Episode 0 - What you will create & Introduction

Hello,
My name is Jaroslav Pulik and I'm proud to announce my new Yii 2 tutorial series - **Create your own Yii 2 powered blog**!
In this tutorial series we will create full featured Yii 2 powered blog (with frontend and backend) focused on SEO and future expandability. We will create also the administration area (backend) to manage our blog as comfortable as possible. My goal is to teach you all of the main Yii 2 principles. After graduating this course you will be able to fully understand how your blog works and you will have all of the knowledge how to extend and maintain your blog to perfectly fit your needs.

Happy coding! :-)

PS: If do you have any suggestions for the new episodes, which topics would you like to see in this tutorial, you can leave me message in the comments section below :-).

Premium episodes explanation

Episodes 01 - 05 are free, but all other are premium. I have decided to charge this tutorial series with a **small fee** for the several reasons. I'm spending **many hours** by creating these **high quality** tutorials. I have **no annoying ads** on this site. Also, by buying this tutorial series you will **support** this page and **help me** bring **another great episodes for you**. You will be able to **read all of the new episodes** which will release later (after your purchase) in this series. I will **update** individual episodes if any uncertainties or Yii core changes will occur. If you will have any questions or problems related to this tutorial series, I will provide **instant support and help for you**.

What you will learn

- Install and set up **Yii 2 advanced application template**
- Create and apply database **migrations**
- Setting up the **RBAC** (Role based access control) to your application
- Generate models and **CRUD** code (Create, Read, Update, Delete) with Gii
- Customize and enhance generated models

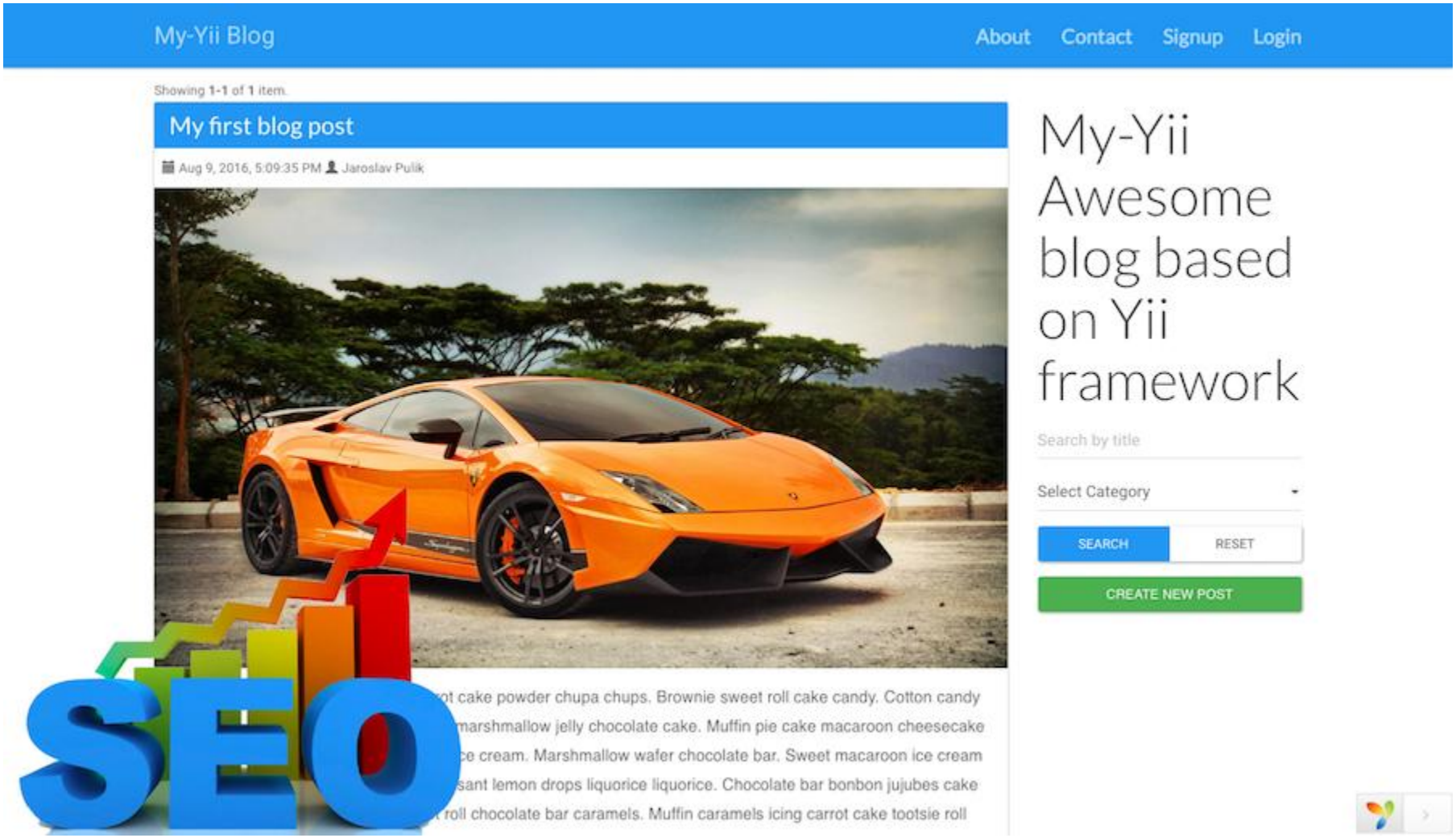
- Use Yii 2 **model behaviors** to automate common tasks
- Customize and enhance generated CRUD code
- **Upload files** and handle them with Yii
- Theme application fontend and backend with custom **templates**
- Use **AJAX** and **PJAX** in your Yii 2 application
- Tune up your application (**Caching**, **Eager loading**)
- How to **analyze**, **test** and **optimize** your blog for **search engines** (**SEO**)
- **Many another things will come in the upcoming episodes!**

