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }
        return false;
    }
}

$auth = Yii::$app->authManager;

$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... add permissions as children of $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... add permissions as children of $admin ...

```

Обратите внимание, так как “author” добавлен как дочерняя роль к “admin”, следовательно в реализации метода `execute()` класса правила вы должны учитывать эту иерархию. Именно поэтому для роли “author” метод `execute()` вернёт истину, если пользователь принадлежит к группам 1 или 2 (это означает, что пользователь находится в группе администраторов или авторов)

Далее настроим `authManager` с помощью перечисления ролей в свойстве `yii\rbac\BaseManager::$defaultRoles`:

```

return [
    // ...
    'components' => [

```

```
'authManager' => [  
    'class' => 'yii\rbac\PhpManager',  
    'defaultRoles' => ['admin', 'author'],  
],  
    // ...  
],  
];
```

Теперь, если вы выполните проверку доступа, для обоих ролей `admin` и `author` будет выполнена проверка правила, ассоциированного с ними. Если правило вернёт истину, это будет означать, что роль применяется к текущему пользователю. На основании правила, реализованного выше: если пользователь входит в группу 1, пользователю будет назначена роль `admin`; и если значение `group` равно 2, будет применена роль `author`.

9.3 Работа с паролями

Многие разработчики знают, что хранить пароль открытым текстом нельзя, но многие до сих пор считают безопасным использование для хеширования паролей `md5` или `sha1`. Раньше упомянутых алгоритмов было достаточно, но современное оборудование позволяет подобрать эти хеши очень быстро, методом простого перебора.

Для того, чтобы обеспечить повышенную безопасность паролей ваших пользователей даже в худшем случае (ваше приложение взломано), нужно использовать алгоритм шифрования, устойчивый к атаке перебором. Лучший вариант в текущий момент `bcrypt`. В PHP вы можете использовать хеши `bcrypt` через функцию `crypt`³. Yii обеспечивает две вспомогательные функции, которые упрощают использование функции `crypt` для генерации и проверки пароля.

Когда пользователь задаёт пароль (например во время регистрации), пароль должен быть захеширован:

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

Хеш можно связать с соответствующим атрибутом модели, так чтобы он сохранялся в базе для последующего использования.

Когда пользователь попытается войти, отправленный пароль должен быть хеширован и сравнён с ранее сохранённым хешем:

```
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {  
    // всё хорошо, пользователь может войти  
} else {  
    // неправильный пароль  
}
```

³<http://php.net/manual/en/function.crypt.php>

Error: not existing file: security-cryptography.md

**Error: not existing file: [https://github.com/yiisoft/yii2-authclient/blob/master/docs/guid
ru/README.md](https://github.com/yiisoft/yii2-authclient/blob/master/docs/guid
ru/README.md)**

9.4 Лучшие практики безопасности

Ниже мы рассмотрим основные принципы безопасности и опишем, как избежать угроз при разработке на Yii.

9.4.1 Основные принципы

Есть два основных принципа безопасности, независимо от того, какое приложение разрабатывается:

1. Фильтрация ввода.
2. Экранирование вывода.

Фильтрация ввода

Фильтрация ввода означает, что входные данные никогда не должны считаться безопасными и вы всегда должны проверять, являются ли полученные данные допустимыми. Например, если мы знаем, что сортировка может быть осуществлена только по трём полям `title`, `created_at` и `status`, и поле может передаваться через ввод пользователем, лучше проверить значение там, где мы его получили:

```
$sortBy = $_GET['sort'];  
if (!in_array($sortBy, ['title', 'created_at', 'status'])) {  
    throw new Exception('Invalid sort value.');
```

В Yii, вы, скорее всего, будете использовать [валидацию форм](#), чтоб делать такие проверки.

Экранирование вывода

Экранирование вывода означает, что данные в зависимости от контекста должны экранироваться, например в контексте HTML вы должны экранировать `<`, `>` и похожие специальные символы. В контексте JavaScript или SQL будет другой набор символов. Так как ручное экранирование чревато ошибками, Yii предоставляет различные утилиты для экранирования в различных контекстах.

9.4.2 Как избежать SQL-инъекций

SQL-инъекции происходят, когда текст запроса формируется склеиванием не экранированных строк, как показано ниже:

```
$username = $_GET['username'];  
$sql = "SELECT * FROM user WHERE username = '$username'";
```

Вместо того, чтобы подставлять корректное имя пользователя, злоумышленник может передать вам в приложение что-то вроде `' ; DROP TABLE user ; --`. В результате SQL будет следующий:

```
SELECT * FROM user WHERE username = '' ; DROP TABLE user ; --'
```

Это валидный запрос, который сначала будет искать пользователей с пустым именем, а затем удалит таблицу `user`. Скорее всего будет сломано приложение и будут потеряны данные (вы ведь делаете регулярное резервное копирование?).

Большинство запросов к базе данных в Yii происходит через [Active Record](#), который правильно использует подготовленные запросы PDO внутри. При использовании подготовленных запросов невозможно манипулировать запросом как это показано выше.

Тем не менее, иногда нужны [сырые запросы](#) или [построитель запросов](#). В этом случае вы должны использовать безопасные способы передачи данных. Если данные используются для сравнения со значением столбцов предпочтительнее использовать подготовленные запросы:

```
// query builder
$userIDs = (new Query())
    ->select('id')
    ->from('user')
    ->where('status=:status', [':status' => $status])
    ->all();

// DAO
$userIDs = $connection
    ->createCommand('SELECT id FROM user where status=:status')
    ->bindValues([':status' => $status])
    ->queryColumn();
```

Если данные используются в качестве имён столбцов или таблиц, то лучший путь - это разрешить только предопределённый набор значений:

```
function actionList($orderBy = null)
{
    if (!in_array($orderBy, ['name', 'status'])) {
        throw new BadRequestHttpException('Only name and status are allowed to order by.')
    }

    // ...
}
```

Если это невозможно, то имена столбцов и таблиц должны экранироваться. Yii использует специальный синтаксис для экранирования для всех поддерживаемых баз данных:

```
$sql = "SELECT COUNT([[ $column ]]) FROM {{ $table }}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

Вы можете получить подробную информацию о синтаксисе в [Экранирование имён таблиц и столбцов](#).

9.4.3 Как избежать XSS

XSS или кросс-сайтинговый скриптинг становится возможен, когда не экранированный выходной HTML попадает в браузер. Например, если пользователь должен ввести своё имя, но вместо **Alexander** он вводит `<script>alert('Hello!');</script>`, то все страницы, которые его выводят без экранирования, будут выполнять JavaScript `alert('Hello!')`, и в результате будет выводиться окно сообщения в браузере. В зависимости от сайта, вместо невинных скриптов с выводом всплывающего hello, злоумышленниками могут быть отправлены скрипты, похищающие личные данные пользователей сайта, либо выполняющие операции от их имени.

В Yii избежать XSS легко. На месте вывода текста необходимо выбрать один из двух вариантов:

1. Вы хотите вывести данные в виде обычного текста.
2. Вы хотите вывести данные в виде HTML.

Если вам нужно вывести простой текст, то экранировать лучше следующим образом:

```
<?= \yii\helpers\Html::encode($username) ?>
```

Если нужно вывести HTML, вам лучше воспользоваться `HtmlPurifier`:

```
<?= \yii\helpers\HtmlPurifier::process($description) ?>
```

Обратите внимание, что обработка с помощью `HtmlPurifier` довольно тяжела, так что неплохо бы задуматься о кешировании.

9.4.4 Как избежать CSRF

CSRF - это аббревиатура для межсайтинговой подмены запросов. Идея заключается в том, что многие приложения предполагают, что запросы, приходящие от браузера, отправляются самим пользователем. Это может быть неправдой.

Например, сайт `an.example.com` имеет URL `/logout`, который, используя простой GET, разлогинивает пользователя. Пока это запрос выполняется самим пользователем - всё в порядке, но в один прекрасный день злоумышленники размещают код `` на форуме с большой посещаемостью. Браузер не делает никаких отличий между запросом изображения и запросом страницы, так что когда пользователь откроет страницу с таким тегом `img`, браузер отправит GET запрос на указанный адрес, и пользователь будет разлогинен.

Вот основная идея. Можно сказать, что в разлогировании пользователя нет ничего серьёзного, но с помощью этой уязвимости, можно выполнять опасные операции. Представьте, что существует страница <http://an.example.com/logout>:

`//an.example.com/purse/transfer?to=anotherUser&amout=2000`, обращение к которой с помощью GET запроса, приводит к перечислению 2000 единиц валюты со счета авторизованного пользователя на счет пользователя с логином `anotherUser`. Учитывая, что браузер для загрузки контента отправляет GET запросы, можно подумать, что разрешение на выполнение такой операции только POST запросом на 100

Для того, чтоб избежать CSRF вы должны всегда:

1. Следуйте спецификациям HTTP, например запрос GET не должен менять состояние приложения.
2. Держите защиту CSRF в Yii включенной.

9.4.5 Как избежать нежелательного доступа к файлам

По умолчанию, webroot сервера указывает на каталог `web`, где лежит `index.php`. В случае использования виртуального хостинга, это может быть недостижимо, в конечном итоге весь код, конфиги и логи могут оказаться в webroot сервера.

Если это так, то нужно запретить доступ ко всему, кроме директории `web`. Если на вашем хостинге такое невозможно, рассмотрите возможность смены хостинга.

9.4.6 Как избежать отладочной информации и утилит в продуктиве

В режиме отладки, Yii отображает довольно подробные ошибки, которые полезны во время разработки. Дело в том, что подробные ошибки удобны для нападающего, так как могут раскрыть структуру базы данных, параметров конфигурации и части вашего кода. Никогда не запускайте продуктивное приложение с `YII_DEBUG` установленным в `true` в вашем `index.php`.

Вы никогда не должны включать Gii на продуктиве. Он может быть использован для получения информации о структуре базы данных, кода и может позволить заменить файлы, генерируемые Gii автоматически.

Также следует избегать включения на продуктиве панели отладки, если только в этом нет острой необходимости. Она раскрывает всё приложение и детали конфигурации. Если вам все таки нужно запустить панель отладки на продуктиве, проверьте дважды что доступ ограничен только вашими IP-адресами.

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 её последующем извлечении. 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, можно изменить конфигурацию выше для использования APC `cache` следующим образом:

¹<http://memcached.org/>

```
'components' => [  
    'cache' => [  
        'class' => 'yii\caching\ApcCache',  
    ],  
],
```

Подсказка: 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 так, как это описано в `yii\caching\DbCache::$cacheTable`;
- `yii\caching\DummyCache`: является кэшем-пустышкой, не реализующим реального кэширования. Смысл этого компонента в упрощении кода, проверяющего наличие кэша. Вы можете использовать данный тип кэша и переключиться на реальное кэширование позже. Примеры: использование при разработке; если сервер не поддерживает кэш. Для извлечения данных в этом случае используется один и тот же код `Yii::$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);

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/>

- `yii\caching\WinCache`: использует расширение PHP WinCache⁶ (смотрите также⁷);
- `yii\caching\XCache`: использует расширение PHP XCache⁸;
- `yii\caching\ZendDataCache`: использует Zend Data Cache⁹.

Подсказка: 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. Если данные не найдены или устарели, то значение `false` будет возвращено;
- `set()`: сохраняет данные по ключу;
- `add()`: сохраняет данные по ключу если такого ключа ещё нет;
- `multiGet()`: извлекает сразу несколько элементов данных из кэша по заданным ключам;
- `multiSet()`: сохраняет несколько элементов данных. Каждый элемент идентифицируется ключом;
- `multiAdd()`: сохраняет несколько элементов данных. Каждый элемент идентифицируется ключом. Если ключ уже существует, то сохранения 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 возможно использовать при помощи `multiGet()` и `multiAdd()`. В

⁶<http://iis.net/downloads/microsoft/wincache-extension>

⁷<http://php.net/manual/en/book.wincache.php>

⁸<http://xcache.lighttpd.net/>

⁹http://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

случае, если хранилище не поддерживает эту функцию, она будет имитироваться.

Так как `yii\caching\Cache` реализует `ArrayAccess` - следовательно компонент кэша можно использовать как массив:

```
$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 также 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` используются следующие части для ключа:

```
[  
    __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, стоит указать префикс ключа во избежание конфликтов. 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 и могут быть удалены только 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 на то, 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, 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 `set()` метод, 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, построенная на основе кэширования данных. Она предназначена для кэширования результатов запросов к базе данных.

Кэширование запросов требует `DB connection` действительный `cache` компонента приложения. Предполагая, что `$db` это `yii\db\Connection` экземпляр, простое использование запросов кэширования происходит следующим образом:

```
$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 любого механизма на стороне сервера БД. КЗ описано разделом выше. Оно имеет безусловное преимущество, поскольку, благодаря ему, можно указать гибкие зависимости кэша и это более эффективно.

Очистка кэша

Для очистки всего кэша, вы можете вызвать `yii\caching\Cache::flush()`.

Также вы можете очистить кэш из консоли, вызвав `yii cache/flush`.

- `yii cache`: отображает список доступных кэширующих компонентов приложения
- `yii cache/flush cache1 cache2`: очищает кэш в компонентах `cache1`, `cache2` (можно передать несколько названий компонентов кэширования, разделяя их пробелом)

¹⁰<http://dev.mysql.com/doc/refman/5.1/en/query-cache.html>

- `yii cache/flush-all`: очищает кэш во всех кеширующих компонентах приложения

Информация: Консольное приложение использует отдельный конфигурационный файл по умолчанию. Для получения должного результата, убедитесь, что в конфигурациях консольного и веб-приложения у вас одинаковые компоненты кэширования.

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 кэширования результата. Для бесконечного кэша используйте 0. Именно это значение выставляется `yii\db\Connection::cache()`, если не указать время явно;
- `queryCache`: ID компонента кэширования. По умолчанию `'cache'`. Кэширование запросов работает только в том случае, 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'`, то 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()` для отключения кэширования 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 кэширования 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 РНР 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 РНР код.

Вы можете вызвать `yii\base\View::renderDynamic()` в пределах кэширования фрагмента для вставки динамического содержимого в нужное 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 РНР кода как параметр. Возвращаемое значение этого кода будет вставлено в динамическое содержимое. Этот РНР 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 [кэширования 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 закешировал её.

Вы можете настроить свойство `yii\filters\HttpCache::$lastModified`, чтобы включить отправку заголовка Last-Modified. Свойство должно содержать PHP-функцию, возвращающую временную метку UNIX timestamp времени последнего изменения страницы. Сигнатура PHP-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 закешированную 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 стра-

ницы 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 заголовков 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
```

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.

¹¹<http://tools.ietf.org/html/rfc7232#section-2.4>

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 распространённых 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 добавляют правило для контроллера `user`, которое предоставляет доступ к данным пользователя через красивые URL и логичные глаголы HTTP.

11.1.3 Включение JSON на прием данных

Для того чтобы API мог принимать данные в формате JSON, сконфигурируйте `parsers` свойство у компонента `request` [application component](#) на использование `yii\web\JsonParser` JSON данных на входе:

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

Примечание: Конфигурация, приведенная выше необязательна. Без приведенной выше конфигурации, API сможет определить только `application/x-www-form-urlencoded` и `multipart/form-data` форматы.

11.1.4 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 наверное 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 «Ресурсы».

11.1.5 Резюме

Используя Yii в качестве RESTful API фреймворка, мы реализуем точки 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

³<http://ru.wikipedia.org/wiki/Model-View-Controller>

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
```

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 в 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:

```

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"
        }
    }
}

```

⁴<http://en.wikipedia.org/wiki/HATEOAS>

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;
- **authenticator**: 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 запрошенному 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 вызывать 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 точку 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 **форматтера ответов**, регистрируемого с помощью компонента приложения `response`.

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
```

⁵http://en.wikipedia.org/wiki/Content_negotiation

```
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 *Асепт* 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 должны 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\Arrayable` и `yii\data\DataProviderInterface`. Для объектов ресурсов как правило реализуется интерфейс `yii\base\Arrayable`, 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
}
}

```

Настройка форматирования JSON

Ответ в формате JSON генерируется при помощи класса `JsonResponseFormatter`, который использует внутри хелпер `JSON`. Данный форматтер гибко настраивается. Например, опция `$prettyPrint` полезна на время разработки так как при её использовании ответы получаются более читаемыми. `$encodeOptions` может пригодиться для более тонкой настройки кодирования.

Свойство `formatters` компонента приложения `response` может быть сконфигурировано следующим образом:

```

'response' => [
    // ...
    'formatters' => [
        \yii\web\Response::FORMAT_JSON => [
            'class' => 'yii\web\JsonResponseFormatter',
            'prettyPrint' => YII_DEBUG, // используем "pretty" в режиме
            отладки
            'encodeOptions' => JSON_UNESCAPED_SLASHES |
            JSON_UNESCAPED_UNICODE,
            // ...
        ],
    ],
],

```

При работе с базой данных через `DAO` все данные представляются в виде строк, что не всегда корректно. Особенно учитывая, что в JSON для чисел есть соответствующий тип. При использовании `ActiveRecord` значения числовых столбцов приводятся к `integer` на этапе выборки из базы: `yii\db\ActiveRecord::populateRecord()`.

