**ЛАБОРАТОРНАЯ РАБОТА № 2**

**НАСТРОЙКА БАЗОВЫХ ЭЛЕМЕНТОВ В YII 2 FRAMEWORK. РАБОТА С БАЗОЙ ДАННЫХ**

**ЧАСТЬ 1. Управление базовыми элементами**

***Формы и валидация***

Для создания формы необходимо:

* модель формы;
* action, который будет отвечать за получение данных;
* вид, в котором будет выведена форма.

Начнем с создания модели. В папке моделей создадим класс TestForm.

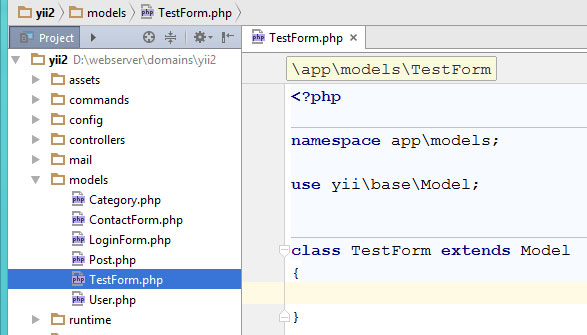
*namespace app\models;*

*use yii\base\Model;*

*class TestForm extends Model*

*{*

*}*



Модель наследует класс yii\base\Model, именно он, как правило, используется для работы с формами. В модели мы должны объявить публичные свойства, которые будут соответствовать полям формы. Например, пусть это будет форма с полями для ввода имени и email.

Объявим оба публичных свойства:

*class TestForm extends Model*

*{*

*public $name;*

*public $email;*

*}*

Создадим action для работы со страницей формы и данными из нее. Для этого необходимо создать контроллер TestController или воспользоваться любым уже имеющимся. Создадим действие actionPage и в этом действии создадим экземпляр модели и передадим его в представление.

*namespace app\controllers;*

*use yii\web\Controller;*

*use app\models\TestForm;*

*class TestController extends Controller*

*{*

*public function actionIndex()*

*{*

*return $this->render('index');*

*}*

*public function actionPage()*

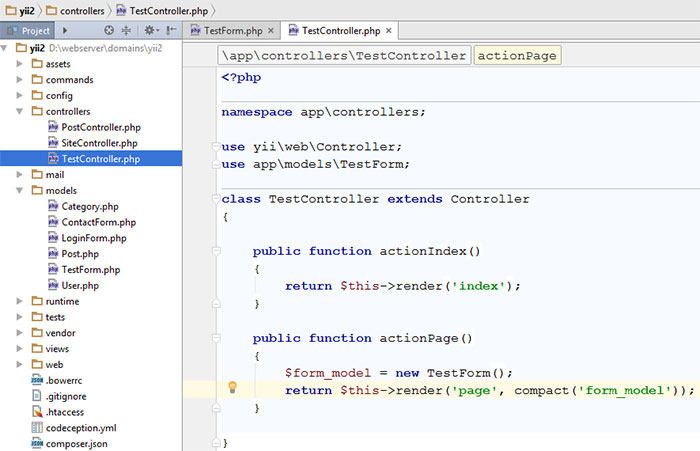
*{*

*$form\_model = new TestForm();*

*return $this->render('page', compact('form\_model'));*

*}*

*}*



Осталось в виде \views\test\page.php отрисовать форму. Для этого мы воспользуемся возможностями виджета yii\widgets\ActiveForm и хэлпером yii\helpers\Html.

*<?php*

*use yii\helpers\Html;*

*use yii\widgets\ActiveForm;*

*?>*

*<?php $form = ActiveForm::begin() ?>*

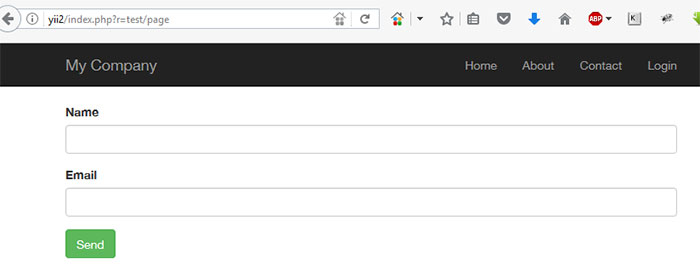
*<?= $form->field($form\_model, 'name') ?>*

*<?= $form->field($form\_model, 'email') ?>*

*<?= Html::submitButton('Send', ['class' => 'btn btn-success']) ?>*

*<?php ActiveForm::end() ?>*

Теперь обратимся к нужной нам странице, указав имя контроллера и действия: index.php?r=test/page. В результате мы должны увидеть нашу форму.



Теперь задача состоит в том, чтобы принять данные из формы. По умолчанию она будет отправлена в это же действие, т.е. TestController::actionPage – там и попробуем принять данные. Для получения данных методом POST мы воспользуемся методом post() класса yii\web\Request. Попробуем загрузить полученные данные в модель с помощью метода load() в действии. Для этого немного изменим код действия:

*public function actionPage()*

*{*

*$form\_model = new TestForm();*

*if($form\_model->load(\Yii::$app->request->post())){*

*var\_dump(\Yii::$app->request->post());*

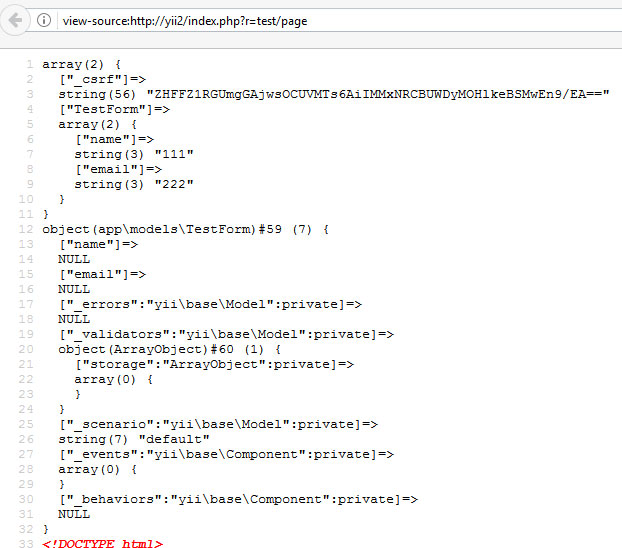
*var\_dump($form\_model);*

*}*

*return $this->render('page', compact('form\_model'));*

*}*

Здесь мы пытаемся загрузить данные в модель и затем распечатать ее. Также мы дополнительно распечатываем сами данные из массива $\_POST. Давайте попробуем отправить тестовые данные и взглянем на то, что будет распечатано.



В результате мы увидели, что данные приходят, но вот модель почему-то ими не заполняется, свойства name и email модели оказываются пустыми. Все правильно. Дело в том, что в модель могут быть загружены только проверенные данные, которые прошли валидацию. Для этого в модели нам потребуется создать массив правил валидации.

Здесь мы познакомимся лишь с некоторыми из валидаторов. С полным же их списком вы можете познакомиться в  к фреймворку Yii2 - <http://www.yiiframework.com/doc-2.0/guide-tutorial-core-validators.html> .

Итак, правила валидации полей формы мы должны описывать в методе rules() модели формы. Данный метод должен возвращать массив, в котором к свойствам модели (полям формы) применены те или иные валидаторы. Давайте используем первый валидатор, которым пользуются чаще других – это валидатор required. Как следует из названия, валидатор проверяет, заполнено ли поле. Если поле окажется пустым – валидатор вернет ошибку валидации.