List of the all episodes in this series

1. Installing and setting up your blog
2. Creating Post, Category, Tag, PostTag and RBAC tables
3. Setting up the Role Based Access Control for our blog
4. Generating Models with Gii
5. Generating CRUD code with Gii
6. Managing categories and tags from backend
7. Creating and updating Posts
8. Customizing Posts index page
9. Installing new frontend theme
10. Customizing Posts view page
11. Installing AdminLTE template for backend
12. Managing Users from blog's backend
13. Managing Users from blog's backend Part II.

14. Securing the blog's backend
15. Creating new Posts with AJAX
16. Securing frontend and fixing our blog
17. Search Engine Optimization
18. Search Engine Optimization II
19. ... 20. more episodes are coming soon

Frontend preview (not finished yet, 19/20 episodes):



Backend preview (not finished yet, 19/20 episodes):

Jaroslav Pulik
Online

Menu Yii2

- Gii
- Debug
- Tags
- Categories
- Users

Users

Home > Users

Users

Showing **1-5** of **5** items.

#	Username	Email	User Role	Status	Created At	Updated At	
1	Jaroslav Pulik	jaroslav.pulik@gmail.com	admin	Active	Sep 13, 2015 1:08:39 PM	Sep 13, 2015 1:08:39 PM	
2	Janko Hrasko	janko.hrasko@gmail.com	author	Active	Oct 6, 2015 11:29:52 PM	Oct 9, 2015 1:46:35 PM	
3	John Doe	john.doe@example.com	user	Active	Oct 9, 2015 2:24:32 PM	Oct 9, 2015 2:24:32 PM	
4	Alex Mike	a.m@example.com	user	Active	Oct 9, 2015 2:24:53 PM	Oct 9, 2015 2:24:53 PM	
5	Sam Sim	sam.sim@example.com	user	Active	Oct 9, 2015 2:25:21 PM	Oct 9, 2015 2:25:21 PM	

Episode 1 - Installing and setting up your blog

Hello,
In this tutorial we will create our own SEO friendly blog based on Yii v2 framework. For more information please check Episode 0 - What you will create.

Prereq

This tutorial series gives you the step by step instructions on how to create awesome Yii applications. You will learn how to work with Yii 2 framework by examples. It is really simple, but I assume that you have at least some experience with OOP PHP, MySQL, HTML and JavaScript already. If you will have any questions or if you will need deeper explanation, please leave your questions in the comments section below this article.

Installation

So let's start, first we need to install and setup our Yii application project. For this project we will use the advanced application template. If you need more information what is Yii Advanced application template and how to install Composer, please visit our another episode: Yii2 Essentials - Yii2 Installation.
If you have already installed Composer and you are familiar with Yii application templates, please continue reading here. Now you need to open console and **navigate to the root directory of your web server**, then run these commands in console:

```
composer global require "fxp/composer-asset-plugin:~1.1.1"
```

and

```
composer create-project yiisoft/yii2-app-advanced blog 2.0.8
```

We must initialize our application after installation. Advanced application template have two default environments: `dev` and `prod`. First is for development. It has all the developer tools and debug turned on. Second is for production server deployments. It has debug and developer tools turned off.
Type and run these commands:

```
cd blog
```

and

```
php init
```

To choose `dev` environment type “0” and press RETURN. After this you must confirm your selection. To confirm type “yes” and than press RETURN again.