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.

⁶http://en.wikipedia.org/wiki/Basic_access_authentication

⁷<http://oauth.net/2/>

⁸<http://tools.ietf.org/html/rfc6750>

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.

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 каж-

дого 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.

Реализация в модели `User` может быть, например, такой:

```
public function getRateLimit($request, $action)
{
    return [$this->rateLimit, 1]; // $rateLimit запросов в секунду
}

public function loadAllowance($request, $action)
{
    return [$this->allowance, $this->allowance_updated_at];
}

public function saveAllowance($request, $action, $allowance, $timestamp)
{
    $this->allowance = $allowance;
    $this->allowance_updated_at = $timestamp;
    $this->save();
}
```

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.

Подсказка: Чтобы узнать больше о выборе версий обратитесь к Semantic Versioning⁹.

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`:

```
// как параметр  
Accept: application/json; version=v1  
// как тип содержимого, определенный поставщиком API  
Accept: application/vnd.company.myapp-v1+json
```

⁹<http://semver.org/>

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 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 закэшированную 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: Не поддерживаемый 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: <https://github.com/yiisoft/yii2-debug/blob/master/docs/>

Error: not existing file: <https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md>

Error: not existing file: <https://github.com/yiisoft/yii2-apidoc>

Глава 13

Тестирование

13.1 Тестирование

Тестирование является важной составляющей разработки программного обеспечения. Мы проводим тестирование непрерывно, осознаем мы это или нет. Например, когда мы пишем класс на языке PHP, мы можем отлаживать его шаг за шагом или просто использовать `echo` или `die` для проверки, что реализация работает в соответствии с намеченным планом. В случае веб приложения, мы вводим некоторые тестовые данные в форму для того, чтобы убедиться, что страница взаимодействует с нами, как ожидается.

Процесс тестирования может быть автоматизирован так, что каждый раз когда нам нужно что-то проверить, мы просто должны вызвать код, который сделает это за нас. Код, который проверяет, что результат совпадает с тем, что мы планировали, называется тестом, а процесс создания тестов и их последующего использования - автоматизированным тестированием, что и является главной темой данного раздела.

13.1.1 Разработка с тестами

Разработка через тестирование (TDD) и разработка через поведение (BDD) - это подходы разработки программного обеспечения, в рамках которых поведение части кода или целая фича описывается в виде набора сценариев или тестов ДО написания фактического кода и только затем создается реализация. Тем самым мы можем использовать данные тесты для проверки, что достигается заданное поведение.

Процесс разработки фичи следующий:

- Создать новый тест, который описывает функцию, которая будет реализована.
- Запустить новый тест и убедиться, что он терпит неудачу. Это ожидаемо, т.к. на данный момент еще нет конкретной реализации.
- Написать простой код, чтобы новый тест отрабатывал без ошибок.

- Запустить все тесты и убедиться, что они отрабатывают без ошибок
- Улучшить код и убедиться, что все тесты все еще отрабатывают без ошибок

После того как это завершено процесс повторяется снова для другой фичи или улучшения. Если существующая фича должна быть изменена, то и тесты также должны быть изменены.

Подсказка: Если вы чувствуете, что вы теряете время выполняя много мелких и простых итераций, попробуйте покрыть это вашим тестовым сценарием перед следующим выполнением тестов. Если вы слишком много отлаживаете, попробуйте сделать обратное.

Написание тестов до реализации конкретного функционала позволяет нам сосредоточиться на том, что мы хотим достичь и полностью погрузиться в “как это сделать” впоследствии.

Обычно это приводит к лучшим абстракциям и более легкой поддержке тестов, когда речь идет о корректировке фичи или уменьшении связанности компонентов.

Таким образом плюсы этого подхода следующие:

- Позволяет вам сосредоточиться на одной вещи, что в свою очередь приводит к улучшению планирования и реализации.
- Более подробное покрытие тестами функционала, таким образом, если все тесты отрабатывают без ошибок, скорее всего, ничего не сломано.

В долгосрочной перспективе это, как правило, дает вам хороший эффект экономии времени.

Подсказка: Если вы хотите узнать больше о принципах сбора требования программного обеспечения и моделирования предметной области, рекомендуем изучить Проблемно-ориентированное проектирование (DDD)¹.

13.1.2 Когда и как тестировать

Принцип разработки описанный выше имеет смысл применять для долгосрочных и относительно сложных проектов, в то время как для простых это может быть излишним. Есть несколько показателей того, когда данный подход уместен:

- Проект уже большой и сложный.
- Требования к проекту начинают усложняться. Проект постоянно растет.
- Долгосрочный проект.

¹https://en.wikipedia.org/wiki/Domain-driven_design

- Цена ошибки очень высока

Нет ничего плохого в создании тестов, покрывающих поведение существующей реализации.

- Legасу-проект который постоянно обновляется.
- Вам поручили работу над проектом, в котором нет ни одного теста.

В некоторых случаях автоматизированно тестирование может быть излишним:

- Проект простой и не станет более сложным.
- Это одноразовый проект, который больше не будет дорабатываться.

Тем не менее, если у вас есть время, было бы хорошо автоматизировать тестирование и в этих случаях.

13.1.3 Что почитать

- Экстремальное программирование. Разработка через тестирование / Кент Бек. ISBN: 0321146530.

13.2 Настройка тестового окружения

Примечание: Данный раздел находится в разработке.

Yii 2 официально поддерживает интеграцию с фреймворком для тестирования Codeception², который позволяет вам проводить следующие типы тестов:

- **Модульное тестирование** - проверяет что отдельный модуль кода работает верно;
- **Функциональное тестирование** - проверяет пользовательские сценарии через эмуляцию браузера;
- **Приёмочное тестирование** - проверяет пользовательские сценарии в браузере.

Все три типа тестов представлены в шаблонах проектов yii2-basic³ и yii2-advanced⁴.

Для того, чтобы запустить тесты необходимо установить Codeception⁵. Сделать это можно как локально, то есть только для текущего проекта, так и глобально для компьютера разработчика.

Для локальной установки используйте следующие команды:

```
composer require "codeception/codeception=2.1.*"
composer require "codeception/specify=*"
composer require "codeception/verify=*
```

²<https://github.com/Codeception/Codeception>

³<https://github.com/yiisoft/yii2-app-basic>

⁴<https://github.com/yiisoft/yii2-app-advanced>

⁵<https://github.com/Codeception/Codeception>

Для глобальной установки необходимо добавить директиву `global`:

```
composer global require "codeception/codeception=2.1.*"  
composer global require "codeception/specify=*"   
composer global require "codeception/verify=*" 
```

Если вы никогда не пользовались Composer для установки глобальных пакетов, запустите `composer global status`. На выходе вы должны получить:

```
Changed current directory to <directory>
```

Затем `<directory>/vendor/bin` добавьте в переменную окружения `PATH`. После этого можно использовать `codecept` глобально из командной строки.

Примечание: глобальная установка позволяет вам использовать Codeception для всех проектов на компьютере разработчика путём запуска команды `codecept` без указания пути. Тем не менее, данный подход может не подойти. К примеру, в двух разных проектах может потребоваться установить разные версии Codeception. Для простоты все команды в разделах про тестирование используются так, будто Codeception установлен глобально.

13.3 Модульные тесты

Примечание: Данный раздел находится в разработке.

Модульный тест проверяет что отдельный модуль кода работает верно. В ООП самым базовым модулем является класс. То есть модульный тест проверяет все методы интерфейса класса. На вход подаются различные параметры и тест проверяет, что методы возвращают ожидаемые значения. Модульные тесты обычно пишутся тем же, кто реализует тестируемый класс.

Модульное тестирование в Yii использует PHPUnit и, опционально, Codeception. Рекомендуется проверить его документацию:

- Документация PHPUnit начиная с главы 2⁶.
- Codeception Unit Tests⁷.

13.3.1 Запуск тестов шаблонов проектов basic и advanced

Следуйте инструкциям в `apps/advanced/tests/README.md` и `apps/basic/tests/README.md`.

⁶<http://phpunit.de/manual/current/en/writing-tests-for-phpunit.html>

⁷<http://codeception.com/docs/05-UnitTests>

13.3.2 Модульные тесты фреймворка

Если вам необходимо запустить набор модульных тестов для самого Yii, прочитайте “Подготовка к разработке Yii 2⁸”.

13.4 Функциональные тесты

Примечание: Данный раздел находится в разработке.

- Codeception Functional Tests⁹

13.4.1 Запуск функциональных тестов для шаблонов проектов basic и advanced

Следуйте инструкциям в `apps/advanced/tests/README.md` и `apps/basic/tests/README.md`.

13.5 Приёмочное тестирование

Примечание: Данный раздел находится в разработке.

- Codeception Acceptance Tests¹⁰

13.5.1 Запуск тестов в шаблонах проектов basic и advanced

Инструкции приведены в `apps/advanced/tests/README.md` и `apps/basic/tests/README.md`.

13.6 Фикстуры

Фикстуры (англ. fixtures) - это важная составляющая тестирования. Их основная задача заключается в подготовке окружения с заранее фиксированным/известным состоянием для гарантии повторяемости процесса тестирования. Yii предоставляет фреймворк, который позволяет легко и точно определять фикстуры и использовать их в ваших тестах.

Ключевым понятием в фреймворке фикстур Yii является так называемый *объект фикстуры*. Объект фикстуры представляет собой особый аспект тестового окружения, который наследуется от `yii\test\Fixture` или его наследников. Например, вы можете использовать `UserFixture` для

⁸<https://github.com/yiisoft/yii2/blob/master/docs/internals-ru/getting-started.md>

⁹<http://codeception.com/docs/04-FunctionalTests>

¹⁰<http://codeception.com/docs/03-AcceptanceTests>

того, чтобы быть уверенным, что таблица пользователей содержит известный набор данных. Вы загружаете один или несколько объектов фикстур перед запуском теста и выгружаете их после его завершения.

Фикстура может зависеть от других фикстур, заданных через свойство `yii\test\Fixture::$depends`. Когда фикстура загружается, фикстуры от которых она зависит будут автоматически загружены ДО нее, а когда она выгружается все зависимые фикстуры будут выгружены ПОСЛЕ нее.

13.6.1 Объявление фикстуры

Для объявления фикстуры создайте новый класс унаследованный от `yii\test\Fixture` или `yii\test\ActiveFixture`. Первый лучше всего подходит для фикстур общего назначения, в то время как последний имеет расширенные функции, специально предназначенные для работы с базой данных и `ActiveRecord`.

Следующий код показывает как объявить фикстуру для модели `ActiveRecord` `User`, которая соответствует таблице пользователей.

```
<?php
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture
{
    public $modelClass = 'app\models\User';
}
```

Подсказка: каждая `ActiveFixture` предназначена для подготовки таблицы базы данных для тестирования. Вы можете указать таблицу как через свойство `yii\test\ActiveFixture::$tableName`, так и через свойство `yii\test\ActiveFixture::$modelClass`. Если последнее, то в этом случае имя таблицы будет взято из модели `ActiveRecord`, указанной в `modelClass`.

Примечание: `yii\test\ActiveFixture` используется только для реляционных баз данных. Для NoSQL решений Yii предоставляет следующие классы `ActiveFixture`:

- Mongo DB: `yii\mongodb\ActiveFixture`
- Elasticsearch: `yii\elasticsearch\ActiveFixture` (начиная с версии 2.0.2)

Данные для фикстуры `ActiveFixture` как правило находятся в файле `FixturePath/data/TableName.php`, где `FixturePath` указывает на директорию в которой располагается файл

класса фикстуры, а `tableName` имя таблицы с которой она ассоциируется. Для примера выше, данные должны быть в файле `@app/tests/fixtures/data/user.php`. Данный файл должен вернуть массив данных для строк, которые будут вставлены в таблицу пользователей. Например

```
<?php
return [
    'user1' => [
        'username' => 'lmayer',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/
iK0r3jRuWQEs2ldRu.a2',
    ],
    'user2' => [
        'username' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphh0h9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6
viYG5xJExU6',
    ],
];
```

Вы можете задать псевдоним строке для того, чтобы в будущем вы могли ссылаться на нее в ваших тестах. В примере выше 2 строки имеют псевдонимы `user1` и `user2`, соответственно.

Также вам не нужно указывать данные для столбцов с авто-инкрементом. Yii автоматически заполнит значения данных столбцов в момент загрузки фикстуры.

Подсказка: вы можете указать свой путь до файла данных через свойство `yii\test\ActiveFixture::$dataFile`. Вы также можете переопределить метод `yii\test\ActiveFixture::getData()`, чтобы предоставить данные.

Как мы описали ранее, фикстура может зависеть от других фикstur. Например, для `UserProfileFixture` возможно потребуется зависимость от `UserFixture` так как таблица пользовательских профилей содержит внешний ключ, указывающий на таблицу пользователей. Зависимость указывается через свойство `yii\test\Fixture::$depends`, как в следующем примере

```
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserProfileFixture extends ActiveFixture
{
    public $modelClass = 'app\models\UserProfile';
    public $depends = ['app\tests\fixtures\UserFixture'];
}
```

Зависимость также гарантирует, что фикстуры загружаются и выгружаются в определенном порядке. В предыдущем примере `UserFixture` будет автоматически загружена до `UserProfileFixture`, тем самым гарантируя существование всех внешних ключей, и будет выгружена после того как выгрузится `UserProfileFixture` по тем же причинам.

Выше мы показали как объявить фикстуру для таблицы базы данных. Для объявления фикстуры не связанной с базой данных (например, фикстуры для определенных файлов и директорий), вам следует унаследовать ее от класса `yii\test\Fixture` и переопределить методы `load()` и `unload()`.

13.6.2 Использование фикстур

Если вы используете Codeception¹¹ для тестирования вашего кода, вам следует рассмотреть вопрос об использовании расширения `yii2-codeception`, которое имеет встроенную поддержку загрузки фикстур и доступа к ним. Если вы используете другой фреймворк для тестирования, вы можете использовать `yii\test\FixtureTrait` в ваших тестах для этих целей.

Далее мы опишем как написать класс модульного тестирования для модели `UserProfile` с использованием расширения `yii2-codeception`.

Объявите какие фикстуры вы хотите использовать в методе `fixtures()` вашего класса модульного тестирования, унаследованного от `yii\codeception\DbTestCase` или `yii\codeception\TestCase`. Например,

```
namespace app\tests\unit\models;

use yii\codeception\DbTestCase;
use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends DbTestCase
{
    public function fixtures()
    {
        return [
            'profiles' => UserProfileFixture::className(),
        ];
    }

    // методы... тестирования...
}
```

Фикстуры перечисленные в методе `fixtures()` будут автоматически загружены перед выполнением каждого метода тестирования тест-кейса и выгружены после завершения каждого метода тестирования. И, как мы описали ранее, когда фикстура загружается, все зависимые от нее

¹¹<http://codeception.com/>

фикстуры будут автоматически загружены в первую очередь. В приведенном выше примере, при выполнении любого метода тестирования в тест-кейсе последовательно будут загружены две фикстуры: `UserFixture` и `UserProfileFixture`, поскольку `UserProfileFixture` зависит от `UserFixture`.

Для определения фикстур в методе `fixtures()` вы можете использовать либо имя класса, либо массив настроек. С помощью массива настроек вы можете настроить свойства фикстуры, которые будут установлены при ее загрузке.

Вы также можете назначить фикстуре псевдоним. В примере выше, `profiles` является псевдонимом фикстуры `UserProfileFixture`. С помощью псевдонима вы можете получить объект фикстуры в ваших методах тестирования. Например, `$this->profiles` вернет объект `UserProfileFixture`.

Поскольку `UserProfileFixture` наследуется от `ActiveFixture`, вы можете также использовать следующий синтаксис для доступа к данным фикстуры:

```
// вернет строку данных для псевдонима 'user1'
$row = $this->profiles['user1'];
// вернет модель UserProfile, соответствующую строке данных для псевдонима '
  user1'
$profile = $this->profiles('user1');
// обход данных фикстуры в цикле
foreach ($this->profiles as $row) ...
```

Информация: `$this->profiles` продолжает быть объектом класса `UserProfileFixture`. Указанные особенности доступа реализуются через магические методы PHP.

13.6.3 Определение и использование глобальных фикстур

Фикстуры, описанные выше, в основном используются в рамках определенных тест-кейсов. В большинстве случаев, вам также нужны глобальные фикстры, которые применяются во ВСЕХ или большинстве тест-кейсов. Примером является фикстура `yii\test\InitDbFixture`, которая делает 2 вещи:

- Запускает скрипт `@app/tests/fixtures/initdb.php` для выполнения ряда общих задач инициализации тестового окружения;
- Отключает проверку целостности данных перед загрузкой остальных фикстур, и включает ее обратно после того как все остальные фикстуры будут выгружены.