*class TestForm extends Model*

*{*

*public $name;*

*public $email;*

*public function rules()*

*{*

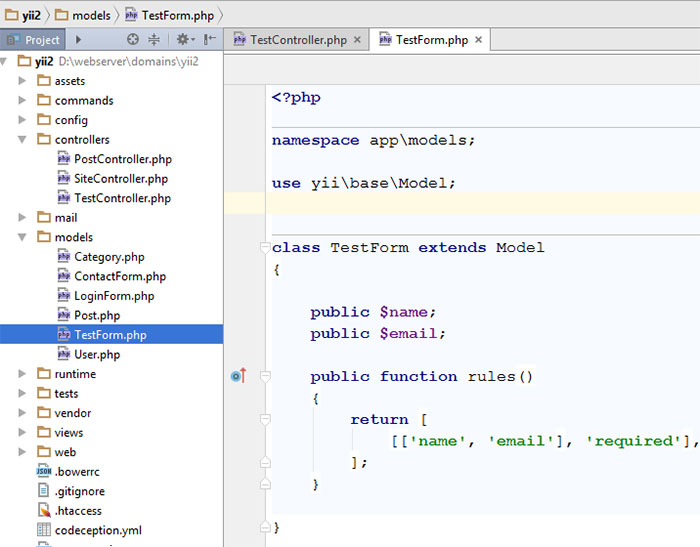
*return [*

*[['name', 'email'], 'required'],*

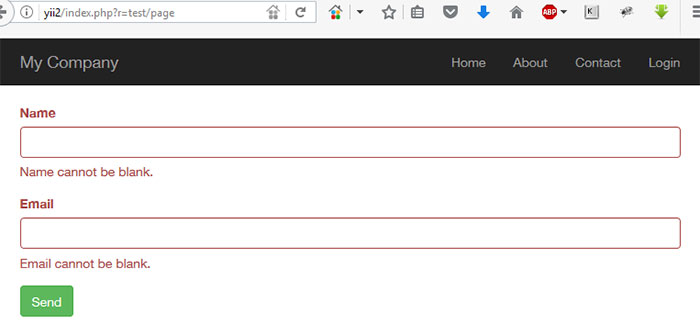
*];*

*}*

*}*



Давайте теперь попробуем вернуться к форме на странице, поместить курсор в поля формы и после убрать курсор, не заполняя поле.



Валидация работает и при том работает даже на клиенте, что очень удобно. Давайте используем еще один валидатор, который так и называется – email. Данный валидатор проверяет строку на соответствие формату адреса email.

*public function rules()*

*{*

*return [*

*[['name', 'email'], 'required'],*

*['email', 'email'],*

*];*

*}*



Как видим, новое правило валидации помогло проверить валидность email. Проверим теперь, сработает ли загрузка данных из формы в модель, отправив тестовые данные. Код действия, принимающего данные, будет пока что примерно таким:

*public function actionPage()*

*{*

*$form\_model = new TestForm();*

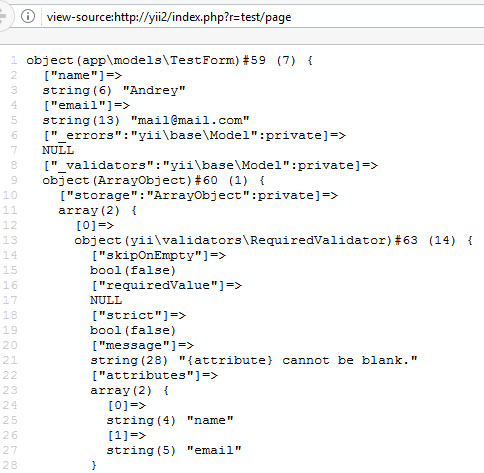
*if($form\_model->load(\Yii::$app->request->post())){*

*var\_dump($form\_model);*

*}*

*return $this->render('page', compact('form\_model'));*

*}*



Как видим, данные прошли валидацию и были успешно загружены в модель формы, соответствующие свойства модели получили значения, и мы можем работать с полученными данными.

**Виджеты**

**Виджеты** – это некоторые компоненты, которые предназначены в первую очередь для реализации элементов интерфейса и которые можно использовать многократно. Классическим примером такого элемента может быть меню. Меню используется практически на каждой странице вашего сайта и от страницы к странице остается неизменным. То есть, от страницы к странице мы должны повторять один и тот же код для получения меню и его вывода. Но благодаря виджетам мы можем избежать этой утомительной процедуры и создать меню только один раз, а затем просто вызывать его.

В Yii2 уже имеется большое количество различных виджетов: для создания форм, меню, jQuery UI и другие. Кроме того, фреймворк позволяет нам создавать собственные виджеты. Давайте на простом примере разберем создание простейшего виджета.

Создаются они в папке components, которую мы можем создать в корне приложения. Для создания виджета мы должны унаследовать класс yii\base\Widget. Внутри класса виджета мы можем переопределить методы init и run. Также можем создавать и собственные методы. Метод init, как правило, используется для нормализации свойств виджета (установка значений), а метод run возвращает результат рендеринга. Результат можно вывести с помощью echo или вернуть через return.

Давайте создадим простой пример виджета, который будет приветствовать посетителей сайта.

*class HiWidget extends Widget*

*{*

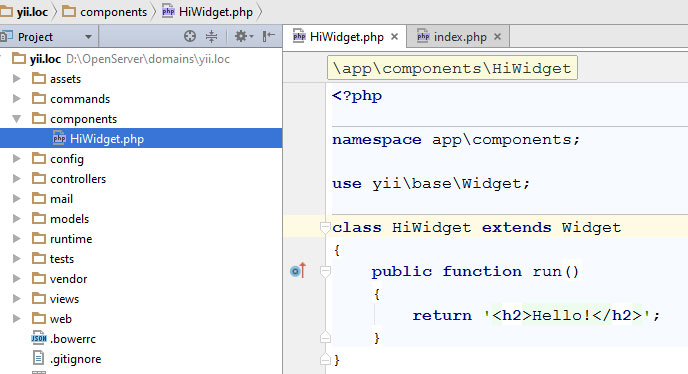
*public function run()*

*{*

*return '<h2>Hello!</h2>';*

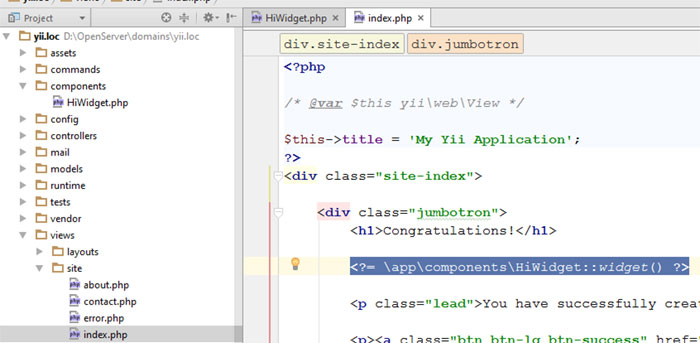
*}*

*}*



Теперь попробуем вызвать созданный виджет в представлении. Для этого необходимо обратиться к созданному классу и вызвать статичный метод widget.

*<?= \app\components\HiWidget::widget() ?>*



В результате на странице мы увидим выводимую приветственную строку.

Yii предлагает нам создать рядом с классом виджета папку views, в которую и поместить представление для виджета. Само представление можно вызвать методом render.

Создадим вид hi.php:

*<h2>Hello!</h2>*

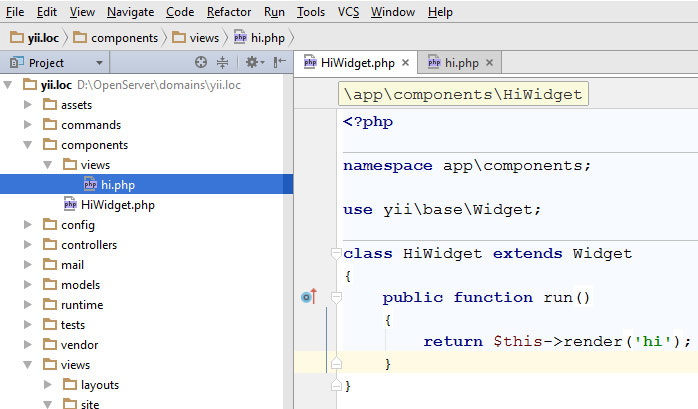
А в методе run вернем его:

*public function run()*

*{*

*return $this->render('hi');*

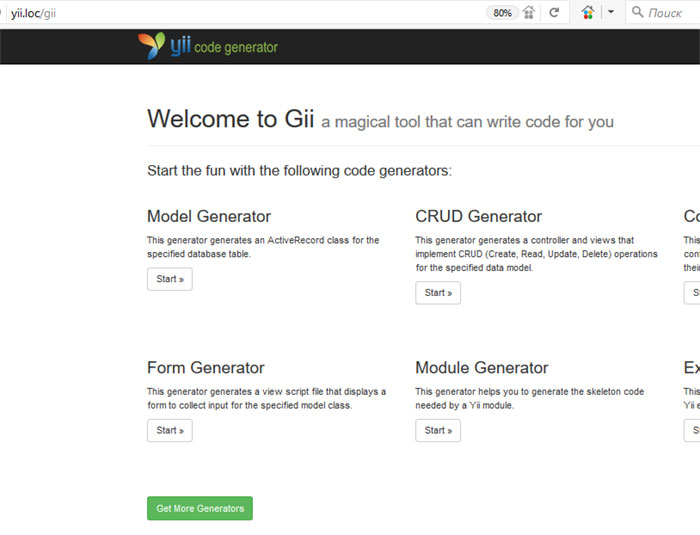
*}*



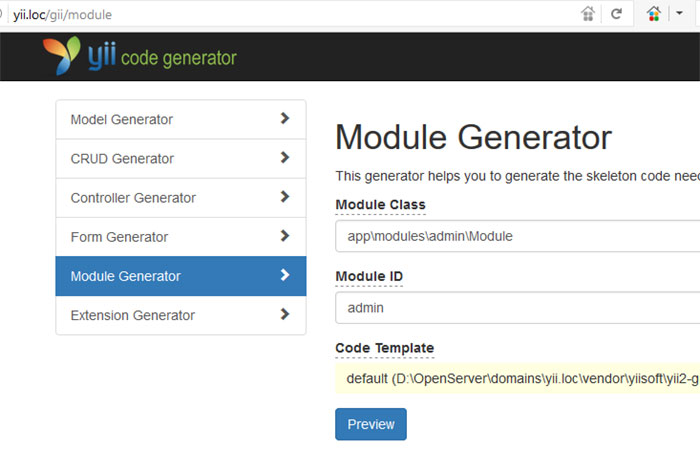
**Модули**

**Модули** – это по сути приложения в приложении. Это означает, что модуль может иметь свои контроллеры, модели, представления и прочие компоненты, как и основное приложение.

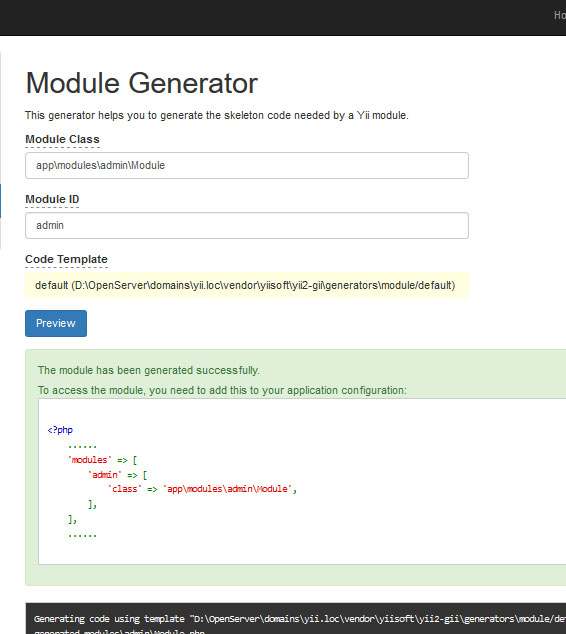
Простой способ создания модулей заключается в использовании генератора кода – модуля Gii.



Нажмем кнопку Start блока Module Generator и перейдем на страницу создания модуля, где заполним форму из двух полей. В форме укажем класс модуля и его ID.



Далее, как обычно, жмем кнопку Preview и на следующей странице – кнопку Generate. На новой странице нам сообщают об успешном создании модуля и предлагают скопировать блок кода и вставить его в файл config\web.php, данный блок кода включит созданный фреймворком модуль админки.



Откроем файл конфигурации веб-приложения и вставим в него предлагаемый код.

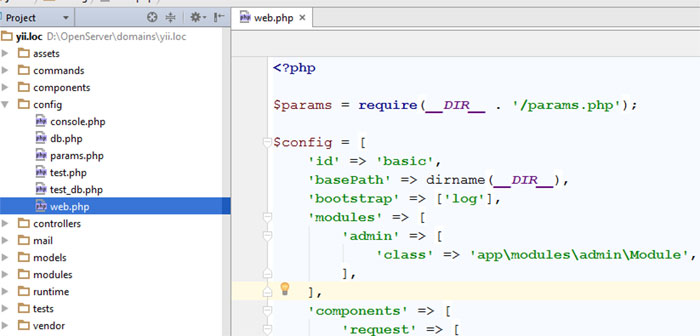
*'modules' => [*

*'admin' => [*

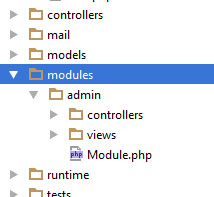
*'class' => 'app\modules\admin\Module',*

*,*

*,*



К слову, созданный модуль находится в папке modules, которая должна появиться в корне приложения Yii.

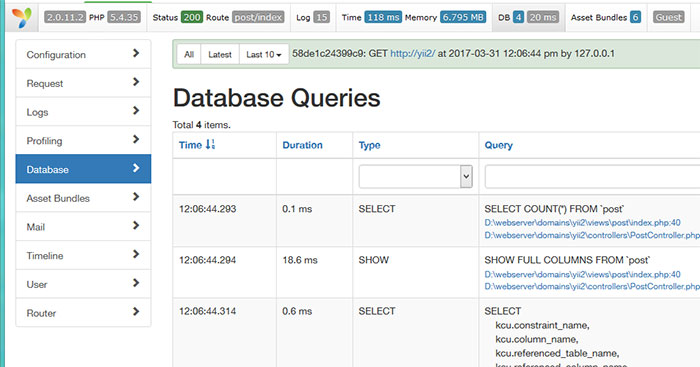


В данной папке дублируется структура приложения Yii, т.е. уже есть папка controllers с контроллерами, папка views с представлениями. Также мы можем создать папку models, в которой будут храниться модели модуля. В общем, как я и говорил выше, модуль – это приложение в приложении, законченный программный блок.

В папке контроллеров модуля уже создан дефолтный контроллер DefaultController, который мы и можем протестировать. Для этого достаточно набрать в URL адрес нашего модуля – http://yii.loc/admin/.

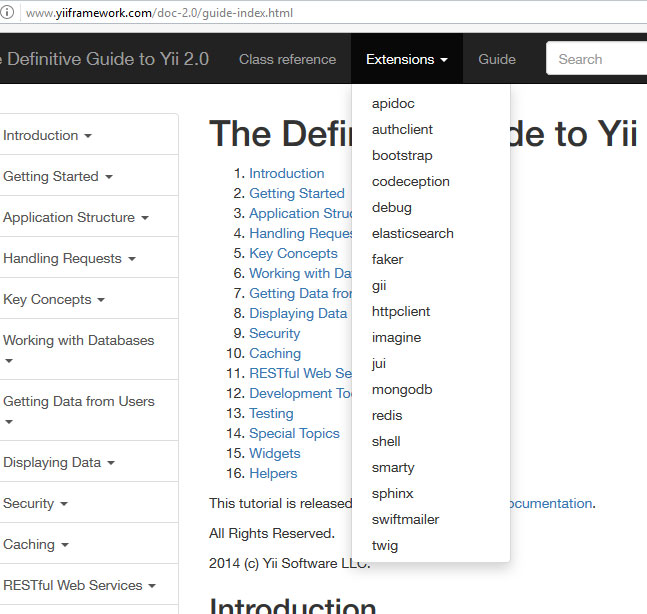
**Расширения**

**Расширения** – это готовые программные пакеты, позволяющие добавлять на сайт новый функционал. Например, в Yii2 из коробки есть готовое расширение Yii Debug, представляющее из себя удобную панель в нижней части страницы. В ней можно смотреть различную отладочную информацию: запросы к БД, всевозможные логи, содержимое переменных окружения и многое другое. Есть удобный модуль Gii, позволяющий генерировать типовой код.



Как вы понимаете, мы можем добавлять и использовать сторонние расширения. Использование сторонних расширений для фреймворка – достаточно простая процедура. Все что нам необходимо сделать – это установить расширение через Composer и при необходимости настроить его по инструкции. После чего расширение готово к работе и его можно использовать.

Где искать расширения? Они есть во множестве на GitHub. Также их можно найти и на [официальном сайте](http://www.yiiframework.com/doc-2.0/guide-index.html) в разделе Extensions.