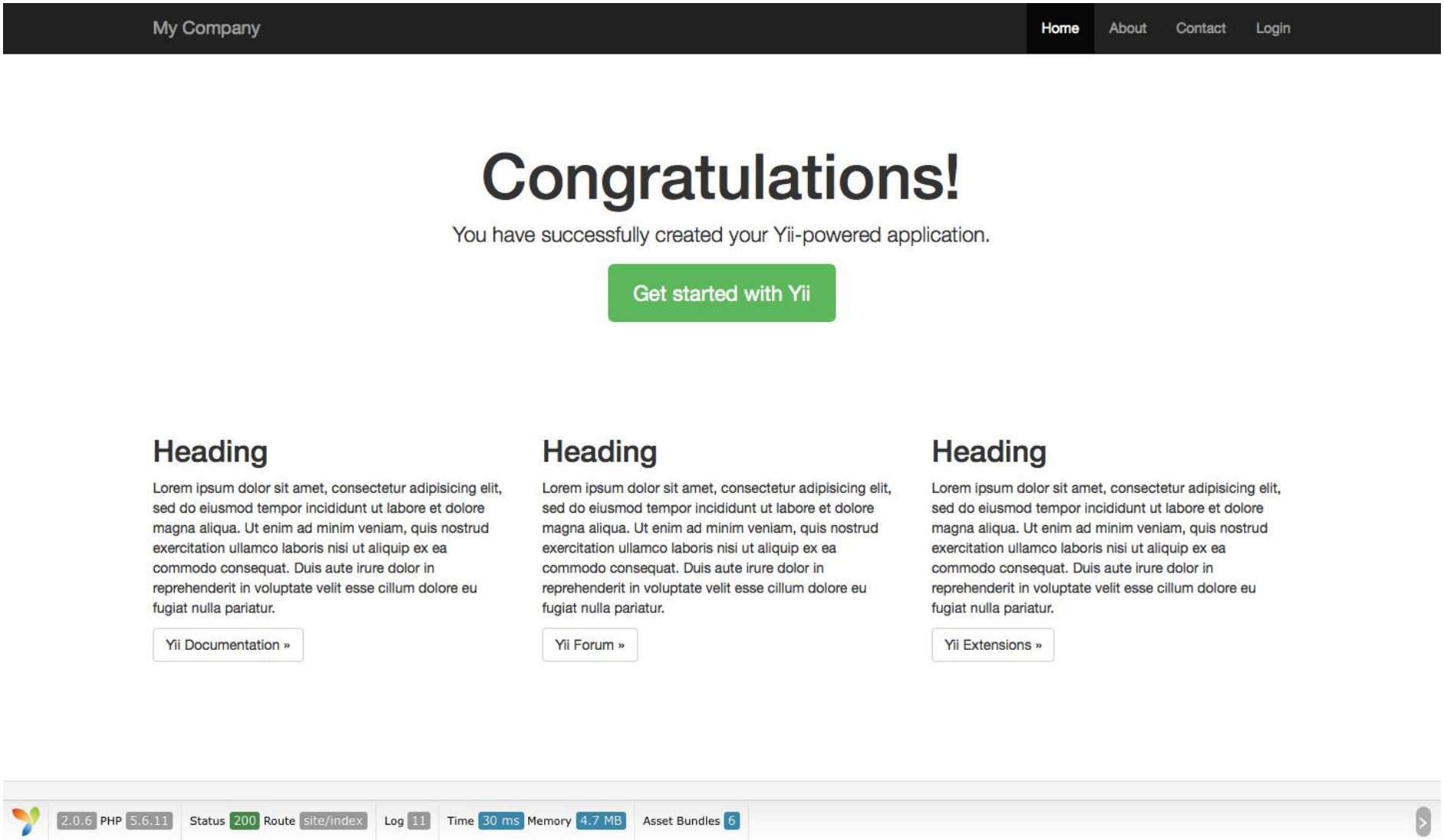
You should see something like this:

```
Yii Application Initialization Tool v1.0

Which environment do you want the application to be initialized in?

  [0] Development
  [1] Production

  Your choice [0-1, or "q" to quit] 0

  Initialize the application under 'Development' environment? [yes|no] yes

  Start initialization ...

   generate backend/config/main-local.php
   generate backend/config/params-local.php
   generate backend/web/index-test.php
   generate backend/web/index.php
   generate common/config/main-local.php
   generate common/config/params-local.php
   generate console/config/main-local.php
   generate console/config/params-local.php
   generate frontend/config/main-local.php
   generate frontend/config/params-local.php
   generate frontend/web/index-test.php
   generate frontend/web/index.php
   generate yii
   generate cookie validation key in backend/config/main-local.php
   generate cookie validation key in frontend/config/main-local.php
       chmod 0777 backend/runtime
       chmod 0777 backend/web/assets
       chmod 0777 frontend/runtime
       chmod 0777 frontend/web/assets
       chmod 0755 yii
       chmod 0755 tests/codeception/bin/yii

  ... initialization completed.
```