Использование глобальных фикстур схоже с использованием не глобальных. Единственное отличие в том, что вы должны объявить эти фикстуры в методе `yii\codeception\TestCase::globalFixtures()`, а не `fixtures()`. Когда тест-кейс загружает фикстуры, сначала загружаются глобальные фикстуры, затем все остальные.

По умолчанию, фикстура `InitDbFixture` уже объявлена в методе `globalFixtures()` класса `yii\codeception\DbTestCase`. Это означает, что вы должны работать только с файлом `@app/tests/fixtures/initdb.php`, если вы хотите чтобы перед каждым тестом выполнялись определенные подготовительные работы. В противном случае вы просто можете сфокусироваться на разработке конкретных тест-кейсов и соответствующих фикстур.

13.6.4 Организация классов фикстур и файлов с данными

По умолчанию, классы фикстур ищут соответствующие файлы данных в директории `data`, которая является подпапкой папки, содержащей файлы классов фикстур. Вы можете следовать этому соглашению при работе над простыми проектами. Есть вероятность, что на больших проектах вам потребуется менять набор данных для одного и того же класса фикстур в разных тестах. Таким образом, мы рекомендуем вам организовать файлы данных иерархически, подобно пространству имен ваших классов. Например,

```
# в папке tests\unit\fixtures

data\
  components\
    fixture_data_file1.php
    fixture_data_file2.php
    ...
    fixture_data_fileN.php
  models\
    fixture_data_file1.php
    fixture_data_file2.php
    ...
    fixture_data_fileN.php
# и так далее
```

Таким образом вы избежите коллизий файлов данных фикстур между тестами и будете использовать их, как вам нужно.

Примечание: в примере выше файлы данных фикстур названы так только в качестве примера. В реальной жизни вам следует называть их в соответствии с тем от какого класса наследуется ваш класс фикстуры. Например, при наследовании от `yii\test\ActiveFixture` для фикстур БД вам следует использовать имя таблицы в качестве имени файла данных; при наследовании от `yii\mongodb\ActiveFixture` для фикстур MongoDB вам следует использовать имя коллекции в качестве имени файла.

Вы можете использовать похожую иерархию для организации файлов классов фикстур. Чтобы избежать конфликта с файлами данных вы

можете использовать в качестве корневой директории `fixtures` вместо `data`.

13.6.5 Резюме

Примечание: Этот раздел находится в разработке.

Выше мы описали как объявлять и использовать фикстуры. Ниже приведен типовой сценарий выполнения модульных тестов, связанных с БД:

1. Используйте команду `yii migrate` для обновления тестовой БД до последней версии;
2. Выполнить тест-кейс:
 - Загрузка фикстур: очищение соответствующих таблиц БД и заполнение их данными фикстур;
 - Выполнение теста;
 - Выгрузка фикстур.
3. Повторение шага 2 до тех пор, пока не выполнятся все тесты.

Будет доработано

13.7 Управление фикстурами

Примечание: Данный раздел находится в разработке.

todo: данный раздел может быть объединен с предыдущими частями `test-fixtures.md`

Фикстуры являются важной составляющей тестирования. Их основная задача в предоставлении набора данных, необходимого для тестирования различных сценариев работы вашего приложения. С этими данными использование ваших тестов становятся более эффективным и полезным.

Yii поддерживает фикстуры через утилиту командной строки `yii fixture`. Эта утилита поддерживает:

- Загрузку фикстур в различные хранилища, такие как: RDBMS, NoSQL и другие;
- Выгрузку фикстур разными способами (как правило очищает хранилище);
- Автоматическую генерацию фикстур и наполнение их случайными данными

13.7.1 Формат фикстуры

Фикстуры - это объекты с различными методами и конфигурацией, с которыми вы можете ознакомиться в официальной документации¹².

Давайте предположим, что у нас есть следующий набор данных фикстуры для загрузки:

```
# файл users.php в директории файлов данных фикстур, по умолчанию @tests\
unit\fixtures\data

return [
    [
        'name' => 'Chase',
        'login' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-0U8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/
iK0r3jRuwQEs2ldRu.a2',
    ],
    [
        'name' => 'Celestine',
        'login' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphh0h9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6
viYG5xJExU6',
    ],
];
```

Если вы используете фикстуру, которая загружает данные в базу данных, то эти строки будут применены к таблице `users`. Если вы используете фикстуру для загрузки данных в `posql`, например, фикстура для `mongodb`, то данные будут применены к коллекции `users`. Для того, чтобы узнать о реализации различных сценариях загрузки фикстур, обратитесь к официальной документации¹³. Предыдущий пример фикстуры был сгенерирован автоматически с использованием расширения `yii2-faker`, подробнее про это читайте в этом разделе. Имя класса фикстуры должно быть в единственном числе.

13.7.2 Загрузка фикстур

Класс фикстур должны содержать суффикс `Fixture`. По умолчанию поиск фикстур выполняется в пространстве имен `tests\unit\fixtures`, но вы можете изменить это поведение через конфигурационный файл или параметры команды. Вы можете исключить некоторые фикстуры из загрузки или выгрузки добавив - перед их именем, например `-User`.

Чтобы загрузить фикстуру, выполните следующую команду::

¹²<https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixture.md>

¹³<https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixture.md>


```
yii fixture/load <fixture_name>
```

Обязательный параметр `fixture_name` указываем на имя фикстуры, которая должна быть загружена. Вы можете загрузить несколько фикстур за раз. Ниже указаны примеры корректного использования данной команды:

```
// загрузить фикстуру 'User'
yii fixture/load User

// тоже что и выше, тк.. "load" является действие по умолчанию для команды "
    fixture"
yii fixture User

// загрузить нескольких фикстур
yii fixture "User, UserProfile"

// загрузить все фикстуры
yii fixture/load "*"

// тоже что и выше
yii fixture "*"

// загрузить все фикстуры кроме указанной
yii fixture "*, -DoNotLoadThisOne"

// загрузка фикстур, но искать их следует в другом пространстве имен.
    Пространство имен по умолчанию: tests\unit\fixtures.
yii fixture User --namespace='alias\my\custom\namespace'

// загрузить глобальную фикстуру 'some\name\space\CustomFixture' перед
    загрузкой остальных фикстур.
// По умолчанию данный параметр установлен в 'InitDbFixture' для
    включения/отключения/ проверки целостности данных.
// Вы можете задать несколько глобальных фикстур, указав их через запятую
yii fixture User --globalFixtures='some\name\space\Custom'
```

13.7.3 Выгрузка фикстур

Для выгрузки фикстур выполните следующую команду:

```
// выгрузить фикстуру 'Users', по умолчанию будут удалены все данные из
    таблицы "users", или из коллекции "users" если это фикстура mongodb
yii fixture/unload User

// выгрузить несколько фикстур
yii fixture/unload "User, UserProfile"

// выгрузить все фикстуры
yii fixture/unload "*"

// выгрузить все фикстуры за исключением указанной
yii fixture/unload "*, -DoNotUnloadThisOne"
```

При выгрузке фикстур вы также можете использовать параметры `namespace` и `globalFixtures`.

13.7.4 Глобальная настройка команды

Хотя параметры командой строки и позволяют нам настраивать команду миграции на лету, иногда нам может понадобиться настроить команду один раз для всех сценариев запуска. Например, вы можете настроить различные пути до файлов с фикстурами как в примере ниже:

```
'controllerMap' => [  
    'fixture' => [  
        'class' => 'yii\console\controllers\FixtureController',  
        'namespace' => 'myalias\some\custom\namespace',  
        'globalFixtures' => [  
            'some\name\space\Foo',  
            'other\name\space\Bar'  
        ],  
    ],  
]
```

13.7.5 Автоматическая генерация фикстур

Yii также может автоматически генерировать для вас фикстуры на основе некоторого шаблона. Вы можете генерировать фикстуры с различным набором данных на разных языках и в разных форматах. Данная возможность основана на использовании библиотеки `Faker`¹⁴ и расширения `yii2-faker`.

Для получения дополнительной информации ознакомьтесь с руководством¹⁵.

¹⁴<https://github.com/fzaninotto/Faker>

¹⁵<https://github.com/yiisoft/yii2-faker>

Глава 14

Специальные темы

Error: not existing file: <https://github.com/yiisoft/yii2-app-advanced/blob/master>

14.1 Создание своей структуры приложения

Примечание: Этот раздел находится на стадии разработки.

Пока шаблоны проектов `basic`¹ и `advanced`² великолепно справляются с большинством ваших потребностей, но вы можете захотеть создать свой собственный шаблон проекта, с которого будете начинать делать ваши проекты.

Шаблоны проектов в Yii - это просто репозитории, содержащие `composer.json` файл, и зарегистрированные как Composer пакет. Любые репозитории, которые могут быть определены как Composer пакеты, становятся установочными через Composer команду `create-project`.

Чтобы построить весь свой шаблон с нуля, нужно затратить много энергии, поэтому лучше использовать один из встроенных шаблонов, как базовый.

14.1.1 Клонирование базового шаблона

Первый шаг для клонирования базового Yii шаблона из Git репозитория:

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

Затем необходимо подождать, чтобы репозитории загрузился на ваш компьютер. С внесенными изменениями шаблон должен быть “запушен”(push) обратно, затем вы можете удалить `.git` директорию и весь загруженный контент на вашем компьютере.

14.1.2 Измените файлы

Следующее, вам надо изменить `composer.json` в соответствии с вашим шаблоном. Измените значения `name`(имя), `description`(описание), `keywords`(ключевые слова), `homepage`(адрес домашней страницы), `license`(лицензия), и `support`(поддержка) для описания вашего нового шаблона. Также установите `require`(зависимости фреймворка), `require-dev`(зависимости от расширений), `suggest`, и другие опции, которые необходимы для вашего шаблона.

Примечание: В файле `composer.json` используйте `writable` параметр внутри `extra`, чтобы указать права доступа к файлам, которые будут установлены, после создания приложения на основе данного шаблона.

Следующее, измените структуру и содержание приложения, по вашему вкусу. В заключении обновите README файл, чтобы он соответствовал конечному варианту вашего шаблона.

¹<https://github.com/yiisoft/yii2-app-basic>

²<https://github.com/yiisoft/yii2-app-advanced>

14.1.3 Создание пакета

Создайте Git репозиторий из созданного шаблона, и запустите(push) его. Если вы собираетесь сделать ваш шаблон с открытым исходным кодом, Github³ - это лучшее место, чтобы разместить его. Если вы собираетесь сохранить ваш шаблон для личных целей, используйте любой Git, предназначенный для этих целей.

Затем, вам необходимо зарегистрировать ваш пакет в Composer. Пакет с публичным шаблоном должен быть зарегистрирован на Packagist⁴. Для частных шаблонов, зарегистрировать шаблона немного сложнее. Для получения инструкции загляните в Composer documentation⁵.

14.1.4 Использование шаблона

Это все, что требуется для создания нового шаблона проекта в Yii. Сейчас вы можете создавать проекты, использующие ваш шаблон:

```
composer global require "fxp/composer-asset-plugin:~1.2.0"
composer create-project --prefer-dist --stability=dev mysoft/yii2-app-
coolone new-project
```

14.2 Консольное приложение

Кроме богатых возможностей для построения веб приложений, Yii также имеет полноценную поддержку консольных приложений, которые обычно используются для создания фоновых и служебных задач, поддерживающих сайт.

Структура консольных приложений очень похожа на структуру веб приложения. Она состоит из одного и более классов `yii\console\Controller`, которые часто называют командами в консольной среде. Каждый контроллер может иметь одно или более действий, как и веб контроллеры.

В обоих шаблонах проектов уже есть консольное приложение. Вы можете запустить его, вызвав скрипт `yii`, который находится в основной директории вашего приложения. Вы получите список доступных команд, если вызовете его без параметров:

³<http://github.com>

⁴<https://packagist.org/>

⁵<https://getcomposer.org/doc/05-repositories.md#hosting-your-own>

```
(lamb) php yii
This is Yii version 2.0.8-dev.

The following commands are available:

- asset
  asset/compress (default) Allows you to combine and compress your JavaScript and CSS files.
  asset/template           Combines and compresses the asset files according to the given configuration.
                           Creates template of configuration file for [[actionCompress]].

- cache
  cache/flush             Allows you to flush cache.
  cache/flush-all        Flushes given cache components.
  cache/flush-schema      Flushes all caches registered in the system.
  cache/index (default)   Clears DB schema cache for a given connection component.
                           Lists the caches that can be flushed.

- fixture
  fixture/load (default)  Manages fixture data loading and unloading.
  fixture/unload          Loads the specified fixture data.
                           Unloads the specified fixtures.

- help
  help/index (default)    Provides help information about console commands.
                           Displays available commands or the detailed information

- message
  message/config          Extracts messages to be translated from source files.
  message/config-template Creates a configuration file for the "extract" command using command line options specified
  message/extract (default) Extracts messages to be translated from source code.

- migrate
  migrate/create          Manages application migrations.
  migrate/down            Creates a new migration.
  migrate/history         Downgrades the application by reverting old migrations.
  migrate/mark            Displays the migration history.
  migrate/new             Modifies the migration history to the specified version.
  migrate/redo            Displays the un-applied new migrations.
  migrate/to              Redoes the last few migrations.
  migrate/up (default)    Upgrades or downgrades till the specified version.
                           Upgrades the application by applying new migrations.

- serve
  serve/index (default)   Runs PHP built-in web server
                           Runs PHP built-in web server

To see the help of each command, enter:

yii help <command-name>
```

Как вы можете видеть на скриншоте, в Yii уже определён набор доступных по умолчанию команд:

- **AssetController** - Позволяет вам объединять и сжимать ваши JavaScript и CSS файлы. Больше об этой команде вы можете узнать в [Assets Section](#).
- **CacheController** - Позволяет вам сбрасывать кеш приложения.
- **FixtureController** - Управляет загрузкой и выгрузкой данных фикстур для тестирования. Данная команда более подробно описана в [Testing Section about Fixtures](#).
- **HelpController** - Обеспечивает справочную информацию о консольных командах, это команда по умолчанию и она печатает текст, который вы видели выше.
- **MessageController** - Извлекает сообщения для перевода из файлов с исходными тестами. Больше об этой команде вы можете узнать в [I18N Section](#).
- **MigrateController** - Управление миграциями приложения. Миграции базы данных более детально описаны в [Database Migration Section](#).
- **ServeController** - Позволяет запускать встроенный вебсервер PHP.

14.2.1 Использование

Вы можете запустить действие консольного контроллера, используя следующий синтаксис:

```
yii <route> [--option1=value1 --option2=value2 ... argument1 argument2 ...]
```

В приведённом выше примере, <route> относится к действию контроллера. Параметры будут подставляться в свойства класса и в аргументы метода действия.

Для примера, `MigrateController::actionUp()` с `MigrateController::$migrationTable` установкой `migrations` и лимитом в 5 миграций может быть вызвано следующим образом:

```
yii migrate/up 5 --migrationTable=migrations
```

Примечание: При использовании в консоли *, не забудьте поместить её в кавычки "*" чтобы избежать её интерпретации и замены на все имена файлов в данной директории.

14.2.2 Входной скрипт

Входной скрипт консольного приложения - это подобие файла `index.php`, используемого в веб приложении. Входной скрипт консоли, как правило, называется `yii` и располагается в основной директории приложения. Он содержит код похожий на следующее:

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */

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

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

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

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

Этот скрипт будет создан как часть вашего приложения; вы можете его редактировать, если вам это необходимо. `YII_DEBUG` можете установить в `false` если вам не нужно видеть отладочный вывод при ошибке, и/или если вы хотите улучшить общую производительность. В обоих шаблонах приложения, во входном скрипте приложения отладка включена по умолчанию для обеспечения более дружелюбного к разработчику окружения.

14.2.3 Настройка

Как видно из приведённого выше кода, консольное приложение использует свой собственный файл конфигурации, названный `console.php`. В этом файле вы должны произвести настройку различных **компонентов приложения** и свойств консольного приложения.

Если ваше веб и консольное приложение имеет много общих параметров конфигурации, вы можете выделить общую часть в отдельный файл, и включить его в оба файла конфигурации (веб и консоль). Вы можете посмотреть пример в “продвинутом” шаблоне проекта.

Подсказка: Иногда, вам может потребоваться запустить консольную команду используя конфигурацию, отличную от той, что указано во входном скрипте. Для примера, вы можете использовать команду `yii migrate` для обновления тестовой базы данных, которая настраивается для каждого отдельного набора тестов. Для изменения файла конфигурации, просто укажите свой конфигурационный файл через опцию `appconfig` при запуске команды:

```
yii <route> --appconfig=path/to/config.php ...
```

14.2.4 Создание ваших собственных команд

Консольный контроллер и действие

Консольная команда определяется как класс контроллера расширяющий `yii\console\Controller`. В классе контроллера, вы определяете одно или несколько действий, которые соответствуют суб-командам контроллера. В каждом действии вы пишете код, который реализует соответствующие данной суб-команде задачи.