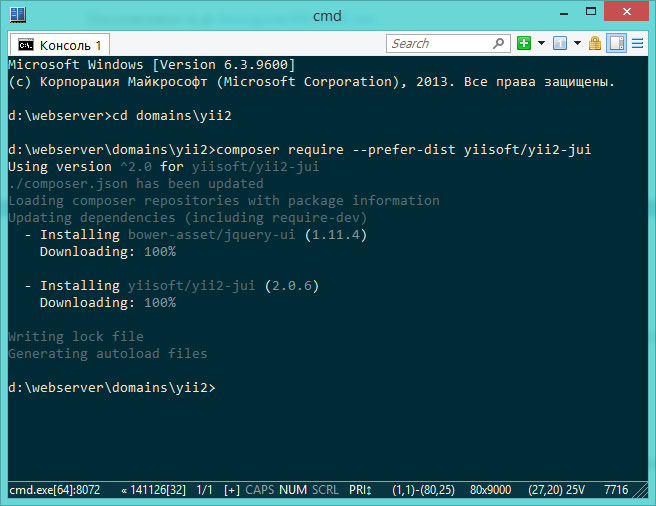
Давайте выберем из предложенного списка расширение [JUI](http://www.yiiframework.com/doc-2.0/ext-jui-index.html) – это набор виджетов jQuery UI, представленных в качестве виджетов Yii2.



На странице расширения мы можем найти варианты его установки, а также инструкцию по использованию данного расширения. Согласно инструкции выполним команду в консоли, чтобы установить расширение через Composer.

*composer require --prefer-dist yiisoft/yii2-jui*

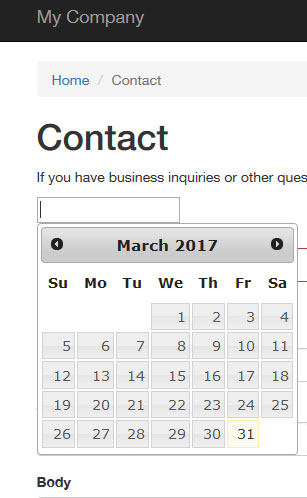
После некоторого ожидания мы должны увидеть сообщение об успешной установке расширения Yii2.



Теперь давайте попробуем использовать его согласно инструкции. Возьмем первый же предлагаемый к использованию код и вставим его в форму обратной связи в представлении views\site\contact.php:

*<?= yii\jui\DatePicker::widget(['name' => 'attributeName']) ?>*

Теперь взглянем на результат. В форме должно появиться новое поле, клик по которому вызовет виджет Datepicker – виджет календаря библиотеки jQueryUI.



**Изображения**

Для работы с загруженным файлом необходимо обратиться к классу yii\web\UploadedFile, объект которого и содержит всю информацию о загруженном файле.

Давайте начнем с создания модели, которая будет осуществлять валидацию загружаемого файла (нам ведь нужен не любой файл, а именно изображение) и сохранять его в выбранную папку. Пусть наша модель называется UploadImage.

*<?php*

*namespace app\models;*

*use yii\base\Model;*

*use yii\web\UploadedFile;*

*class UploadImage extends Model{*

*public $image;*

*public function rules(){*

*return[*

*[['image'], 'file', 'extensions' => 'png, jpg'],*

*];*

*}*

*public function upload(){*

*if($this->validate()){*

*var\_dump($this->image);*

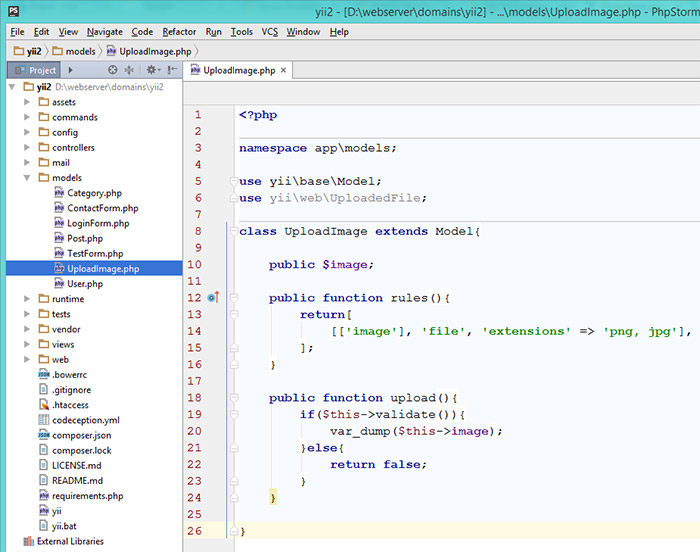
*}else{*

*return false;*

*}*

*}*

*}*



Как видим, в модели мы указали свойство $image, которое и будет отвечать за работу с загружаемым изображением. В методе rules использованы правила валидации для получаемого файла, которыми мы сообщаем, что во-первых, поле формы input должно быть корректным загружаемым файлом, а во-вторых, расширения этого файла должны быть png или jpg.

Ну и, наконец, у нас есть заготовка метода upload, который будет проверять, пройдена ли валидация, и пока что просто распечатывать на экран содержимое обеъекта.

Далее нам, конечно же, понадобится контроллер и представление. Давайте в классе SiteController создадим действие actionUpload, в котором и получим объект модели.

*public function actionUpload(){*

*$model = new UploadImage();*

*if(Yii::$app->request->isPost){*

*$model->image = UploadedFile::getInstance($model, 'image');*

*$model->upload();*

*return;*

*}*

*return $this->render('upload', ['model' => $model]);*

*}*



В экшене мы создаем объект модели и в свойство $image сохраняем информацию о загруженном файле. Также мы вызываем метод upload() модели, который написали на предыдущем шаге. Теперь создадим несложный вид – upload.php и обратимся к искомой странице с формой (http://yii2/site/upload):

*<?php*

*use yii\widgets\ActiveForm;*

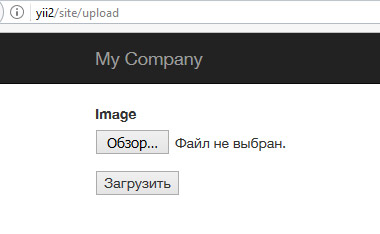
*?>*

*<?php $form = ActiveForm::begin() ?>*

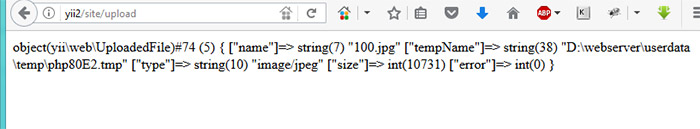
*<?= $form->field($model, 'image')->fileInput() ?>*

*<button>Загрузить</button>*

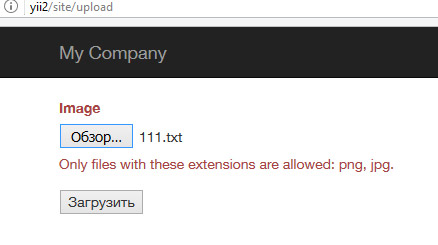
*<?php ActiveForm::end() ?>*



Давайте попробуем загрузить файл через созданную форму. Если мы нигде не ошиблись, тогда метод upload модели распечатает данные загруженного изображения.



К слову, если это будет не картинка с разрешенными расширениями, тогда мы должны увидеть другую картину:



Давайте теперь внесем необходимые правки в код, чтобы загрузить и вывести его. Для начала метод upload модели.

*public function upload(){*

*if($this->validate()){*

*$this->image->saveAs("uploads/{$this->image->baseName}.{$this->image->extension}");*

*}else{*

*return false;*

*}*

*}*

Здесь мы использовали метод saveAs, который сохранит файл по указанному пути. Мы будем сохранять картинку в папку uploads (создайте эту папку в директории web).

Теперь контроллер. Мы хотим не только загрузить картинку, но и реализовать ее вывод на экран.

*public function actionUpload(){*

*$model = new UploadImage();*

*if(Yii::$app->request->isPost){*

*$model->image = UploadedFile::getInstance($model, 'image');*

*$model->upload();*

*return $this->render('upload', ['model' => $model]);*

*}*

*return $this->render('upload', ['model' => $model]);*

*}*

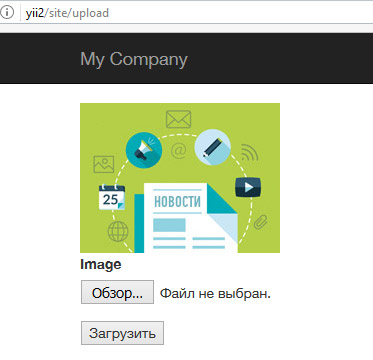
Ну и представление. Давайте проверим наличие изображения и добавим его вывод перед формой:

*<?php if($model->image): ?>*

*<img src="/uploads/<?= $model->image?>" alt="">*

*<?php endif; ?>*

В результате изображение должно быть загружено и успешно выведено сразу после сохранения в папку web\uploads.