При запуске команды, вам необходимо указать маршрут к действию. Например, маршрут `migrate/create` вызывает суб-команду, которая соответствует методу `MigrateController::actionCreate()`. Если маршрут, предложенный при вызове команды, не содержит указания идентификатора действия, будет вызвано действие по умолчанию (так же как и в веб приложении).

Опции

Для переопределения `yii\console\Controller::options()` метода, вы можете указать опции, которые доступны в консольной команде (`controller/actionID`). Метод должен возвращать список публичных атрибутов класса. При запуске команды вы можете указать значение опций, используя синтаксис `--OptionName=OptionValue`. Это свяжет `OptionValue` с атрибутом `OptionName` класса контроллера.

Если значение по умолчанию опции - это массив, то при установке этой опции, при выполнении команды, значение будет преобразовано в массив путём разделения входящей строки по запятым.

Аргументы

Кроме опций, команда может получать аргументы. Аргументы будут переданы в качестве параметров в метод действия, соответствующего запрошенной суб-команде. Первый аргумент соответствует первому параметру, второй соответственно второму, и так далее. Если переданных аргументов при вызове команды будет недостаточно, то параметрам будут назначены по умолчанию, если они определены. Если значения по умолчанию не определены, и не были переданы, команда завершит выполнение с ошибкой.

Вы можете использовать указание типа `array`, чтобы указать, что аргумент должен рассматриваться как массив. Массив будет сгенерирован путём разделения входной строки по запятым.

Псевдонимы опций

Начиная с версии 2.0.8 в классе консольной команды доступен метод `yii\console\Controller::optionAliases()`, позволяющий добавлять псевдонимы для опций.

Для того, чтобы задать псевдоним, переключите метод `yii\console\Controller::optionAliases()` в вашем контроллере:

```
namespace app\commands;

use yii\console\Controller;

class HelloController extends Controller
{
    public $message;

    public function options()
    {
        return ['message'];
    }

    public function optionAliases()
    {
        return ['m' => 'message'];
    }

    public function actionIndex()
    {
        echo $this->message . "\n";
    }
}
```

Теперь для запуска команды можно использовать следующий синтаксис:

```
yii hello -m=hello
```

Следующий пример показывает как описывать аргументы:

```
class ExampleController extends \yii\console\Controller
{
    // Команда "yii example/create test" вызовет "actionCreate('test')"
    public function actionCreate($name) { ... }

    // Команда "yii example/index city" вызовет "actionIndex('city', 'name')"
    // Команда "yii example/index city id" вызовет "actionIndex('city', 'id')"
    public function actionIndex($category, $order = 'name') { ... }

    // Команда "yii example/add test" вызовет "actionAdd(['test'])"
    // Команда "yii example/add test1,test2" вызовет "actionAdd(['test1', 'test2'])"
    public function actionAdd(array $name) { ... }
}
```

Код возврата

При разработке консольного приложения принято использовать код возврата. Принято, код 0 означает, что команда выполнена удачно. Если команда вернула код больше нуля, то это говорит об ошибке. Номер, который был возвращён при ошибке, потенциально может быть использован для поиска более детальной информации об ошибке. Для примера 1 может указывать на неизвестную ошибку, а все коды выше могут быть зарезервированы под специфичные ошибки: ошибки ввода, повреждённые файлы, и что-то другое.

Для того, чтобы ваша консольная команда возвращала код возврата, просто верните целое число в методе действия контроллера:

```
public function actionIndex()
{
    if (/* возникла проблема */) {
        echo "Возникла проблема!\n";
        return 1;
    }
    // делаем что-нибудь
    return 0;
}
```

Есть несколько предопределённых констант, которые вы можете использовать:

- `Controller::EXIT_CODE_NORMAL` со значением 0;
- `Controller::EXIT_CODE_ERROR` со значением 1.

Хорошая практика, определять значимые для вашего контроллера константы в случае, если вы используете больше типов ошибок.

Форматирование и цвета

Консоль Yii поддерживает форматирование вывода, который автоматически деградирует до не форматированного, если это не поддерживается в терминале, где запускается команда.

Вывод форматированных строк прост. Вот как можно вывести некоторый жирный текст:

```
$this->stdout("Hello?\n", Console::BOLD);
```

Если вам нужно собрать строку динамически объединяя несколько стилей, лучше использовать `ansiFormat()`:

```
$name = $this->ansiFormat('Alex', Console::FG_YELLOW);  
echo "Hello, my name is $name.";
```

14.3 Встроенные валидаторы

Yii предоставляет встроенный набор часто используемых валидаторов, расположенных, в первую очередь, в пространстве имен `yii\validators`. Вместо того, чтобы использовать длинные имена классов валидаторов, вы можете использовать *псевдонимы*, чтобы указать на использование этих валидаторов. Например, вы можете использовать псевдоним `required`, чтобы сослаться на класс `yii\validators\RequiredValidator`:

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

Все поддерживаемые псевдонимы валидаторов можно увидеть в свойстве `yii\validators\Validator::$builtInValidators`.

Ниже мы опишем основные способы использования и свойства всех встроенных валидаторов.

14.3.1 boolean

```
[  
    // Проверяет 'selected' на равенство 0 или 1, без учета типа данных  
    ['selected', 'boolean'],  
  
    // Проверяет, что "deleted" - это тип данных boolean и содержит true  
    // или false  
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false, '  
    strict' => true],  
]
```

Этот валидатор проверяет, что второе значение является *boolean*.

- `trueValue`: значение, соответствующее *true*. По умолчанию - `'1'`.

- `falseValue`: значение, соответствующее *false*. По умолчанию - `'0'`.
- `strict`: должна ли проверка учитывать соответствие типов данных `trueValue` или `falseValue`. По умолчанию - `false`.

Примечание: Из-за того, что как правило данные, полученные из HTML-форм, представляются в виде строки, обычно вам стоит оставить свойство `strict` равным `false`.

14.3.2 captcha

```
[  
    ['verificationCode', 'captcha'],  
]
```

Этот валидатор обычно используется вместе с `yii\captcha\CaptchaAction` и `yii\captcha\Captcha`, чтобы убедиться, что данные в инпуте соответствуют верификационному коду, отображенному с помощью виджета CAPTCHA.

- `caseSensitive`: необходимо ли учитывать чувствительность к регистру при сравнении. По умолчанию - `false`.
- `captchaAction`: маршрут, соответствующий CAPTCHA action, который рендерит изображение с CAPTCHA. По умолчанию - `'site/captcha'`.
- `skipOnEmpty`: может ли валидация быть пропущена, если *input* пустой. По умолчанию - `false`, что означает, что *input* обязателен.

14.3.3 compare

```
[  
    // проверяет, является ли значение атрибута "password" таким же, как "  
    password_repeat"  
    ['password', 'compare'],  
  
    // проверяет, что возраст больше или равен 30  
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],  
]
```

Этот валидатор сравнивает значение указанного атрибута с другим, чтобы удостовериться, что их отношение соответствует описанному в свойстве `operator`.

- `compareAttribute`: имя атрибута, с которым нужно сравнить значение. Когда валидатор используется для проверки атрибута, значением по умолчанию для этого свойства будет имя этого же атрибута с суффиксом `_repeat`. Например, если проверяющийся атрибут - `password`, то значение свойства по умолчанию будет `password_repeat`.
- `compareValue`: постоянное значение, с которым будут сравниваться входящие данные. Когда одновременно указаны это свойство и `compareAttribute`, это свойство получит приоритет.

- **operator**: оператор сравнения. По умолчанию `==`, что означает проверку на эквивалентность входящих данных и в `compareAttribute`, и в `compareValue`. Поддерживаются следующие операторы:
 - `==`: проверяет два значения на эквивалентность. Сравнение не учитывает тип данных.
 - `===`: проверяет два значения на эквивалентность. Сравнение строгое и учитывает тип данных.
 - `!=`: проверяет, что два значения не эквивалентны. Сравнение не учитывает тип данных.
 - `!==`: проверяет, что два значения не эквивалентны. Сравнение строгое и учитывает тип данных.
 - `>`: проверяет, что валидируемое значение больше, чем то, с которым происходит сравнение.
 - `>=`: проверяет, что валидируемое значение больше или равно тому, с которым происходит сравнение.
 - `<`: проверяет, что валидируемое значение меньше, чем то, с которым происходит сравнение.
 - `<=`: проверяет, что валидируемое значение меньше или равно тому, с которым происходит сравнение.

14.3.4 date

```
[
    [['from', 'to'], 'date'],
]
```

Этот валидатор проверяет соответствие входящих данных форматам *date*, *time* или *datetime*. Опционально, он может конвертировать входящее значение в UNIX timestamp и сохранить в атрибуте, описанном здесь: `timestampAttribute`.

- **format**: формат даты/времени, согласно которому должна быть сделана проверка. Чтобы узнать больше о формате строки, пожалуйста, посмотрите руководство PHP по `date_create_from_format()`⁶. Значением по умолчанию является `'Y-m-d'`.
- **timestampAttribute**: имя атрибута, которому этот валидатор может передать UNIX timestamp, конвертированный из строки даты/времени.

14.3.5 default

```
[
    // установить null для "age" в качестве значения по умолчанию
    ['age', 'default', 'value' => null],
]
```

⁶<http://www.php.net/manual/ru/datetime.createfromformat.php>

```
// установить "USA" в качестве значения по умолчанию для "country"
['country', 'default', 'value' => 'USA'],

// установить в "from" и "to" дату 3 дня и 6 дней от сегодняшней, если
они пустые
[['from', 'to'], 'default', 'value' => function ($model, $attribute) {
    return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6
days')));
}],
]
```

Этот валидатор не проверяет данные. Вместо этого он присваивает значения по умолчанию проходящим проверку атрибутам, если они пусты.

- **value:** значение по умолчанию или функция обратного вызова, которая возвращает значение по умолчанию, которое будет присвоено проверяемому атрибуту, если он пустой. Функция обратного вызова должна выглядеть так:

```
function foo($model, $attribute) {
    // ... вычисление $value ...
    return $value;
}
```

Информация: Как определить, является значение пустым или нет, более подробно описано в отдельной статье в секции [Пустые значения](#).

14.3.6 double

```
[
    // проверяет, является ли "salary" числом типа double
    ['salary', 'double'],
]
```

Этот валидатор проверяет, что входящее значение является корректным *double* числом. Он идентичен валидатору *number*. (Прим. пер.: корректным *float* числом).

- **max:** верхний лимит (включительно) для значений. Если не установлен, значит, валидатор не будет проверять верхний лимит.
- **min:** Нижний лимит (включительно) для значений. Если не установлен, валидатор не будет проверять нижний лимит.

14.3.7 email

```
[
    // проверяет, что "email" - это корректный email-адрес
    ['email', 'email'],
]
```

Валидатор проверяет, что значение входящих данных является корректным email-адресом.

- **allowName**: можно ли передавать в атрибут имя (пример: John Smith <john.smith@example.com>). По умолчанию - **false**.
- **checkDNS**, проверяет, существует ли доменное имя для введенного адреса (и A, и MX запись). Учтите, что проверка может закончиться неудачей, что может быть вызвано временными проблемами с DNS, даже если email-адрес корректен. По умолчанию - **false**.
- **enableIDN**, нужно ли учитывать IDN (многоязычные доменные имена). По умолчанию - **false**. Учтите, что для использования IDN-валидации вам нужно установить и включить PHP расширение `intl`, иначе будет выброшено исключение.

14.3.8 exist

```
[
    // a1 должно существовать в столбце, который представляется атрибутом "
    a1"
    ['a1', 'exist'],

    // a1 должно существовать, но его значение будет использовать a2 для
    проверки существования
    ['a1', 'exist', 'targetAttribute' => 'a2'],

    // и a1, и a2 должны существовать, в противном случае оба атрибута
    будут возвращать ошибку
    [['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

    // и a1, и a2 должны существовать, но только атрибут a1 будет
    возвращать ошибку
    ['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 требует проверки существования a2 и a3 используя( значение a1)
    ['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

    // a1 должен существовать. Если a1 - массив, то каждый его элемент
    должен существовать
    ['a1', 'exist', 'allowArray' => true],
]
```

Этот валидатор ищет входящие данные в столбце таблицы. Он работает только с атрибутами модели [Active Record](#). Он поддерживает проверку и одного столбца, и нескольких.

- **targetClass**: имя класса [Active Record](#), который должен быть использован для проверки входящего значения. Если не установлен, будет использован класс текущей модели.
- **targetAttribute**: имя атрибута в **targetClass** который должен быть использован для проверки существования входящего значения. Если не установлен, будет использовано имя атрибута, который про-

веряется в данный момент. Вы можете использовать массив для валидации нескольких столбцов одновременно. Значения массива являются атрибутами, которые будут использованы для проверки существования, тогда как ключи массива будут являться атрибутами, чьи значения будут проверены. Если ключ и значения одинаковы, вы можете указать только значение.

- **filter**: дополнительный фильтр, который будет добавлен к запросу в базу данных для проверки на существование значения. Это может быть строка или массив, представляющие дополнительные условия в запросе (подробнее о формате значений запроса: `yii\db\Query::where()`), или анонимная функция с сигнатурой `function ($query)`, где `$query` - это `Query` объект, который вы можете модифицировать в функции.
- **allowArray**: разрешать ли значению быть массивом. По умолчанию - `false`. Если свойство установлено в `true` и тип входящих данных - массив, тогда каждый его элемент должен существовать в соответствующем столбце таблицы. Помните, что это свойство не может быть установлено в `true`, если вы валидируете несколько столбцов, передавая их в `targetAttribute` как массив.

14.3.9 file

```
[
    // проверяет, что "primaryImage" - это загруженное изображение в
    // формате PNG, JPG или GIF
    // размер файла должен быть меньше 1MB
    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'], 'maxSize'
    ' => 1024*1024],
]
```

Этот валидатор проверяет, что `input` является корректным загруженным файлом.

- **extensions**: список имен расширений, которые допустимы для загрузки. Это также может быть или массив, или строка, содержащая имена файловых расширений, разделенных пробелом или запятой (пр.: "gif, jpg"). Имя расширения не чувствительно к регистру. По умолчанию - `null`, что значит, что все имена файловых расширений допустимы.
- **mimeType**: список MIME-типов, которые допустимы для загрузки. Это может быть или массив, или строка, содержащая MIME-типы файлов, разделенные пробелом или запятой (пример: "image/jpeg, image/png"). В именах MIME-типов допустимо использовать символ `*` для выбора группы MIME-типов. Например, `image/*` разрешит все типы, которые начинаются с `image/` (пример: `image/jpeg`, `image/png`). Имена MIME-типов не чувствительны к регистру. По умолчанию

- `null`, что значит, что допустимы все MIME-типы. Более детальную информацию можно найти в списке MIME-типов⁷.

- `minSize`: минимальный размер файла в байтах, разрешенный для загрузки. По умолчанию - `null`, что значит, что нет минимального лимита.
- `maxSize`: максимальный размер файла в байтах, разрешенный для загрузки. По умолчанию - `null`, что значит, что нет максимального лимита.
- `maxFiles`: максимальное количество файлов, которое может быть передано в атрибут. По умолчанию 1, что значит, что `input` должен быть файлом в единственном экземпляре. Если больше, чем 1, то атрибут должен быть массивом, состоящим из не более, чем `maxFiles` загруженных файлов.
- `checkExtensionByMimeType`: нужно ли проверять расширение файла исходя из его MIME-типа. Если они не соответствуют друг другу, то файл будет считаться некорректным. По умолчанию - `true`, то есть проверка будет произведена.

`FileValidator` используется вместе с `yii\web\UploadedFile`. Пожалуйста, посмотрите раздел [Загрузка файлов](#) для более полного понимания загрузки и проверки файлов.

14.3.10 filter

```
[
    // обрезает пробелы вокруг "username" и "email"
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' =>
    true],

    // нормализует значение "phone"
    ['phone', 'filter', 'filter' => function ($value) {
        // нормализация значения происходит тут
        return $value;
    }],
]
```

Этот валидатор не проверяет данные. Вместо этого он применяет указанный фильтр к входящему значению и присваивает результат применения фильтра атрибуту.

- `filter`: PHP-callback, осуществляющий фильтрацию. Это может быть глобальная php функция, анонимная функция и т.д. Функция должна выглядеть как `function ($value) { return $newValue; }`. Это свойство обязательно должно быть установлено.
- `skipOnArray`: нужно ли пропускать валидацию, если входящим значением является массив. По умолчанию - `false`. Помните, что если

⁷https://ru.wikipedia.org/wiki/\T2A\CYRS\T2A\cyrp\T2A\cyri\T2A\cyrs\T2A\cyro\T2A\cyrk_MIME-\T2A\cyrt\T2A\cyri\T2A\cyrp\T2A\cyro\T2A\cyrv

фильтр не может принимать массив, вы должны установить это значение в `true`. Иначе могут произойти различные ошибки PHP.

Трюк: Если вы хотите удалить пробелы вокруг значений атрибута, вы можете использовать валидатор `trim`.

14.3.11 image

```
[
    // проверяет, что "primaryImage" - это валидное изображение с
    // указанными размерами
    ['primaryImage', 'image', 'extensions' => 'png, jpg',
     'minWidth' => 100, 'maxWidth' => 1000,
     'minHeight' => 100, 'maxHeight' => 1000,
    ],
]
```

Этот валидатор проверяет, что входящие данные являются корректным файлом изображения. Он расширяет `file` валидатор и наследует все его свойства. Кроме того, он поддерживает следующие дополнительные свойства, специфичные для валидации изображений:

- `minWidth`: минимальная ширина изображения. По умолчанию `null`, что значит, что нет нижнего лимита.
- `maxWidth`: максимальная ширина изображения. По умолчанию `null`, что значит, что нет верхнего лимита.
- `minHeight`: минимальная высота изображения. По умолчанию `null`, что значит, что нет нижнего лимита.
- `maxHeight`: максимальная высота изображения. По умолчанию `null`, что значит, что нет верхнего лимита.

14.3.12 ip

```
[
    // проверяет, что "ip_address" - это валидный IPv4 или IPv6 адрес
    ['ip_address', 'ip'],

    // проверяет, что "ip_address" - это валидный IPv6 адрес или подсеть,
    // значение будет развернуто в формат полной записи IPv6 адреса
    ['ip_address', 'ip', 'ipv4' => false, 'subnet' => null, 'expandIPv6' =>
     true],

    // проверяет, что "ip_address" - это валидный IPv4 или IPv6 адрес,
    // разрешает использования символа отрицания '!'
    ['ip_address', 'ip', 'negation' => true],
]
```

Этот валидатор проверяет, является ли входящее значение валидным IPv4/IPv6 адресом или подсетью. Он также может изменять значение

атрибута, если включена нормализация или развертывание IPv6 адресов.

Валидатор имеет такие свойства:

- **ipv4**: может ли проверяемое значение быть IPv4 адрессом. По умолчанию **true**.
- **ipv6**: может ли проверяемое значение быть IPv6 адрессом. По умолчанию **true**.
- **subnet**: может ли проверяемое значение быть IP адрессом с CIDR (подсетью), например 192.168.10.0/24
 - **true** - указание подсети обязательно;
 - **false** - указание подсети запрещено;
 - **null** - указание подсети возможно, но не обязательно.

По умолчанию **false**.

- **normalize**: нормализовать ли проверяемый IP адрес без CIDR к IP адресу с наименьшим CIDR (32 для IPv4 или 128 для IPv6). Свойство действует только если **subnet** не установлен в **false**. Например:
 - 10.0.1.5 будет приведен к 10.0.1.5/32
 - 2008:db0::1 будет приведен к 2008:db0::1/128
- **negation**: может ли проверяемое значение иметь символ отрицания ! в начале, например !192.168.0.1. По умолчанию **false**.
- **expandIPv6**: разворачивать ли IPv6 адрес в формат полной записи. Например, IPv6 адрес 2008:db0::1 будет развернут в 2008:0db0:0000:0000:0000:0000:0000:0001. По умолчанию **false**.
- **ranges**: массив IPv4 или IPv6 диапазонов, которые разрешены или запрещены.

Если свойство не установлено, все IP адреса разрешены. Иначе, правила будут проверяться последовательно до первого вхождения. IP адрес будет запрещен, если не подпадет ни под одно правило. Например: 'php [

```
'client_ip', 'ip', 'ranges' => [
    '192.168.10.128'
    '!192.168.10.0/24',
    'any' // разрешает все остальные IP адреса
]
```

] ' В этом примере, доступ разрешен для всех IPv4 и IPv6 адресов кроме подсети 192.168.10.0/24. IPv4 адрес 192.168.10.128 также разрешен, так как находится перед запрещающим правилом.

- **networks**: массив псевдонимов, которые могут быть использованы в **ranges**. Формат массива:
 - ключ - имя псевдонима
 - значение - массив строк. Строка может быть как диапазоном адресов, так и другим псевдонимом. Строка может иметь символ отрицания ! в начале (не зависит от свойства **negation**).

Следующие псевдонимы определены по умолчанию:

- *: any
- any: 0.0.0.0/0, ::/0
- private: 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, fd00::/8
- multicast: 224.0.0.0/4, ff00::/8
- linklocal: 169.254.0.0/16, fe80::/10
- localhost: 127.0.0.0/8, ::1
- documentation: 192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, 2001:db8::/32
- system: multicast, linklocal, localhost, documentation

Информация: Этот валидатор стал доступным начиная с версии 2.0.7.

14.3.13 in

```
[
  // проверяет, что значение "level" равно 1, 2 или 3
  ['level', 'in', 'range' => [1, 2, 3]],
]
```

Этот валидатор проверяет, что входящее значение соответствует одному из значений, указанных в `range`.

- `range`: список значений, с которыми будет сравниваться входящее значение.
- `strict`: должно ли сравнение входящего значения со списком значений быть строгим (учитывать тип данных). По умолчанию `false`.
- `not`: должен ли результат проверки быть инвертирован. По умолчанию - `false`. Если свойство установлено в `true`, валидатор проверяет, что входящее значение НЕ соответствует ни одному из значений, указанных в `range`.
- `allowArray`: устанавливает, допустимо ли использовать массив в качестве входных данных. Если установлено в `true` и входящие данные являются массивом, для каждого элемента входящего массива должно быть найдено соответствие в `range`.

14.3.14 integer

```
[
  // проверяет "age" на то, что это integer значение
  ['age', 'integer'],
]
```

Проверяет, что входящее значение является `integer` значением.

- `max`: верхний лимит (включительно) для числа. Если не установлено, валидатор не будет проверять верхний лимит.

- **min**: нижний лимит (включительно) для числа. Если не установлено, валидатор не будет проверять нижний лимит.

14.3.15 match

```
[
    // проверяет, что "username" начинается с буквы и содержит только
    // буквенные символы,
    // числовые символы и знак подчеркивания
    ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']
]
```

Этот валидатор проверяет, что входящее значение совпадает с указанным регулярным выражением.

- **pattern**: регулярное выражение, с которым должно совпадать входящее значение. Это свойство должно быть установлено, иначе будет выброшено исключение.
- **not**: инвертирует регулярное выражение. По умолчанию **false**, что значит, что валидация будет успешна, только если входящее значение совпадает с шаблоном. Если установлено в **true**, валидация будет успешна, только если входящее значение НЕ совпадает с шаблоном.

14.3.16 number

```
[
    // проверяет, является ли "salary" числом
    ['salary', 'number'],
]
```

Этот валидатор проверяет, являются ли входящие значения числовыми. Он эквивалентен валидатору `double`.

- **max**: верхний лимит (включительно) для числа. Если не установлено, валидатор не будет проверять верхний лимит.
- **min**: нижний лимит (включительно) для числа. Если не установлено, валидатор не будет проверять нижний лимит.

14.3.17 required

```
[
    // проверяет, являются ли "username" и "password" не пустыми
    ['username', 'password'], 'required',
]
```

Этот валидатор проверяет, являются ли входящие значения не пустыми.

- **requiredValue**: желаемое значение, которому должны соответствовать проверяемые данные. Если не установлено, это значит, что данные должны быть не пустыми.

- **strict**: учитывать или нет соответствие типу данных при валидации (строгое сравнение). Если `requiredValue` не установлено, а это свойство установлено в `true`, валидатор проверит, что входящее значение строго не соответствует `null`; если свойство установлено в `false`, валидатор будет проверять значение на пустоту с приведением типов. Если `requiredValue` установлено, сравнение между входящими данными и `requiredValue` будет также учитывать тип данных, если это свойство установлено в `true`.

Информация: как определить, является ли значение пустым или нет, подробнее рассказывается в секции [Пустые значения](#).

14.3.18 safe

```
[
  // обозначает "description" как safe атрибут
  ['description', 'safe'],
]
```

Этот валидатор не проверяет данные. Вместо этого он указывает, что атрибут является **безопасным атрибутом**.

14.3.19 string

```
[
  // проверяет, что "username" это строка с длиной от 4 до 24 символов
  ['username', 'string', 'length' => [4, 24]],
]
```

Этот валидатор проверяет, что входящее значение - это корректная строка с указанной длиной.

- **length**: описывает длину для строки, проходящей валидацию. Может быть определен следующими способами:
 - числом: точная длина, которой должна соответствовать строка;
 - массив с одним элементом: минимальная длина входящей строки (напр.: `[8]`). Это перезапишет `min`.
 - массив с двумя элементами: минимальная и максимальная длина входящей строки (напр.: `[8, 128]`). Это перезапишет и `min`, и `max`.
- **min**: минимальная длина входящей строки. Если не установлено, то не будет ограничения на минимальную длину.
- **max**: максимальная длина входящей строки. Если не установлено, то не будет ограничения на максимальную длину.
- **encoding**: кодировка входящей строки. Если не установлено, будет использовано значение из `charset`, которое по умолчанию установлено в UTF-8.

14.3.20 trim

```
[
    // обрезает пробелы вокруг "username" и "email"
    [['username', 'email'], 'trim'],
]
```

Этот валидатор не производит проверки данных. Вместо этого он будет обрезать пробелы вокруг входящих данных. Помните, что если входящие данные являются массивом, то они будут проигнорированы этим валидатором.

14.3.21 unique

```
[
    // a1 должен быть уникальным в столбце, который представляет "a1"
    // атрибут
    ['a1', 'unique'],

    // a1 должен быть уникальным, но для проверки на уникальность
    // будет использован столбец a2
    ['a1', 'unique', 'targetAttribute' => 'a2'],

    // a1 и a2 вместе должны быть уникальны, и каждый из них
    // будет получать сообщения об ошибке
    [['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 и a2 вместе должны быть уникальны, но только a1 будет получать
    // сообщение об ошибке
    ['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 должен быть уникальным, что устанавливается проверкой
    // уникальности a2 и a3
    // используя( значение a1)
    ['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],
]
```

Этот валидатор проверяет входящие данные на уникальность в столбце таблицы. Он работает только с атрибутами модели [Active Record](#). Он поддерживает проверку либо одного столбца, либо нескольких.

- **targetClass:** имя класса [Active Record](#), который должен быть использован для проверки значения во входящих данных. Если не установлен, будет использован класс модели, которая в данный момент проходит проверку.
- **targetAttribute:** имя атрибута в **targetClass**, который должен быть использован для проверки на уникальность входящего значения. Если не установлено, будет использован атрибут, проверяемый в данный момент. Вы можете использовать массив для проверки нескольких столбцов таблицы на уникальность. Значения массива - это атрибуты, которые будут использованы для валидации, а ключи массива - это атрибуты, которые предоставляют данные для

валидации. Если ключ и значение одинаковые, вы можете указать только значение.

- **filter**: дополнительный фильтр, который можно присоединить к запросу в БД, чтобы использовать его при проверке значения на уникальность. Это может быть строка или массив, представляющие дополнительные условия для запроса (см. `yii\db\Query::where()` о формате условий в запросе), или анонимная функция вида `function ($query)`, где `$query` это объект `Query`, который вы можете изменить в функции.

14.3.22 url

```
[
    // Проверяет, что "website" является корректным URL. Добавляет http://
    к атрибуту "website".
    // если у него нет URI схемы
    ['website', 'url', 'defaultScheme' => 'http'],
]
```

Этот валидатор проверяет, что входящее значение является корректным URL.

- **validSchemes**: массив с указанием на URI-схему, которая должна считаться корректной. По умолчанию `['http', 'https']`, что означает, что и `http`, и `https` URI будут считаться корректными.
- **defaultScheme**: схема URI, которая будет присоединена к входящим данным, если в них отсутствует URI-схема. По умолчанию `null`, что значит, что входящие данные не будут изменены.
- **enableIDN**: должна ли валидация учитывать IDN (интернационализованные доменные имена). По умолчанию - `false`. Учтите, что для того, чтобы IDN валидация работала корректно, вы должны установить `intl` PHP расширение, иначе будет выброшено исключение.

14.4 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][[]], [форматированием чисел и дат][[]].

14.4.1 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 трёхбуквенный код языка в нижнем регистре в соответствии со стандартом ISO-639⁸, а `cc` - это код страны в соответствии со стандартом ISO-3166⁹.

Примечание: больше информации о синтаксисе и концепции локалей можно получить в документации проекта ICU¹⁰.

14.4.2 Перевод сообщений

Перевод используется для локализации сообщений, которые будут выведены в приложении в соответствии с языком, который выбрал пользователь.

⁸<http://www.loc.gov/standards/iso639-2/>

⁹<http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html>

¹⁰<http://userguide.icu-project.org/locale#T0C-The-Locale-Concept>

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 PHP-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`.

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 PHP-файлах (используя `PhpMessageSource`), Yii предоставляет ещё два класса:

- `yii\i18n\GettextMessageSource`, использующий GNU Gettext для MO или PO 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 intl¹¹. 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 ICU¹², но далее в документации будут приведены примеры использования расширенных возможностей интернационализации.

```
$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);
```

¹¹<http://www.php.net/manual/en/intro.intl.php>

¹²<http://icu-project.org/apiref/icu4c/classMessageFormat.html>

Или определить свой формат:

```
$sum = 42;  
echo \Yii::t('app', 'Balance: {0, number, ,000,000000}', $sum);
```

Описание форматирования¹³.

```
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 форматирования¹⁴.

```
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 форматирования¹⁵.

```
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!”.

```
echo \Yii::t('app', 'Вы находитесь здесь уже {n, duration}', ['n' => 47]);
```

Выведет сообщение “Вы находитесь здесь уже 47 сек.”.

¹³http://icu-project.org/apiref/icu4c/classicu_1_1DecimalFormat.html

¹⁴http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html

¹⁵http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html

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} oneлежит{ #
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.org¹⁶.

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',
```

¹⁶http://unicode.org/repos/cldr-tmp/trunk/diff/supplemental/language_plural_rules.html

```
'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 категории `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 не переведённые 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 handleMissingTranslation(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.

14.4.3 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.

14.4.4 Форматирование чисел и дат

См. описание [форматирования дат](#).

14.4.5 Настройка РНР-окружения

Для работы с большей частью функций интернационализации, Yii использует РНР-расширение `intl`¹⁷. Например, это расширение используют классы, отвечающие за форматирование чисел и дат `yii\i18n\Formatter` 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.

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

PHP-расширение intl¹⁸ основано на библиотеке ICU¹⁹, 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, ICU 51 и так далее.

14.4.6 Известные проблемы

- В ICU версии 52.1 было испорчено форматирование множественных чисел (plural) в русском языке. Проблема решается обновлением ICU до версии 53.1 или старше.

14.5 Отправка почты

Примечание: Этот раздел находится в стадии разработки.

Yii позволяет оформлять и посылать E-mail сообщения. Однако, ядро фреймворка предоставляет только функциональность оформления и основной интерфейс. Фактический механизм отправки почты должен быть предоставлен с помощью расширения, потому что различным проектам могут потребоваться различные реализации и обычно они зависят от внешних сервисов и библиотек.

Для наиболее распространенных ситуаций вы можете использовать официальное расширение yii2-swiftmailer²⁰.

¹⁸<http://php.net/manual/en/book.intl.php>

¹⁹<http://site.icu-project.org/>

²⁰<https://github.com/yiisoft/yii2-swiftmailer>

14.5.1 Настройка

Настройка почтового компонента зависит от расширения, которое вы выбрали. В целом настройка вашего приложения должна выглядеть так:

```
return [  
    //....  
    'components' => [  
        'mailer' => [  
            'class' => 'yii\swiftmailer\Mailer',  
        ],  
    ],  
];
```

14.5.2 Основы использования

Когда 'mailer' компонент настроен, вы можете использовать следующий код, чтобы отправить почтовое сообщение:

```
Yii::$app->mailer->compose()  
->setFrom('from@domain.com')  
->setTo('to@domain.com')  
->setSubject('Тема сообщения')  
->setTextBody('Текст сообщения')  
->setHtmlBody('<b>текст</b> сообщения в формате HTML</b>')  
->send();
```

В показанном выше примере метод `compose()` создает экземпляр почтового сообщения, который затем заполняется и отправляется. Вы можете использовать более сложную логику в этом процессе, если вам понадобится:

```
$message = Yii::$app->mailer->compose();  
if (Yii::$app->user->isGuest) {  
    $message->setFrom('from@domain.com')  
} else {  
    $message->setFrom(Yii::$app->user->identity->email)  
}  
$message->setTo(Yii::$app->params['adminEmail'])  
->setSubject('Тема сообщения')  
->setTextBody('Текст сообщения')  
->send();
```

Примечание: каждое 'mailer' расширение имеет два главных класса: 'Mailer' и 'Message'. 'Mailer' всегда знает имя класса и специфику 'Message'. Не пытайтесь создать экземпляр объекта 'Message' напрямую - всегда используйте для этого метод `compose()`.

Вы также можете послать несколько сообщений за раз:

```

$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);

```

В некоторых почтовых расширениях этот подход может быть полезен, так как использует одиночное сетевое сообщение.

14.5.3 Компоновка почтовых сообщений

Yii предоставляет возможность оформления содержания почтовых сообщений через специальные файлы вида. По умолчанию эти файлы должны быть расположены в директории '@app/mail'.

Пример содержания почтового файла вида:

```

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

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\BaseMessage instance of newly created mail
    message */

?>
<h2>This message allows you to visit our site home page by one click</h2>
<?= Html::a('Go to home page', Url::home('http')) ?>

```

Для того, чтобы оформить содержание сообщения через файл вида, просто передайте название файла вида в `compose()` метод:

```

Yii::$app->mailer->compose('home-link') // здесь устанавливается результат
    рендеринга вида в тело сообщения
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->send();

```

Вы можете передать дополнительный параметр, относящийся к виду в `compose()` метод, который будет доступен внутри файла вида:

```

Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);

```

Вы можете указать разные файлы видов для HTML и простого текста в содержании сообщения:

```

Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);

```

```
1);
```

Если вы укажете название вида как строку, результат рендеринга в теле сообщения будет использоваться как HTML, в то время как при обычном тексте в теле сообщения при компоновке будут удаляться все HTML теги.

Результат рендеринга вида может быть вставлен в макет (layout), который может быть установлен, используя `yii\mail\BaseMailer::$htmlLayout` и `yii\mail\BaseMailer::$textLayout`. Это будет работать аналогично макетам в обычном веб приложении. Макет может использовать CSS стили или другие общие элементы страниц для использования в сообщении:

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

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\MessageInterface the message being composed */
/* @var $content string main view render result */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/
    TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=?= Yii::
        $app->charset ??" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">With kind regards, <?= Yii::$app->name ?> team</div>
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

14.5.4 Прикрепление файлов

Вы можете прикрепить вложения к сообщению с помощью методов `attach()` и `attachContent()`:

```
$message = Yii::$app->mailer->compose();

// Прикрепление файла из локальной файловой системы:
$message->attach('/path/to/source/file.pdf');
```

```
// Прикрепить файл на лету
$message->attachContent('Attachment content', ['fileName' => 'attach.txt', '
    contentType' => 'text/plain']);
```

14.5.5 Вложение изображений

Вы можете вставить изображения в содержание сообщения через `embed()` метод. Этот метод возвращает `id` прикрепленной картинки, которые должны быть доступны в `'img'` тегах. Этот метод легко использовать, когда сообщение составляется через файлы представления:

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' => '/path/to/
    image.jpg'])
// ...
->send();
```

Внутри файла представления вы можете использовать следующий код:

```

```

14.5.6 Тестирование и отладка

Разработчикам часто надо проверять, что почтовые сообщения отправляются из приложения, их содержание и так далее. Такая возможность предоставляется в Yii через `yii\mail\BaseMailer::useFileTransport`. Если это опция включена, то она принудительно сохраняет данные почтовых сообщений в локальный файл вместо его отправки. Эти файлы будут сохранены в директории `yii\mail\BaseMailer::fileTransportPath`, которая по умолчанию `'@runtime/mail'`.

Примечание: вы можете либо сохранить сообщения в файл, либо послать его фактическим получателям, но не используйте оба варианта одновременно.

Файл почтового сообщения может быть открыт обычным текстовым редактором, также вы можете просматривать фактические заголовки сообщений, их содержание и так далее. Этот механизм может понадобиться во время отладки приложения, либо прогонки `unit` тестов.

Примечание: содержание файла почтового сообщения формируется через `\yii\mail\MessageInterface::toString()`, правда это зависит от почтового расширения, которое вы используете в своем приложении.

14.5.7 Создание вашего собственного решения

Для того, чтобы создать свое собственное решение, вам надо будет создать два класса: одно для 'Mailer' и другое для 'Message'. Вы можете использовать `yii\mail\BaseMailer` и `yii\mail\BaseMessage` как базовые классы для вашего решения. Эти классы уже содержат базовую логику, которая описана в этом руководстве. Однако, их использование не обязательно, достаточно унаследоваться от `yii\mail\MailerInterface` и `yii\mail\MessageInterface` интерфейсов. Затем вам необходимо обеспечить выполнение всех абстрактных методов этих интерфейсов для построения вашего решения.

14.6 Оптимизация производительности

Существует много факторов, влияющих на производительность веб-приложения. Какие-то относятся к окружению, какие-то к вашему коду, а какие-то к самому Yii. В этом разделе мы перечислим большинство из них и объясним, как можно улучшить производительность приложения, регулируя эти факторы.

14.6.1 Оптимизация окружения PHP

Хорошо сконфигурированное окружение PHP очень важно. Для получения максимальной производительности,

- Используйте последнюю стабильную версию PHP. Мажорные релизы PHP могут принести значительные улучшения производительности.
- Включите кеширование байткода в OpCache²¹ (PHP 5.5 и старше) или APC²² (PHP 5.4 и более ранние версии). Кеширование байткода позволяет избежать затрат времени на обработку и подключение PHP скриптов при каждом входящем запросе.

14.6.2 Отключение режима отладки

При запуске приложения в производственном режиме, вам нужно отключить режим отладки. Yii использует значение константы `YII_DEBUG` чтобы указать, следует ли включить режим отладки. Когда режим отладки включен, Yii тратит дополнительное время чтобы создать и записать отладочную информацию.

Вы можете разместить следующую строку кода в начале входного скрипта чтобы отключить режим отладки:

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

²¹<http://php.net/opcache>

²²<http://ru2.php.net/apc>

Информация: Значение по умолчанию для константы `YII_DEBUG` — `false`. Так что, если вы уверены, что не изменяете значение по умолчанию где-то в коде приложения, можете просто удалить эту строку, чтобы отключить режим отладки.

14.6.3 Использование техник кеширования

Вы можете использовать различные техники кеширования чтобы значительно улучшить производительность вашего приложения. Например, если ваше приложение позволяет пользователям вводить текст в формате Markdown, вы можете закешировать разобранное содержимое Markdown, чтобы избежать разбора одной и той же разметки Markdown неоднократно при каждом запросе. Пожалуйста, обратитесь к разделу [Кеширование](#) чтобы узнать о поддержке кеширования, которую предоставляет Yii.

14.6.4 Включение кеширования схемы

Кеширование схемы - это специальный *тип кеширования*, который должен быть включен при использовании [Active Record](#). Как вы знаете, Active Record достаточно умен, чтобы обнаружить информацию о схеме (например, имена столбцов, типы столбцов, ограничения) таблицы БД без необходимости описывать ее вручную. Active Record получает эту информацию, выполняя дополнительные SQL запросы. При включении кеширования схемы, полученная информация о схеме будет сохранена в кэше и повторно использована при последующих запросах.

Чтобы включить кеширование схемы, сконфигурируйте [компонент приложения](#) `cache` для хранения информации о схеме и установите `yii\db\Connection::$enableSchemaCache` в `true` в конфигурации приложения:

```
return [
    // ...
    'components' => [
        // ...
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase',
            'username' => 'root',
            'password' => '',
            'enableSchemaCache' => true,

            // Продолжительность кеширования схемы.
            'schemaCacheDuration' => 3600,

            // Название компонента кеша, используемого для хранения
            // информации о схеме
            'schemaCache' => 'cache',
        ],
    ],
];
```

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

14.6.5 Объединение и минимизация ресурсов

Сложные веб-страницы часто подключают много CSS и/или JavaScript файлов. Для уменьшения числа HTTP запросов и общего размера загрузки этих ресурсов, вы должны рассмотреть вопрос об их объединении в один файл и его сжатии. Это может сильно увеличить скорость загрузки страницы и снизить нагрузку на сервер. Для получения более подробной информации обратитесь, пожалуйста, к разделу [Ресурсы](#)

14.6.6 Оптимизация хранилища сессий

По умолчанию данные сессий хранятся в файлах. Это удобно для разработки или в маленьких проектах. Но когда дело доходит до обработки множества параллельных запросов, то лучше использовать более сложные хранилища, такие как базы данных. Yii поддерживает различные хранилища “из коробки”. Вы можете использовать эти хранилища, сконфигурировав компонент `session` в [конфигурации приложения](#) как показано ниже,

```
return [
    // ...
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',

            // Установите следующее, если вы хотите использовать компонент
            // БД, с названием
            // отличным от значения по умолчанию 'db'.
            'db' => 'mydb',

            // Чтобы перезаписать таблицу сессий, заданную по умолчанию,
            // установите
            'sessionTable' => 'my_session',
        ],
    ],
];
```

Приведенная выше конфигурация использует таблицу базы данных для хранения сессионных данных. По умолчанию, используется компонент приложения `db` для подключения к базе данных и сохранения сессионных данных в таблице `session`. Вам нужно будет создать таблицу `session` заранее:

```
CREATE TABLE session (
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
```

```
data BLOB  
)
```

Вы также можете хранить сессионные данные в кеше с помощью `yii\web\CacheSession`. Теоретически, вы можете использовать любое поддерживаемое [хранилище кеша](#). Тем не менее, помните, что некоторые хранилища кеша могут *сбрасывать* закешированные данные при достижении лимитов хранилища. По этой причине, вы должны в основном использовать хранилища кеша, которые не имеют таких лимитов.

Если на вашем сервере установлен Redis²³, настоятельно рекомендуется выбрать его в качестве хранилища сессий используя `yii\redis\Session`.

14.6.7 Оптимизация базы данных

Выполнение запросов к БД и выборки данных часто являются узким местом производительности веб-приложения. Хотя использование техник [кэширования данных](#) может *смягчить* снижение производительности, оно не решает проблему полностью. Когда база данных содержит огромное количество данных, и данные в кэше невалидны, получение свежих данных без правильного проектирования БД и запросов может быть чрезмерно ресурсоемкой операцией.

Общей методикой для повышения производительности запросов к БД является создание индексов для тех столбцов таблицы, по которым делается выборка. Например, если вам нужно найти запись о пользователе по `username`, вам надо создать индекс на `username`. Обратите внимание, что в то время как индексирование может сделать `SELECT` запросы намного быстрее, оно будет замедлять `INSERT`, `UPDATE` и `DELETE` запросы.

Для сложных запросов к БД рекомендуется создавать представления базы данных (`views`), чтобы сэкономить время подготовки и разбора запросов.

Последнее, хотя и не менее важное: используйте `LIMIT` в ваших `SELECT` запросах. Это позволяет избежать извлечения большого количества данных из базы данных и исчерпания памяти, выделенной для PHP.

14.6.8 Использование обычных массивов

Хотя [Active Record](#) очень удобно использовать, это не так эффективно, как использование простых массивов, когда вам нужно получить большое количество данных из БД. В этом случае, вы можете вызвать `asArray()` при использовании `Active Record` для получения данных, чтобы извлеченные данные были представлены в виде массивов вместо громоздких записей `Active Record`. Например,

²³<http://redis.io/>

```
class PostController extends Controller
{
    public function actionIndex()
    {
        $posts = Post::find()->limit(100)->asArray()->all();

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

В приведенном выше коде, `$posts` будет заполнен массивом строк из таблицы. Каждая строка - это обычный массив. Чтобы получить доступ к столбцу `title` в `i`-й строке, вы можете использовать выражение `$posts[$i]['title']`.

Вы также можете использовать [DAO](#) для создания запросов и извлечения данных в виде обычных массивов.

14.6.9 Оптимизация автозагрузчика Composer

Поскольку автозагрузчик Composer'a используется для подключения большого количества файлов сторонних классов, вы должны оптимизировать его, выполнив следующую команду:

```
composer dumpautoload -o
```

14.6.10 Асинхронная обработка данных

Когда запрос включает в себя некоторые ресурсоемкие операции, вы должны подумать о том, чтобы выполнить эти операции асинхронно, не заставляя пользователя ожидать их окончания.

Существует два метода асинхронной обработки данных: `pull` и `push`.

В методе `pull`, всякий раз, когда запрос включает в себя некоторые сложные операции, вы создаете задачу и сохраняете ее в постоянном хранилище, таком как база данных. Затем в отдельном процессе (таким как задание `cron`) получаете эту задачу и обрабатываете ее.

Этот метод легко реализовать, но у него есть некоторые недостатки. Например, задачи надо периодически забирать из места их хранения. Если делать это слишком редко, задачи будут обрабатываться с большой задержкой, а если слишком часто - это будет создавать большие накладные расходы.

В методе `push`, вы можете использовать очереди сообщений (например, `RabbitMQ`, `ActiveMQ`, `Amazon SQS`, и т.д.) для управления задачами. Всякий раз, когда новая задача попадает в очередь, это инициирует обработку этой задачи обработчиком.

14.6.11 Профилирование производительности

Вы должны профилировать код, чтобы определить узкие места в производительности и принять соответствующие меры. Следующие инструменты для профилирования могут оказаться полезными:

- Отладочный тулбар Yii и отладчик²⁴
- Профайлер XDebug²⁵
- XHProf²⁶

14.7 Окружение виртуального хостинга

Зачастую окружение виртуальных хостингов весьма ограничено как в настройках конфигурации, так и в настройках структуры директорий. В большинстве случаев, однако, возможно запустить Yii 2 на виртуальном хостинге, внося некоторые корректировки.

14.7.1 Установка приложения Basic.

Поскольку на виртуальном хостинге обычно только один webroot, то лучше использовать шаблонное приложение Basic. Прочитайте раздел [Установка Yii](#) и локально установите приложение. После того как оно начнет работать, можно внести необходимые корректировки, которые помогут разместить Basic на виртуальном хостинге.

Переименование webroot

Подключитесь к вашему виртуальному хостингу, используя FTP или другой способ. Скорее всего вы увидите следующее:

```
config
logs
www
```

В приведенном выше описании `www` - это webroot директория веб-сервера. Она может называться по-другому. Возможные названия: `www`, `htdocs` или `public_html`.

В Basic webroot называется `web`. Перед загрузкой своего приложения на виртуальный хостинг, переименуйте локальный webroot на название webroot виртуального хостинга. Например, `web` в `www` или `public_html`, в зависимости от наименования webroot вашего хостинга.

²⁴<https://github.com/yiisoft/yii2-debug/blob/master/docs/guide/README.md>

²⁵<http://xdebug.org/docs/profiler>

²⁶<http://www.php.net/manual/en/book.xhprof.php>

Корневая директория FTP доступна для записи

Если вы можете записать в корневую директорию, где располагаются `config`, `logs` и `www`, то загрузите сюда же `assets`, `commands` и остальные директории, так же, как и у вас, локально.

Добавим настройки для веб-сервера

В случае, если ваш сервер Apache, добавьте в директорию `web` или аналогичную, где располагается `index.php`, файл `.htaccess` со следующим содержанием:

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

В случае использования nginx не требуется каких-либо дополнительных настроек.

Проверка требований

Для того чтобы запустить Yii, ваш веб-сервер должен соответствовать его требованиям. Минимальное требование к PHP - это его версия 5.4. Для того чтобы проверить требования, скопируйте `requirements.php` из корневого каталога в каталог `webroot` и запустите его с помощью браузера, используя url `http://example.com/requirements.php`. Не забудьте после проверки требований удалить файл `requirements.php`.

14.7.2 Установка шаблона приложения Advanced

Установка шаблона Advanced немного сложнее, чем установка Basic, из-за того, что в Advanced имеются две директории `webroot`, работа с которыми на виртуальном хостинге не поддерживается. По этой причине нам потребуется внести изменения в структуру директорий.

Перемещение входных скриптов в одну директорию webroot

Для начала нам необходима директория `webroot`. Создайте новую директорию и назовите её так же, как на виртуальном хостинге, например, `www` или `public_html`, как описывалось выше в разделе Переименование `webroot`. Затем создайте следующую структуру в `www`:

```
www
  admin
backend
common
console
environments
frontend
...
```

Нашей фронтенд директорией будет `www`. Переместите в неё всё из `frontend/web`. Так же поступите и для `backend/web`, скопировав всё в `www/admin`. В каждом случае нужно настроить пути внутри файлов `index.php` и `index-test.php`.

Отдельные сессии и куки

Изначально подразумевалось, что приложения бекенд и фронтенд располагаются на разных доменах. Теперь, когда мы перенесли всё на один домен, куки и сессии из бекенда и фронтенда стали пересекаться. Для решения этой проблемы требуется внести следующие настройки в конфигурацию бекенд-приложения `backend/config/main.php`:

```
'components' => [
    'request' => [
        'csrfParam' => '_backendCSRF',
        'csrfCookie' => [
            'httpOnly' => true,
            'path' => '/admin',
        ],
    ],
    'user' => [
        'identityCookie' => [
            'name' => '_backendIdentity',
            'path' => '/admin',
            'httpOnly' => true,
        ],
    ],
    'session' => [
        'name' => 'BACKENDSESSID',
        'cookieParams' => [
            'path' => '/admin',
        ],
    ],
],
```


14.8 Использование шаблонизаторов

По умолчанию, Yii использует РНР в шаблонах, но вы можете настроить Yii на поддержку других шаблонизаторов, таких как Twig²⁷ или Smarty²⁸, которые доступны в расширениях.

view компонент, ответственный за генерацию видов. Вы можете добавить шаблонизатор, с помощью перенастройки поведения компонента:

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // Array of twig options:
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
                ],
                // ...
            ],
        ],
    ],
]
```

В коде, показанном выше, оба шаблонизатора Smarty и Twig настроены, чтобы использоваться в файле вида. Но чтобы добавить эти расширения в ваш проект, вам необходимо также изменить ваш `composer.json` файл. Добавить в него:

```
"yiisoft/yii2-smarty": "~2.0.0",
"yiisoft/yii2-twig": "~2.0.0",
```

Это код вставляется в секцию `require` файла `composer.json`. После изменения и сохранения этого файла, вы можете установить расширение, запустив `composer update --prefer-dist` в командной строке.