**Задание**

1. Создать новую вкладку в меню сайта.
2. Обеспечить по ней вызов новой произвольной формы.
3. Добавить на форму произвольные поля и обеспечить их валидацию.
4. Добавить на форму 3 элемента с подключением к ним сторонних расширений.
5. Добавить на форму кнопку загрузки изображения.
6. Обеспечить просмотр добавляемого изображения.
7. Создать произвольный виджет для отображения изображения.

**ЧАСТЬ 2. Работа с БД**

***Создание сложных связей в БД***

С помощью PhpMyAdmin cоздадим новую таблицу для формирования статей (Post).

*CREATE TABLE IF NOT EXISTS `post` (*

*`id` int(10) unsigned NOT NULL AUTO\_INCREMENT,*

*`category\_id` int(10) unsigned NOT NULL,*

*`title` varchar(255) NOT NULL,*

*`excerpt` varchar(255) NOT NULL,*

*`text` text NOT NULL,*

*`keywords` varchar(255) DEFAULT NULL,*

*`description` varchar(255) DEFAULT NULL,*

*`created` date NOT NULL,*

*PRIMARY KEY (`id`)*

*) ENGINE=InnoDB  DEFAULT CHARSET=utf8;*

Здесь:

* id – ID статьи;
* category\_id – ссылка на категорию (которой принадлежит статья) будущей таблицу категорий;
* title – название статьи;
* excerpt – цитата статьи (часть текста статьи для списка статей);
* text – полный текст статьи;
* keywords – ключевые слова;
* description – мета-описание;
* created – дата создания статьи.

Теперь создадим модель и контроллер для работы с этой таблицей в папке Frontend.

Добавим в SiteController запись

***use*** *frontend\models\Post;*

Затем отредактируем функцию actionIndex()

*public function actionIndex()*

*{*

*$posts = Post::find()->all();*

*return $this->render('index', compact('posts'));*

}

Мы получили массив статей в виде массива объектов (post). Используя метод render() контроллера, мы передали полученные данные в представление. Теперь нужно вывести эти данные на главную страницу сайта. Для этого достаточно запустить цикл foreach и пройтись по массиву статей.

В site/index.php добавим следующий код:

*<h1>Список статей</h1>*

*<?php foreach($posts as $post): ?>*

*<div class="panel panel-default">*

*<div class="panel-heading">*

*<h3 class="panel-title"><?= $post->title ?></h3>*

*</div>*

*<div class="panel-body">*

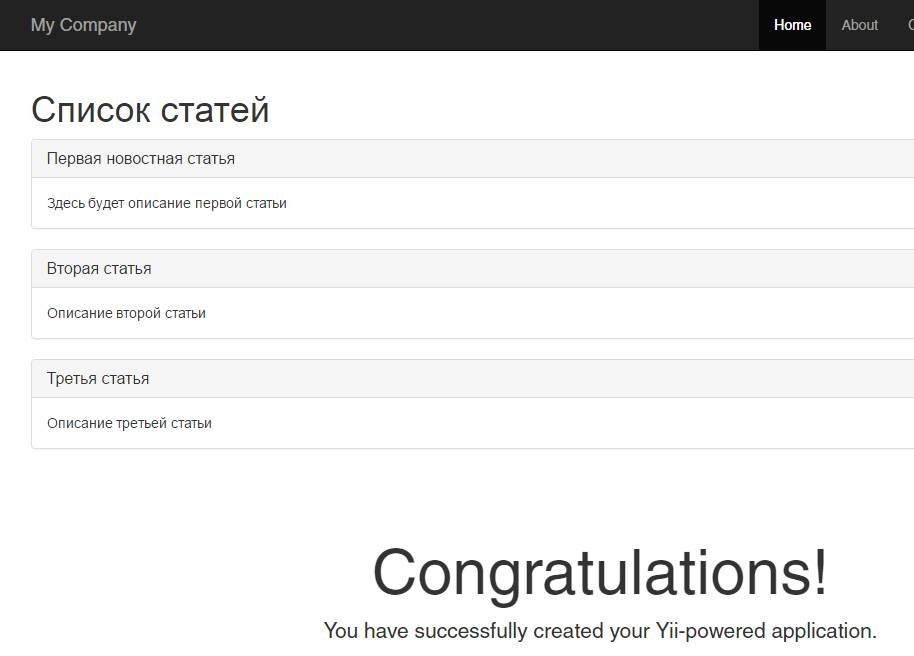
*<?= $post->excerpt ?>*

*</div>*

*</div>*

*<?php endforeach; ?>*

В результате на главной странице сайта появится новое представление:



Но каждая статья принадлежит к определенной категории. Давайте создадим такую таблицу и заполним ее тестовыми данными.

*CREATE TABLE IF NOT EXISTS `category` (*

*`id` int(10) unsigned NOT NULL AUTO\_INCREMENT,*

*`name` varchar(255) NOT NULL,*

*`keywords` varchar(255) DEFAULT NULL,*

*`description` varchar(255) DEFAULT NULL,*

*PRIMARY KEY (`id`)*

*) ENGINE=InnoDB DEFAULT CHARSET=utf8;*

Как вы помните, поле category\_id таблицы post ссылает на поле id таблицы category и указывает на принадлежность статьи к определенной категории. Так вот, как бы нам, доставая статьи, заодно достать и категории, к которым относятся данные статьи? Проще говоря, нужно выполнить что-то вроде объединения таблиц (JOIN).

Для решения данной задачи во фреймворке предусмотрены связи. Фактически это аналог тех самых связей SQL, которые мы знаем под названиями один-к-одному, один-ко-многим и т.д. Итак, для реализации связи нам потребуется создать для начала еще одну model, которая будет работать с категориями.

В папке Models создайте новый файл Category.php.

*namespace frontend\models;*

*use Yii;*

*class Category extends \yii\db\ActiveRecord*

*{*

*}*

Ну а теперь необходимо связать модели, используя метод **геттер**. В классе Post создадим следующий метод:

*class Post extends ActiveRecord*

*{*

*public function getCategory(){*

*return $this->hasOne(Category::className(), ['id' => 'category\_id']);*

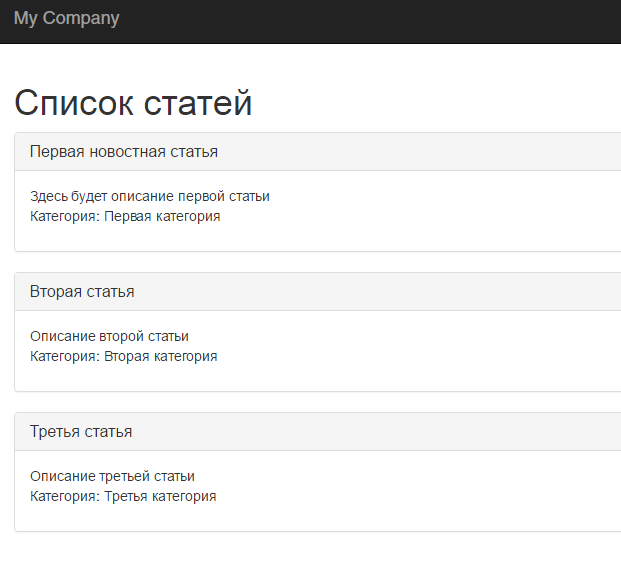
*}*

*}*

Давайте разберем этот метод. Начнем с названия – **getCategory**. Фактически это имя связи, которое скажет нам, что метод устанавливает связь с Category. Метод обращается к методу **hasOne**, который описывает связь один-к-одному. Первым параметром методу hasOne() мы должны передать имя класса модели, с которой необходимо установить связь. Делать это рекомендуется именно так, как мы прописали в коде НазваниеМодели::className(). Вторым параметром мы передаем массив, где ключом выступает поле таблицы, с которой мы связываемся, а значением – поле текущей таблицы. В нашем примере ключ id – это поле `category`.`id`, значение – `post`.`category\_id`.

Как теперь получить доступ к связанным данным? После определения связи, нам доступно виртуальное свойство с именем геттера связи. Поскольку геттер мы назвали getCategory, виртуальное свойство будет называться category. Почему виртуальное? Потому что в объектах массива $posts у нас сейчас не будет такого свойства. Тем не менее, если мы обратимся сейчас в цикле к этому свойству – мы получим связанные данные.

Осталось лишь обратиться к нужным свойствам данного объекта. Например, выведем наименование категории, которой принадлежит тот или иной пост.



Для того, чтобы достать коллекцию записей из таблицы category, мы использовали всего пару строк кода:

*$modelCategory = new Category();*

*$cats = $modelCategory::find()->all();*

В этих строках мы как раз и используем возможности класса Active Record. В общем случае для работы с указанным классом необходимо:

* получить объект модели и создать объект запроса, вызвав метод find;
* при необходимости настроить объект запроса, обращаясь к методам построения запроса (where, limit, orderBy и т.д.);
* вызвать необходимый метод получения данных, чаще всего это будут all, one или count.

В коде выше мы фактически пропустили этап настройки запроса и в результате запрос получился примитивным, вида SELECT \* FROM category. Давайте теперь попробуем совсем немного усложнить его, обратившись к нескольким методам настройки запросов.

Самый простой вариант – метод where. Попробуем вытащить категорию с id = 3:

*$modelCategory = new Category();*

*$cats = $modelCategory::find()->where('id=3')->all();*

*echo '<pre>';*

*print\_r($cats);*

*die;*

В итоге мы получим один объект в массиве с категорией, id которой равняется запрошенному.

В качестве параметра методу when мы передали строковое выражение. Однако, данный метод понимает также и формат массива. Например, тот же результат мы получим и с такой записью:

*$cats = $modelCategory::find()->where(['id' => 3])->all();*

Также мы можем передать массив операторов. Давайте для примера выполним

*$cats = $modelCategory::find()->where(['like', 'name', 's'])->all();*

Данный вариант записи сформирует запрос вида SELECT \* FROM category WHERE name LIKE ‘%s%’

Передавая различные параметры в данный метод, вы можете получить нужные вам данные по условию. Фактически данный метод является аналогом оператора WHERE в языке SQL.

Однако, несмотря на добавленное условие, мы все равно можем получать излишек данных, поскольку Yii генерирует запрос вида SELECT \*, а это не всегда хорошо. Например, у нас может быть большая таблица с десятками полей. Из всей этой таблицы нам всего-то нужно получить несколько нужных полей. Фреймворк же по умолчанию запросит все поля. Как указать, что мы хотим получить вполне конкретный набор полей?

Поможет метод select, который может принимать как строковый параметр, так и массив. В обоих случаях достаточно через запятую перечислить имена полей, который необходимы. Например, выберем поля id и name:

*$cats = Category::find()->select('id, name')->where(['like', 'name', 's'])->all();*

*$cats = Category::find()->select(['id', 'name'])->where(['like', 'name', 's'])->all();*

Оба варианта записи вернут один и тот же результат, а фреймворк в этот раз будет генерировать запрос вида:

*SELECT `id`, `name` FROM `category` WHERE `name` LIKE '%s%'*

На практике нам часто необходимо добавить в запрос несколько различных условий. При этом в различных ситуациях мы хотим чтобы выполнялось каждое из условий или хотя бы одно из перечисленных условий. В SQL это будут запросы вида:

*SELECT \* FROM table WHERE condition1 AND condition2...;*

*SELECT \* FROM table WHERE condition1 OR condition2...;*

Как реализовать такие запросы в Yii? Для этого можно использовать методы

*$cats = Category::find()->select(['id', 'name'])->where(['like', 'name', 's'])->andWhere('id<4')->all();*

В результате мы получим следующий запрос:

*SELECT `id`, `name` FROM `category` WHERE (`name` LIKE '%s%') AND (id<4)*

Аналогичным образом вы можете использовать метод orWhere, в этом случает будет добавлено альтернативное условие с помощью оператора OR.

Метод orderBy позволяет упорядочить выборку, сортируя ее по указанным полям. Например, давайте упорядочим выбираемые нами категории по названию:

*$cats = Category::find()->select(['id', 'name'])->where(['like', 'name', 's'])->andWhere('id<4')->orderBy('name ASC')->all();*

*SELECT `id`, `name` FROM `category` WHERE (`name` LIKE '%s%') AND (id<4) ORDER BY `name`*

Следующая пара полезных методов: limit и offset. Эти методы всегда принимают участие в постраничной навигации и отвечают они за получение определенной части данных. Метод limit указывает на число записей, которое необходимо запросить, а метод offset позволяет указать число, с которого необходимо начать выборку.

Давайте попробуем выбрать первые три записи (ниже необходимый код и SQL-

*$cats = Category::find()->select(['id', 'name'])->limit(3)->all();*

*SELECT `id`, `name` FROM `category` LIMIT 3;*

В результате мы получили первые три записи таблицы. Теперь попробуем

*$cats = Category::find()->select(['id', 'name'])->limit(3)->offset(1)->all();*

*SELECT `id`, `name` FROM `category` LIMIT 3 OFFSET 1*

Yii предлагает несколько методов, с помощью которых мы можем организовать различные варианты выборки. Собственно, с одним из методов мы постоянно и работали – это метод all, который мы вызывали в самом конце. Данный метод возвращает массив объектов. При этом мы получим все строки из запрашиваемой модели.

Однако далеко не всегда нам нужно получить все записи, часто нам нужно получить, к примеру, только одну запись. Это, конечно, можно сделать, использовав метод limit, однако фреймворк предлагает для этой задачи специальный метод one, который вернет первую строку запроса. Давайте попробуем, просто заменим метод all методом one:

*$cats = Categories::find()->one();*

Однако будьте аккуратны. Несмотря на то, что возвращается первая строка запроса, на самом деле выбираются все данные. Это можно легко проверить, заглянув в Yii Debugger.

Т.е. Yii не добавляет к запросу оператор LIMIT. Поэтому используйте метод one в тех случаях, когда заведомо предполагается выборка единственной записи, например, по первичному ключу. Во всех остальных случаях не забывайте добавлять метод limit:

*$cats = Categories::find()->limit(1)->one();*

Познакомимся с еще одним полезным методом – это метод asArray. Как вы заметили, мы всегда получаем данные в виде объектов. Однако, если получаемых записей много (ну хотя бы несколько сотен), тогда происходит достаточно большой расход памяти. В этом случае лучшим вариантом будет получить данные не в виде объектов, а в формате массивов. Метода asArray как раз и поможет справиться с этой задачей. Просто добавим его перед вызовом метода получения данных:

*$cats = Categories::find()->asArray()->all();*

***Метод afterSave***

Данный метод вызывается после метода save, который осуществляет сохранение данных в БД.

Вполне возможно вам может потребовать выполнить некоторые действия после операции сохранения.

В качестве примера возьмем форму редактирования записей контроллера Post. Добавим в основное меню новый пункт для вызова таблицы Роst.

Далее откроем модель Post и допишем в нее метод afterSave:

*public function afterSave($insert, $changedAttributes)*

*{*

*parent::afterSave($insert, $changedAttributes);*

*Yii::$app->session->setFlash('success', 'Запись сохранена');*

*}*

Мы хотим записать сообщение в сессию, чтобы затем вывести его в шаблоне.

Чтобы вывести сообщение, в шаблоне – views\layouts\main.php – добавим следующий код перед обращением к переменной $content:

*<?php if( Yii::$app->session->hasFlash('success') ): ?>*

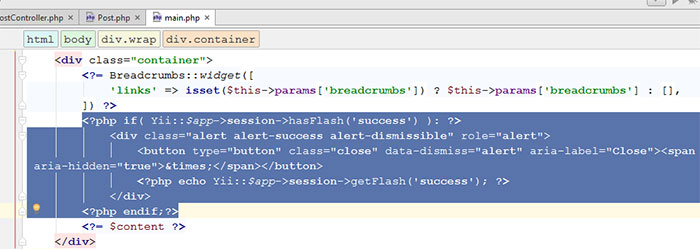
*<div class="alert alert-success alert-dismissible" role="alert">*

*<button type="button" class="close" data-dismiss="alert" aria-label="Close"><span aria-hidden="true">&times;</span></button>*

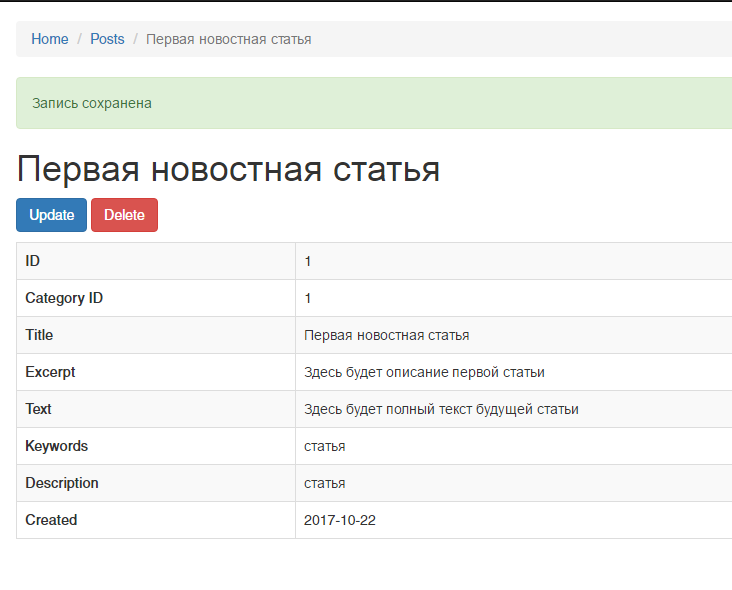
*<?php echo Yii::$app->session->getFlash('success'); ?>*

*</div>*

*<?php endif;?>*



Данный код будет покажет сообщение из сессии с ключом success, если таковое там будет. Давайте протестируем теперь процесс сохранения записи.