Для получения подробной информации об использовании конкретного шаблонизатора обратитесь в их документации:

- Twig guide²⁹
- Smarty guide³⁰

²⁷<http://twig.sensiolabs.org/>

²⁸<http://www.smarty.net/>

²⁹<https://github.com/yiisoft/yii2-twig/tree/master/docs/guide>

³⁰<https://github.com/yiisoft/yii2-smarty/tree/master/docs/guide>

14.9 Работа со сторонним кодом

Иногда необходимо использовать сторонний код в приложениях Yii. Или же есть потребность использовать Yii в качестве библиотеки в сторонних системах. В этом разделе мы рассмотрим, как это происходит.

14.9.1 Использование сторонних библиотек в Yii

Перед тем, как использовать стороннюю библиотеку в приложении Yii, в первую очередь следует убедиться, что в ней либо явно настроена загрузка классов, либо классы могут загружаться автоматически.

Использование пакетов Composer

Многие сторонние библиотеки поставляются в виде пакетов Composer³¹. Для установки таких библиотек достаточно сделать два простых шага:

1. Изменить файл `composer.json` своего приложения и указать, какие пакеты Composer нужно устанавливать.
2. Выполнить команду `composer install`, чтобы установить указанные пакеты.

Классы установленных пакетов Composer поддерживают автозагрузку с помощью автозагрузчика Composer. Убедитесь, что во **входном скрипте** приложения присутствуют следующие строки, подключающие автозагрузчик Composer:

```
// подключение автозагрузчика Composer
require(__DIR__ . '/../vendor/autoload.php');

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

Использование отдельных библиотек

Если библиотека не поставляется в виде пакета Composer, необходимо установить ее согласно ее руководству по установке. В большинстве случаев потребуется вручную скачать файл с релизом и распаковать его в директорию `BasePath/vendor`, где `BasePath` соответствует **базовому пути** приложения.

Если библиотека использует собственный автозагрузчик классов, его можно подключить во **входном скрипте** приложения. Желательно подключить его до того, как подключается файл `Yii.php`, чтобы при автоматической загрузке классов у автозагрузчика классов Yii был приоритет.

³¹<https://getcomposer.org/>

Если библиотека не поставляется с автозагрузчиком классов, но конвенция именования ее классов соответствует PSR-4³², для загрузки ее классов можно использовать автозагрузчик Yii. Для этого достаточно для каждого корневого пространства имен, которые используются в ее классах, объявить **корневой псевдоним**. Предположим, что библиотека установлена в директорию `vendor/foo/bar`, а ее классы объявлены в корневом пространстве имен `xyz`. В конфигурации приложения можно использовать следующий код:

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

Если ни один из предыдущих вариантов не подходит, скорее всего для использования библиотеки нужно настроить в конфигурации РНР директиву `include_path`. Настройте ее, следуя инструкциям, которые поставляются с библиотекой.

В наихудшем случае библиотека требует явного подключения всех файлов, содержащих классы. При этом для подключения классов по требованию можно сделать следующее:

- Определить, какие классы входят в состав библиотеки.
- Перечислить классы и пути к соответствующим файлам в `Yii::$classMap` во **входном скрипте** приложения. Например,

```
Yii::$classMap['Class1'] = 'path/to/Class1.php';
Yii::$classMap['Class2'] = 'path/to/Class2.php';
```

14.9.2 Использование Yii в сторонних системах

Поскольку в Yii реализована масса полезных функций, они могут пригодиться при разработке или расширении сторонних систем, таких как WordPress и Joomla, или приложений, разработанных с помощью других РНР-фреймворков. Например, в сторонней системе можно задействовать класс `yii\helpers\ArrayHelper` или использовать функционал **Active Record**. Для этого обычно нужно сделать две вещи: установить Yii и подключить Yii.

Если сторонняя система использует для управления зависимостями Composer, Yii можно просто установить с помощью следующих команд:

```
composer global require "fxp/composer-asset-plugin:^1.2.0"
composer require yiisoft/yii2
composer install
```

Первая команда устанавливает `composer asset plugin`³³, который позволяет управлять зависимостями пакетов bower и npm через Composer. Даже

³²<http://www.php-fig.org/psr/psr-4/>

³³<https://github.com/francoispluchino/composer-asset-plugin/>

если вы хотите воспользоваться слоем абстракции баз данных или другими элементами Yii, не связанными с ресурсами, этот плагин все равно придется установить, так как без него не установится пакет Yii. В разделе **об установке Yii** более подробно описана работа с Composer и даны решения проблем, которые могут возникнуть при установке.

Также можно скачать³⁴ файл релиза Yii и распаковать его в директорию `BasePath/vendor`.

Далее следует изменить входной скрипт сторонней системы, поместив в его начало следующий код:

```
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');  
  
$yiiConfig = require(__DIR__ . '/../config/yii/web.php');  
new yii\web\Application($yiiConfig); // НЕ ВЫЗЫВАЙТЕ run() в этом месте
```

Как видите, этот код очень похож на код входного скрипта типичного приложения Yii. Единственное отличие заключается в том, что после создания экземпляра приложения не вызывается метод `run()`. Это связано с тем, что при вызове `run()` Yii захватывает контроль над процессом обработки запроса, что в данном случае не требуется, так как эту задачу выполняет существующее приложение.

Как и в случае с приложением Yii, нужно настроить экземпляр приложения исходя из окружения запущенной сторонней системы. Например, чтобы воспользоваться функционалом **Active Record**, нужно передать в **компонент приложения** `$db` настройки для подключения к базе данных, которую использует сторонняя система.

Это позволит задействовать большинство функционала, который предоставляет Yii. Например, можно будет создавать классы типа **Active Record**, и с их помощью взаимодействовать с базой данных.

14.9.3 Использование Yii 2 в связке с Yii 1

Если в прошлом вам приходилось использовать Yii 1, не исключено, что у вас до сих пор где-то используются приложения на этой платформе. Вместо того, чтобы переписывать все приложение под Yii 2, может быть целесообразно расширить его используя отдельные функции, которые появились в Yii 2. Для этого нужно выполнить следующие действия.

Примечание: Yii 2 требует PHP 5.4 или выше. Убедитесь, что и сервер, и существующее приложение поддерживают это.

Во-первых, установите Yii 2 в существующем приложении, выполняя действия, описанные в предыдущем подразделе.

Во-вторых, внесите следующие изменения во входной скрипт приложения:

³⁴<http://www.yiiframework.com/download/>

```
// подключение модифицированного класса Yii, описанного ниже
require(__DIR__ . '/../components/Yii.php');

// настройка приложения Yii 2
$yii2Config = require(__DIR__ . '/../config/yii2/web.php');
new yii\web\Application($yii2Config); // НЕ ВЫЗЫВАЙТЕ run()

// настройка приложения Yii 1
$yii1Config = require(__DIR__ . '/../config/yii1/main.php');
Yii::createWebApplication($yii1Config)->run();
```

Так как класс `Yii` используется и в `Yii 1`, и в `Yii 2`, нужно будет создать его модифицированную версию, обслуживающую обе версии фреймворка. В приведенном выше коде подключается модифицированный файл класса `Yii` со следующим содержанием:

```
$yii2path = '/path/to/yii2';
require($yii2path . '/BaseYii.php'); // Yii 2.x

$yii1path = '/path/to/yii1';
require($yii1path . '/YiiBase.php'); // Yii 1.x

class Yii extends \yii\BaseYii
{
    // скопируйте и вставьте код из YiiBase (1.x)
}

Yii::$classMap = include($yii2path . '/classes.php');
// регистрация автозагрузчика Yii 2 через Yii 1
Yii::registerAutoloader(['Yii', 'autoload']);
// создание контейнера внедрения зависимостей
Yii::$container = new yii\di\Container;
```

Вот и все! Теперь в любом месте кода можно с помощью конструкции `Yii::$app` получить доступ к экземпляру приложения `Yii 2`, а с помощью конструкции `Yii::app()` - к экземпляру приложения `Yii 1`:

```
echo get_class(Yii::app()); // выводит 'CWebApplication'
echo get_class(Yii::$app);  // выводит 'yii\web\Application'
```


Глава 15

Виджеты

Error: not existing file: <https://github.com/yiisoft/yii2-bootstrap/blob/master/d>

Error: not existing file: <https://github.com/yiisoft/yii2-jui/blob/master/docs/guide/README.md>

Глава 16

Хелперы

16.1 Хелперы

Примечание: Этот раздел находится в стадии разработки.

Yii предоставляет много классов, которые помогают упростить общие задачи программирования, такие как манипуляция со строками или массивами, генерация HTML кода, и так далее. Все helper классы организованы в рамках пространства имен `yii\helpers` и являются статическими методами (это означает, что они содержат в себе только статические свойства и методы и объекты статического класса создать нельзя).

Вы можете использовать helper класс с помощью вызова одного из статических методов, как показано ниже:

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

Примечание: Помощь в настройке helper классов, в Yii каждый основной helper состоит из двух классов: базовый класс (например `BaseArrayHelper`) и конкретный класс (например `ArrayHelper`). Когда вы используете helper, вы должны использовать только конкретные версии классов и никогда не использовать базовые классы.

16.1.1 Встроенные хелперы

В этой версии Yii предоставляются следующие основные helper классы:

- [ArrayHelper](#)
- [Console](#)
- [FileHelper](#)
- [FormatConverter](#)
- [Html](#)

- HtmlPurifier
- Imagine (provided by yii2-imagine extension)
- Inflector
- Json
- Markdown
- StringHelper
- [Url](#)
- VarDumper

16.1.2 Настройка хелперов

Для настройки основных helper классов (например `yii\helpers\ArrayHelper`), вы должны создать расширяющийся класс из помощников соответствующих базовых классов (например `yii\helpers\BaseArrayHelper`) и дать похожее название, вашему классу, с соответствующим конкретному классу (например `yii\helpers\ArrayHelper`), в том числе его пространство имен. Тогда созданный класс заменит оригинальную реализацию в фреймворке.

В следующих примерах показывается как настроить `merge()` метод `yii\helpers\ArrayHelper` класса:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // your custom implementation
    }
}
```

Сохраните ваш класс в файле с именем `ArrayHelper.php`. Файл должен находиться в другой директории, например `@app/components`.

Далее, в приложении [входной скрипт](#), добавьте следующую строку кода после подключения `yii.php` файла, которая сообщит [автозагрузка классов Yii](#) загрузить ваш класс вместо оригинального helper класса фреймворка:

```
Yii::$classMap['yii\helpers\ArrayHelper'] = '@app/components/ArrayHelper.php';
```

Обратите внимание что пользовательская настройка helper классов полезна только, если вы хотите изменить поведение существующей функции helper классов. Если вы хотите добавить дополнительные функции, для использования в вашем приложении, будет лучше создать отдельный helper.

16.2 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.

16.2.1 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, 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');
```

¹<http://php.net/manual/en/book.array.php>

В случае если вы хотите получить значение и тут же удалить его из массива, вы можете использовать метод `remove`

```
$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.

16.2.2 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.";
}
```

16.2.3 Извлечение столбцов

Часто нужно извлечь столбец 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'];
});
```

²<http://php.net/manual/en/function.array-key-exists.php>

16.2.4 Переиндексация массивов

Чтобы проиндексировать массив в соответствии с определенным ключом, используется метод `index`. Входящий массив должен быть многомерным 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'];
});
```

16.2.5 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',  
//    ],  
//  ]
```

16.2.6 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.

16.2.7 Определение типа массива

Удобный способ для определения, является массив индексным или ассоциативным. Вот пример:

³<http://php.net/manual/en/function.sort.php>


```
// no keys specified
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// all keys are strings
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

16.2.8 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 установить второй параметр в `false`, то ключи массива будут так же кодированы. Кодирование использует кодировку приложения, которая может быть изменена с помощью третьего аргумента.

16.2.9 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 слияния третьим, четвертым, пятым (и так далее) параметром.

```
ArrayHelper::merge($a, $b);
```

16.2.10 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();
$data = ArrayHelper::toArray($posts, [
    '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 `Arrayable` в этом классе

16.2.11 Проверка на присутствие в массиве

Часто необходимо проверить, содержится ли элемент в массиве, или является ли массив подмножеством другого массива. К сожалению, PHP функция `in_array()` не поддерживает подмножества объектов, реализующих интерфейс `\Traversable`.

Для таких случаев `yii\helpers\ArrayHelper` предоставляет `isIn()` и `isSubset()`. Методы принимают такие же параметры, что и `in_array()`⁴.

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new(ArrayObject['a']));

// true
ArrayHelper::isSubset(new(ArrayObject['a', 'c']), new(ArrayObject['a', 'b', 'c']));
```

⁴<http://php.net/manual/en/function.in-array.php>

16.3 Html-помощник

Каждое веб-приложение формирует большое количество HTML-разметки. Если разметка статическая, её можно эффективно сформировать смешиванием PHP и HTML в одном файле⁵, но когда разметка динамическая, становится сложно формировать её без дополнительной помощи. Yii предоставляет такую помощь в виде Html-помощника, который обеспечивает набор статических методов для обработки часто-используемых HTML тэгов, их атрибутов и содержимого.

Примечание: Если ваша разметка близка к статической, лучше использовать непосредственно HTML. Нет никакой необходимости в том, чтобы всё подряд оборачивать вызовами Html-помощника.

16.3.1 Основы

Так как формирование динамической HTML-разметки при помощи конкатенации строк очень быстро вносит хаос в проект, Yii предоставляет набор методов для управления атрибутами тэгов и формирования тэгов на основе этих атрибутов.

Формирование тэгов

Код формирования тэга выглядит следующим образом:

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

Здесь первый аргумент - это название тэга. Второй - содержимое, которое будет заключено между открывающим и закрывающим тэгами. Обратите внимание, что мы используем `Html::encode`. Это связано с тем, что содержимое не экранируется автоматически, чтобы можно было по-необходимости использовать чистый HTML. Третий аргумент - это массив настроек для HTML-кода, а другими словами - массив атрибутов для тэга. В этом массиве ключи являются названиями атрибутов, например `class`, `href` или `target`, а значения в массиве являются значениями этих атрибутов.

Вышеприведённый код сформирует следующую HTML-разметку:

```
<p class="username">samdark</p>
```

В тех случаях, когда вам необходимо только закрыть или открыть тэг, можно использовать методы `Html::beginTag()` и `Html::endTag()`.

Дополнительные атрибуты используются во многих методах Html-помощника и в различных виджетах. Во всех этих случаях в дело вступают механизмы дополнительной обработки данных, о которых следует знать:

⁵<http://php.net/manual/ru/language.basic-syntax.phpmode.php>

- Если значение равно `null`, соответствующий атрибут не будет выведен.
- Атрибуты, значения которых имеют тип `boolean`, будут интерпретированы как логические атрибуты⁶.
- Значения атрибутов будут экранированы с использованием метода `Html::encode()`.
- Если в качестве значения атрибута передан массив, он будет обработан следующим образом:
 - Если атрибут является одним из атрибутов данных, указанных в `yii\helpers\Html::$dataAttributes`, например `data` или `ng`, то будет сформирован список атрибутов по одному для каждого элемента массива. Например, код `'data' => ['id' => 1, 'name' => 'yii']` сформирует `data-id="1" data-name="yii"`; а код `'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok']` сформирует `data-params='{ "id":1, "name":"yii" }' data-status="ok"`. Заметьте, что в последнем примере используется формат JSON для формирования вывода вложенного массива.
 - Если атрибут НЕ является атрибутом данных, значение будет сконвертировано в формат JSON. Например, код `['params' => ['id' => 1, 'name' => 'yii']]` сформирует `params='{ "id":1, "name":"yii" }'`.