Сейчас одно и то же сообщение мы видим как в случае обновления записи, так и в случае ее создания. А что если мы захотим разграничивать их и показывать различные сообщения в зависимости от выполненной операции, то необходимо использовать первый параметр метода afterSave – $insert. Данный параметр принимает булево значение: true – если запись новая или false – если запись обновляется. Зная это, давайте немного изменим реализацию метода, описав условие для проверки новой записи (isNewRecord):

*public function afterSave($insert, $changedAttributes)*

*{*

*if ($insert) {*

*Yii::$app->session->setFlash('success', 'Запись добавлена');*

*} else {*

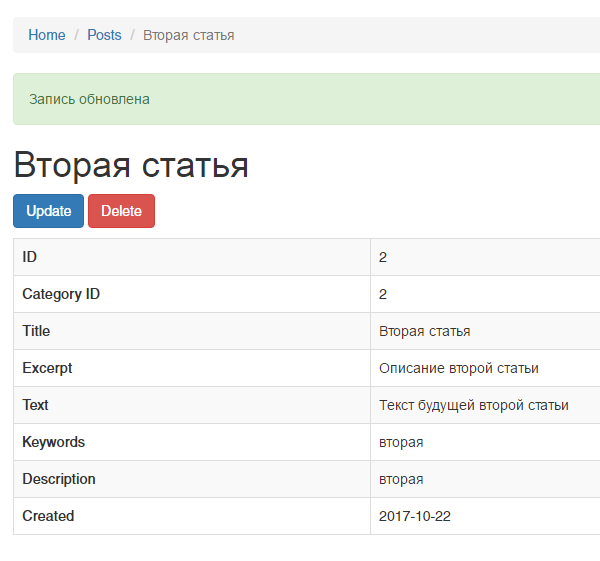
*Yii::$app->session->setFlash('success', 'Запись обновлена');*

*}*

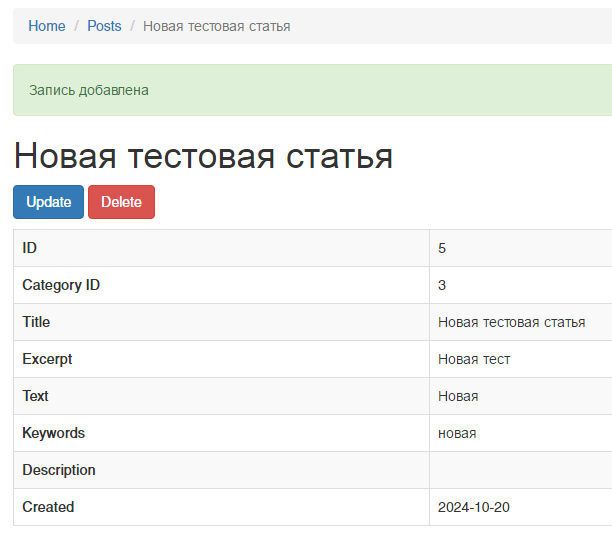
*parent::afterSave($insert, $changedAttributes);*

*}*

Теперь при обновлении записи мы будем видеть одно сообщение:



При добавлении нового поста мы увидим уже другое сообщение:



***Сkeditor***

Одним из популярных вариантов визуальных редакторов является CKEditor. Работать с ним может любой пользователь, знакомый с офисным редактором Word – ничего сложного и все интуитивно понятно.

На странице расширение мы можем найти необходимую команду для установки визуального редактора через Composer. Давайте используем ее:

*composer require --prefer-dist mihaildev/yii2-ckeditor "\*"*

Теперь необходимо вызвать widget. Применим данный виджет для редактирования поля Text в таблице Post. Для этого необходимо изменить форму для данной таблицы. Откройте файл frontend\views\Post\\_form.php В нем содержится форма для добавления/редактирования записей в нужной таблице.

Добавьте к Use:

*use mihaildev\ckeditor\CKEditor;*

Затем замените поле с текстом на следующий код

*<?= $form->field($model, 'text')->widget(CKEditor::className(),[*

*'editorOptions' => [*

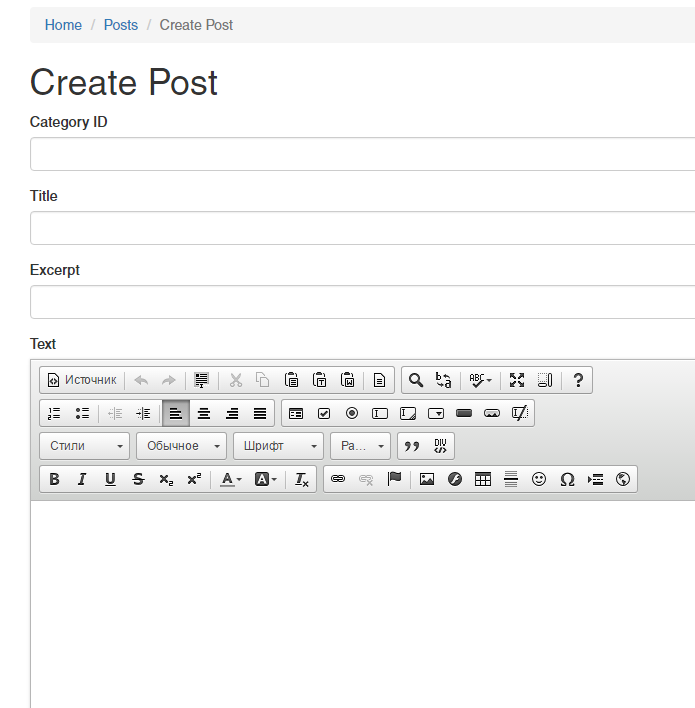
*'preset' => 'full', //разработанны стандартные настройки basic, standard, full данную возможность не обязательно использовать*

*'inline' => false, //по умолчанию false*

*],*

*]); ?>*

Если все было выполнено верно, для редактирования поля будет создан редактор.

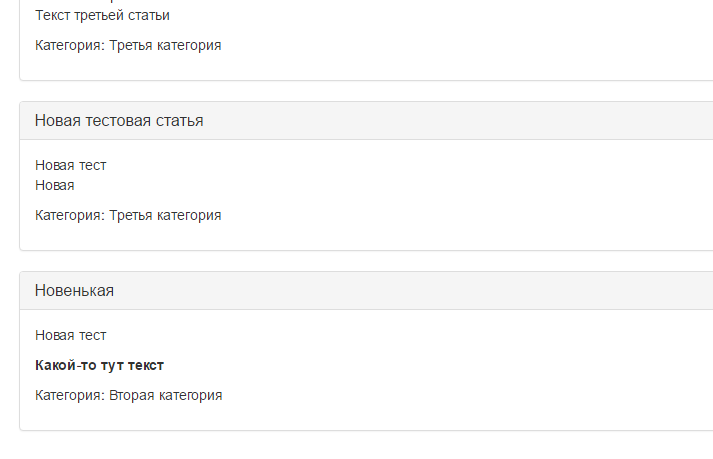


Добавим новую статью, выделив в ней слова жирным.

Далее добавим к представлению на главной странице поле text:

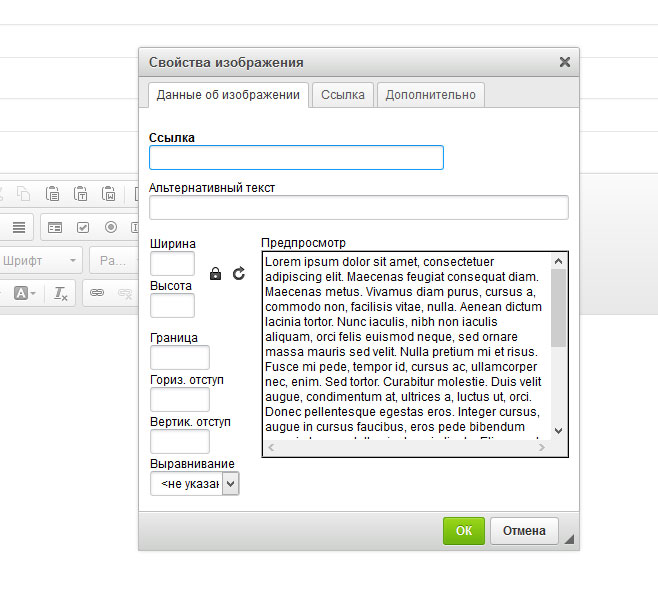
*<br>  
 <p><?= $post->text ?></p>*

И посмотрим результат. Добавленная статья будет иметь форматирование.



***Сkeditor с загрузкой изображений***

**Сkeditor** по умолчанию не позволяет загружать картинки, можно только вставлять ссылки на уже загруженные изображения.



Давайте исправим этот момент и установим файловый менеджер ELFinder. Итак, перейдем на страницу данного расширения на [GitHub](https://github.com/MihailDev/yii2-elfinder" \t "_blank), где можно ознакомиться с инструкцией по его установке и настройке. Для его установки нам потребуется выполнить в Composer следующую команду:

*composer require --prefer-dist mihaildev/yii2-elfinder "\*"*

Расширение установлено. Теперь подключим и настроим его согласно инструкции. Для начала откроем файл common\config\main.php и вставим в него следующий код:

*'controllerMap' => [*

*'elfinder' => [*

*'class' => 'mihaildev\elfinder\PathController',*

*'access' => ['?'],*

*'root' => [*

*'path' => 'common\files',*

*'name' => 'Files'*

*],  
 ]  
],*

Укажем папку, в которую будут сохраняться картинки (patch). Также укажем уровень доступа, заменив значение access с @ (доступ для авторизованных пользователей) на ? (доступ для гостей).

В папке common не забудьте создать папку files, которую мы указали в конфигурации path для сохранения картинок.

Теперь добавим предлагаемый код в представление. Поскольку мы используем ELFinder совместно с расширением mihaildev/yii2-ckeditor, используем соответствующий вариант из документации:



Вместо кода подключенного ранее редактора, используем теперь следующий код:

*use mihaildev\ckeditor\CKEditor;*

*use mihaildev\elfinder\ElFinder;*

*<?=*

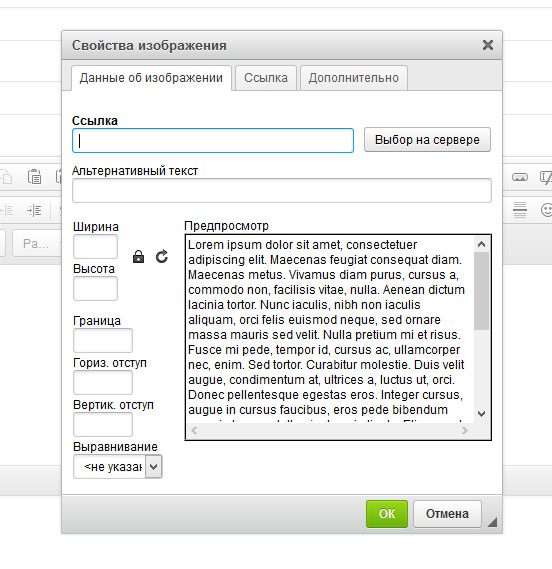
*$form->field($model, 'text')->widget(CKEditor::className(), [*

*'editorOptions' => ElFinder::ckeditorOptions('elfinder',[]),*

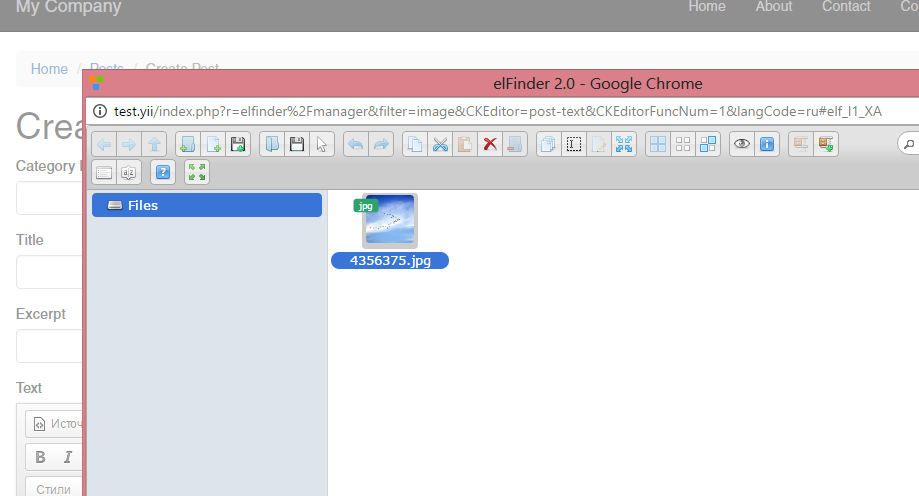
*]);*

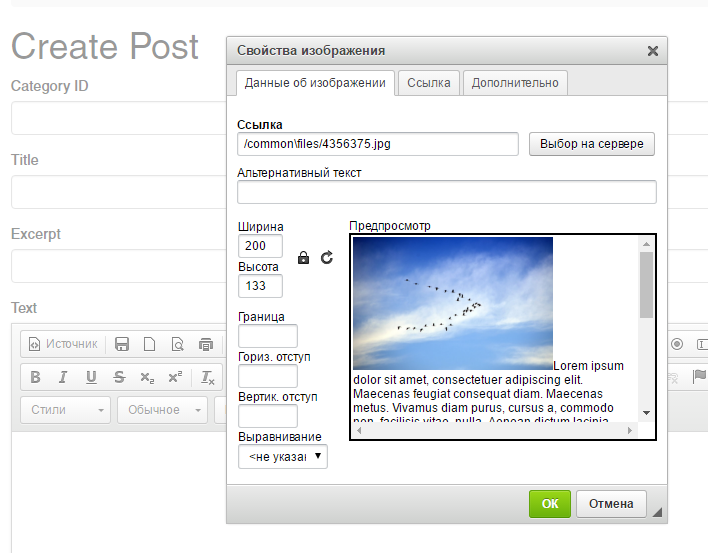
*?>*

Можем проверить работу файлового менеджера и загрузки картинок. Открыв иконку добавления картинки, мы там обнаружим новую кнопку Выбор на сервере:

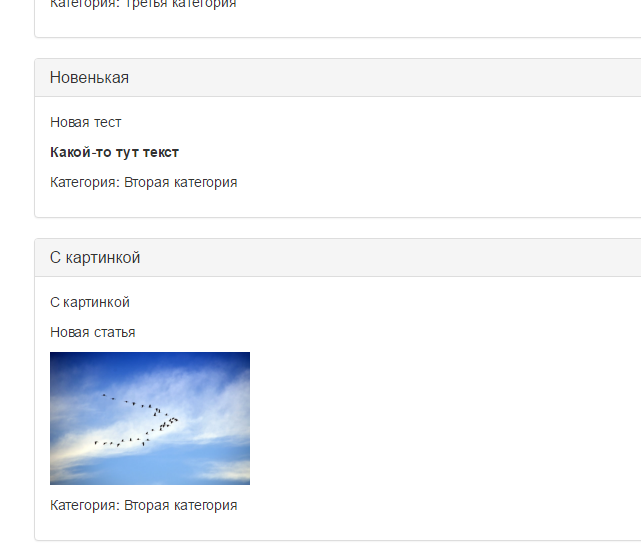


Клик по кнопке откроет окно загрузки изображений:





После добавления новой записи на главной странице появится новая запись с картинкой.



**Задание**

1. Создать таблицы Post и Category, связать их и настроить заполнение в соответствии с предложенными методами.
2. Подключить и привести пример использования Сkeditor с загрузкой изображений.
3. Добавить к заданной таблице индивидуального задания из лаб.работы № 1 две связанные таблицы.
4. Связать таблицы: в PhpMyAdmin и программно в Yii.
5. Для каждой из таблиц установить разные произвольные методы ActiveRecord (использовать метод aftersave и еще 2 на выбор, найденных и разобранных самостоятельно, документация - <https://yiiframework.com.ua/ru/doc/guide/2/db-active-record/> )
6. Реализовать три запроса на выборку из таблиц по разным условиям, обеспечить вывод результатов в любом удобном виде.