Формирование CSS классов и стилей

При формировании атрибутов для HTML-тэгов часто приходится начинать с некоторых атрибутов по умолчанию, которые затем необходимо изменять. Для того, чтобы добавить или удалить CSS-класс, можно использовать следующий подход:

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Всё получилось!', $options);

// в случае, если $type содержит строку 'success', будет сформирован вывод
// <div class="btn btn-successВсё получилось!> получилось!</div>
```

Можно указать несколько CSS-классов, используя синтаксис массива:

```
$options = ['class' => ['btn', 'btn-default']];

echo Html::tag('div', 'Сохранить', $options);
// выведет '<div class="btn btn-defaultСохранить"></div>'
```

⁶<http://www.w3.org/TR/html5/infrastructure.html#boolean-attributes>

При добавлении или удалении классов тоже можно использовать массивы:

```
$options = ['class' => 'btn'];

if ($type === 'success') {
    Html::addCssClass($options, ['btn-success', 'btn-lg']);
}

echo Html::tag('div', 'Сохранить', $options);
// выведет '<div class="btn btn-success btn-lgСохранить"></div>'
```

Html::addCssClass() предотвращает дублирование классов, поэтому можно не беспокоиться о том, что какой-либо класс будет добавлен дважды:

```
$options = ['class' => 'btn btn-default'];

Html::addCssClass($options, 'btn-default'); // класс 'btn-default' уже
добавлен

echo Html::tag('div', 'Сохранить', $options);
// выведет '<div class="btn btn-defaultСохранить"></div>'
```

Если CSS-класс задаётся с помощью массива, можно использовать именованные ключи массива для обозначения логического предназначения класса. В этом случае класс с таким же ключом будет проигнорирован во время использования Html::addCssClass():

```
$options = [
    'class' => [
        'btn',
        'theme' => 'btn-default',
    ]
];

Html::addCssClass($options, ['theme' => 'btn-success']); // ключ 'theme'
уже использован

echo Html::tag('div', 'Сохранить', $options);
// отобразит '<div class="btn btn-defaultСохранить"></div>'
```

CSS-стили могут быть установлены схожим образом с помощью атрибута style:

```
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// в результате будет style="width: 100px; height: 200px; position: absolute;"
Html::addCssStyle($options, 'height: 200px; position: absolute;');

// в результате будет style="position: absolute;"
Html::removeCssStyle($options, ['width', 'height']);
```

При использовании метода addCssStyle() можно указать массив, пары ключ-значение которого соответствуют названиям и значениям CSS-

свойств, или строку, например `width: 100px; height: 200px;`. Эти два формата могут быть преобразованы друг в друга с помощью методов `cssStyleFromArray()` и `cssStyleToArray()`. Метод `removeCssStyle()` принимает на вход массив атрибутов, которые следует удалить. Если удаляется всего один атрибут, его можно передать строкой.

Экранирование контента

Для корректного и безопасного отображения контента специальные символы в HTML-коде должны быть экранированы. В чистом PHP это осуществляется с помощью функций `htmlspecialchars7` и `htmlspecialchars_decode8`. Проблема использования этих функций заключается в том, что приходится указывать кодировку и дополнительные флаги во время каждого вызова. Поскольку флаги всё время одинаковы, а кодировка остаётся одной и той же в пределах приложения, Yii в целях безопасности предоставляет два компактных и простых в использовании метода:

```
$userName = Html::encode($user->name);  
echo $userName;  
  
$decodedUserName = Html::decode($userName);
```

16.3.2 Формы

Разметка форм состоит из повторяющихся действий и часто приводит к ошибкам, поэтому есть целый набор методов, которые помогают справиться с этой задачей.

Примечание: Рассмотрите возможность использования `ActiveForm`, если работаете с моделями и нуждаетесь в валидации данных.

Создание форм

Открывающий тэг формы может быть выведен с помощью метода `beginForm()` как показано ниже:

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' => 'multipart/form-data']) ?>
```

Первый аргумент - это URL-адрес, на который будет отправлена форма. Он может быть задан в виде Yii-маршрута и параметров, подходящих для передачи в метод `Url::to()`. Второй аргумент - способ отправки данных: по умолчанию это `post`. Третий аргумент - массив атрибутов формы. В данном примере мы изменяем способ кодирования данных в

⁷<http://www.php.net/manual/ru/function.htmlspecialchars.php>

⁸<http://www.php.net/manual/ru/function.htmlspecialchars-decode.php>

POST-запросе на `multipart/form-data`. Это необходимо для загрузки файлов.

Закрыть тэг формы можно простым кодом:

```
<?= Html::endForm() ?>
```

Кнопки

Для формирования кнопок можно использовать следующий код:

```
<?= Html::button('Нажми меня!', ['class' => 'teaser']) ?>
<?= Html::submitButton('Отправить', ['class' => 'submit']) ?>
<?= Html::resetButton('Сбросить', ['class' => 'reset']) ?>
```

Первый аргумент во всех трёх методах - это название кнопки, а второй - её атрибуты. Название кнопки не экранируется, поэтому при получении данных от конечных пользователей экранируйте их с помощью метода `Html::encode()`.

Поля ввода

Методы формирования полей ввода делятся на две группы. Первые начинаются со слова `active` и называются “active inputs”, а вторые не содержат в своём названии слова `active`. “Active inputs” формируют поля ввода, которые получают данные из указанного атрибута модели, в то время как обычные методы формирования полей ввода непосредственно принимают данные.

Наиболее общие методы для формирования полей ввода:

```
// тип, название поля ввода, установленное в поле значение, атрибуты поля
// ввода
<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>

// тип, модель, атрибут модели, атрибуты поля ввода
<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

Если вам заранее известен тип поля ввода, удобнее будет пользоваться этими вспомогательными методами:

- `yii\helpers\Html::buttonInput()`
- `yii\helpers\Html::submitInput()`
- `yii\helpers\Html::resetInput()`
- `yii\helpers\Html::textInput()`, `yii\helpers\Html::activeTextInput()`
- `yii\helpers\Html::hiddenInput()`, `yii\helpers\Html::activeHiddenInput()`
- `yii\helpers\Html::passwordInput()` / `yii\helpers\Html::activePasswordInput()`
- `yii\helpers\Html::fileInput()`, `yii\helpers\Html::activeFileInput()`
- `yii\helpers\Html::textarea()`, `yii\helpers\Html::activeTextarea()`

Сигнатура методов для формирования радио-переключателей и чекбоксов немного отличается:

```
<?= Html::radio('agree', true, ['label' => 'Я согласен']);
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement'])

<?= Html::checkbox('agree', true, ['label' => 'Я согласен']);
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement'])
```

Выпадающие и обычные списки могут быть сформированы следующим образом:

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels,
    'id', 'name')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels, 'id',
    'name')) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels, 'id',
    'name')) ?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id', '
    name')) ?>
```

Первый аргумент - это значение атрибута name для поля ввода, второй - выбранное в нём значение и, наконец, третий аргумент - это массив, ключами которого является список значений для формирования списка, а значениями массива являются названия пунктов в нём.

Если вы хотите сделать в списке доступными для выбора несколько вариантов, хорошим выбором будет список чекбоксов:

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels, 'id',
    'name')) ?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels, '
    id', 'name')) ?>
```

Если же нет, используйте радио-переключатель:

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id', '
    name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels, 'id',
    'name')) ?>
```

Тэги label и отображение ошибок

Также как и для полей ввода, есть два метода формирования тэгов label для форм. Есть “active label”, считывающий данные из модели и обычный тэг “label”, принимающий на вход непосредственно сами данные:

```
<?= Html::label('Имя пользователя', 'username', ['class' => 'label username'
    ]) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username'])
```

Для отображения общего списка ошибок формы на основе модели или моделей можно использовать следующий подход:

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

Для отображения одной отдельной ошибки:

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```


Атрибуты name и value для полей ввода

Также имеются методы для получения значений атрибутов id, name и value для полей ввода, сформированных на основе моделей. Эти методы используются в основном внутренними механизмами, но иногда могут оказаться подходящими и для прямого использования:

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// моё первое сообщение
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

В вышеприведённом коде первый аргумент - это модель, а второй аргумент - выражение для поиска атрибута модели. В самом простом случае оно представляет собой название атрибута модели, а вообще это может быть название, которому предшествует (и/или после которого следует) индекс массива. Такой формат используется в основном для табличного ввода данных:

- [0]content используется для табличного ввода данных, чтобы указать на атрибут “content” первой модели табличного ввода;
- dates[0] указывает на первый элемент массива, с помощью которого задан атрибут модели “dates”;
- [0]dates[0] указывает на первый элемент массива, с помощью которого задан атрибут “dates” первой модели табличного ввода.

Для того, чтобы получить название атрибута модели в чистом виде (без суффиксов и префиксов), можно использовать следующий подход:

```
// dates
echo Html::getAttributeName('dates[0]');
```

16.3.3 Подключение встроенных стилей и скриптов

Для формирования встроенных скриптов и стилей есть два метода:

```
<?= Html::style('.danger { color: #f00; }') ?>Результатом
будет:
<style>.danger { color: #f00; }</style>

<?= Html::script('alertПривет("!");', ['defer' => true]);Результатом
будет:
```

```
<script defer>alert("Привет!");</script>
```

Если вы хотите подключить внешний CSS-файл:

```
<?= Htm1::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>В  
результате получится:  
  
<!--[if IE 5]>  
  <link href="http://example.com/css/ie5.css" />  
<![endif]-->
```

Первый аргумент - URL-адрес. Второй - массив настроек. Помимо обычных настроек можно указать следующие:

- `condition` для оборачивания тэга `<link>` с помощью условных комментариев с определённым условием. Надеемся, что вам никогда не понадобятся условные комментарии ;)
- `noscript` может быть установлен в значение `true`, чтобы обернуть тэг `<link>` с помощью тэга `<noscript>`, таким образом скрипт будет подключён только в том случае, если у пользователя в браузере нет поддержки JavaScript или же пользователь сам отключил его.

Для подключения JavaScript-файла используйте код:

```
<?= Htm1::jsFile('@web/js/main.js') ?>
```

Как и в случае с CSS, первый аргумент указывает ссылку на файл, который должен быть подключен. Настройки задаются во втором аргументе. Можно указать настройку `condition` таким же образом, каким она указывается для метода `cssFile`.

16.3.4 Ссылки

Существует удобный метод формирования ссылок:

```
<?= Htm1::a('Профиль', ['user/view', 'id' => $id], ['class' => 'profile-link']) ?>
```

Первый аргумент - это текст ссылки. Он не экранируется, поэтому при использовании данных, полученных от конечных пользователей, необходимо экранировать его с помощью `Htm1::encode()`. Второй аргумент - это содержимое атрибута `href` тэга `<a>`. Смотрите `Url::to()` для получения подробностей относительно значений, которые могут быть переданы в качестве второго аргумента. Третий аргумент - массив атрибутов для тэга.

Если нужно сформировать ссылку `mailto`, можно использовать следующий подход:

```
<?= Htm1::mailto('Обратная связь', 'admin@example.com') ?>
```

16.3.5 Изображения

Для формирования тэга изображения используйте следующий код:

```
<?= Html::img('@web/images/logo.png', ['alt' => 'Наш логотип']) ?>
```

результате получится:

```

```

Помимо псевдонимов первый аргумент может принимать маршруты, параметры и URL-адреса таким же образом как и метод `Url::to()`.

16.3.6 Списки

Неупорядоченные списки могут быть сформированы следующим образом:

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
        ['class' => 'post']
    );
}]) ?>
```

Для формирования упорядоченных списков используйте `Html::ol()`.

16.4 Url хелпер

Url хелпер предоставляет набор статических методов для управления URL.

16.4.1 Получение общих URL

Вы можете использовать два метода получения общих URL: домашний URL (Home) и базовый URL (Base) текущего запроса. Используйте следующий код, чтобы получить домашний URL:

```
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

Если вы не передали параметров, то получите относительный URL. Вы можете передать `true`, чтобы получить абсолютный URL для текущего протокола или явно указать протокол (`https`, `http`).

Чтобы получить базовый URL текущего запроса:

```
$relativeBaseUrl = Url::base();
$absoluteBaseUrl = Url::base(true);
$httpsAbsoluteBaseUrl = Url::base('https');
```

Единственный параметр данного метода работает также как и `Url::home()`.

16.4.2 Создание URL

Чтобы создать URL для соответствующего роута используйте метод `Url::toRoute()`. Метод использует `yii\web\UrlManager`. Для того чтобы создать URL:

```
$url = Url::toRoute(['product/view', 'id' => 42]);
```

Вы можете задать роут строкой, например, `site/index`. А также вы можете использовать массив, если хотите задать дополнительные параметры запроса для URL. Формат массива должен быть следующим:

```
// сгенерирует: /index.php?r=site/index&param1=value1&param2=value2  
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

Если вы хотите создать URL с якорем, то вы можете использовать параметр массива с ключом `#`. Например:

```
// сгенерирует: /index.php?r=site/index&param1=value1#name  
['site/index', 'param1' => 'value1', '#' => 'name']
```

Роут может быть и абсолютным, и относительным. Абсолютный URL начинается со слеша (например, `/site/index`), относительный - без (например, `site/index` or `index`). Относительный URL будет сконvertирован в абсолютный по следующим правилам:

- Если роут пустая строка, то будет использовано текущее значение `route`;
- Если роут не содержит слешей (например, `index`), то он будет считаться экшеном текущего контролера и будет определен с помощью `yii\web\Controller::$uniqueId`;
- Если роут начинается не со слеша (например, `site/index`), то он будет считаться относительным роутом текущего модуля и будет определен с помощью `uniqueId`.

Начиная с версии 2.0.2, вы можете задавать роуты с помощью **псевдонимов**. В этом случае, сначала псевдоним будет сконvertирован в соответствующий роут, который будет преобразован в абсолютный в соответствии с вышеописанными правилами.

Примеры использования метода:

```
// /index.php?r=site/index  
echo Url::toRoute('site/index');  
  
// /index.php?r=site/index&src=ref1#name  
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);  
  
// /index.php?r=post/edit&id=100      псевдоним "@postEdit" задан как "post/  
edit"  
echo Url::toRoute(['@postEdit', 'id' => 100]);  
  
// http://www.example.com/index.php?r=site/index  
echo Url::toRoute('site/index', true);
```

```
// https://www.example.com/index.php?r=site/index
echo Url::toRoute('site/index', 'https');
```

Другой метод `Url::to()` очень похож на `toRoute()`. Единственное отличие: входным параметром должен быть массив. Если будет передана строка, то она будет воспринята как URL.

Первый аргумент может быть:

- массивом: будет вызван `toRoute()`, чтобы сгенерировать URL. Например: `['site/index'], ['post/index', 'page' => 2]`. В разделе `toRoute()` подробно описано как задавать роут;
- Строка, начинающаяся с `@`, будет обработана как псевдоним. Будет возвращено соответствующее значение псевдонима;
- Пустая строка: вернет текущий URL;
- Обычная строка: вернет строку без изменений

Когда у метода задан второй параметр `$scheme` (строка или `true`), то сгенерированный URL будет с протоколом (полученным из `yii\web\UrlManager::$hostInfo`). Если в `$url` указан протокол, то его значение будет заменено.

Пример использования:

```
// /index.php?r=site/index
echo Url::to(['site/index']);

// /index.php?r=site/index&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post/edit&id=100      псевдоним "@postEdit" задан как "post/edit"
echo Url::to(['@postEdit', 'id' => 100]);

// Текущий URL
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// http://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

Начиная с версии 2.0.3, вы можете использовать `yii\helpers\Url::current()`, чтобы создавать URL на основе текущего запрошенного роута и его GET-параметров. Вы можете изменить, удалить или добавить новые GET-параметры передав в метод параметр `$params`. Например:

```
// предположим $_GET = ['id' => 123, 'src' => 'google'], а текущий роут "  
post/view"  
  
// /index.php?r=post/view&id=123&src=google  
echo Url::current();  
  
// /index.php?r=post/view&id=123  
echo Url::current(['src' => null]);  
// /index.php?r=post/view&id=100&src=google  
echo Url::current(['id' => 100]);
```

16.4.3 Запоминание URL

Существуют задачи, когда вам необходимо запомнить URL и потом использовать его в процессе одного или нескольких последовательных запросов. Это может быть достигнуто следующим образом:

```
// Запомнить текущий URL  
Url::remember();  
  
// Запомнить определенный URL. Входные параметры смотрите на примере Url::to()  
Url::remember(['product/view', 'id' => 42]);  
  
// Запомнить URL под определенным именем  
Url::remember(['product/view', 'id' => 42], 'product');
```

В следующем запросе мы можем получить сохраненный URL следующим образом:

```
$url = Url::previous();  
$productUrl = Url::previous('product');
```

16.4.4 Проверить относительность URL

Чтобы проверить относительный URL или нет (например, если в нем не содержится информации о хосте), вы можете использовать следующий код:

```
$isRelative = Url::isRelative('test/it');
```