Now you should be able to go to

```
http://localhost/blog/frontend/web
```

And you should see this:



Setting basic things

.htaccess

We do not want to point to `/frontend/web` folder manually every time. To do this automatically we must create 3 new .htaccess files. First we will create in our application root directory (`/.htaccess`) with this content:

```
<IfModule mod_autoindex.c>

    Options -Indexes
</IfModule>



<IfModule mod_rewrite.c>

    Options +FollowSymlinks

    RewriteEngine On



    RewriteCond %{REQUEST_URI} ^/backend
    RewriteRule ^backend/(.*)$ backend/web/$1 [L]



    RewriteCond %{REQUEST_URI} !^public
    RewriteRule ^(.*)$ frontend/web/$1 [L]



    RewriteCond %{HTTP_USER_AGENT} libwww-perl.*
    RewriteRule .* – [F,L]
</IfModule>



<Files ~ "(.json|.lock|.git)">
    Order allow,deny
    Deny from all
</Files>
```

```
RewriteRule (^\.|/\.) - [F]
```

Second file we will create in the `/frontend/web` directory (`/frontend/web/.htaccess`) and third in the `/backend/web` directory (`/backend/web/.htaccess`) with this (the same) content:

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on



RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d



RewriteRule . index.php
```

Now we should be able to go to this URL:

```
http:
```

and see Yii welcome page.

Pretty URLs

Front end

Below is sample of the current application URL address pointing to `actionIndex()` method of the `SiteController` class:

```
http://localhost/blog/frontend/web/index.php?r=site%2Findex
```

We do not want to have so long and ugly URLs. Also for the good position in search engines like Google (SEO) are pretty URLs necessary. To setup pretty URLs in advanced application template we need to edit `frontend` or `backend` config files. Now we only need to setup frontend routing. So, lets do it. Open your `/frontend/config/main.php` configuration file and add this code to components array:

```
'components' => [




  'urlManager' => [
        'enablePrettyUrl' => true,
        'showScriptName' => false,
        'rules' => [
        ],
  ],




],
```

After saving this configuration file and refreshing our blog, we will see pretty URLs. The URL address bellow points to same controller and action as URL before, but is much more better readable.

```
http://localhost/blog/frontend/web/site/index
```

Back end

We do not need to set up pretty URLs for the `backend` application now. The reason is that only `frontend` will be accessible for standard users and for search engines. But if do you want to set up pretty URLs also for `backend`, just edit the `/backend/config/main.php` configuration file the same way as the `frontend` configuration file.

Application name

The name property specifies the application name that may be displayed to end users. We can set up this property in our `common/config/main.php` configuration file. Add this line of code after the `vendorPath` property:

```
'name' => 'My-Yii Blog',
```

Now we can get the application name with this code:

```
Yii::$app->name
```

Let's use it in our front end main layout file (`frontend/views/layouts/main.php`). Set the `brandLabel` property in `NavBar` widget like this:

```
'brandLabel' => Yii::$app->name,
```

and in our footer container:

```
<p class="pull-left">&copy; <?= Yii::$app->name ?> <?= date('Y') ?></p>
```

It is not necessary to configure this property if none of our code is using it, but I decided to configure it to help you understand Yii config files and theirs properties.

Database connection

Before setting up the database connection we have to create new database. So, create new MySQL database with name `blog`. Leave this database empty for now. We will create tables later.

Before proceeding, make sure you have installed both the PDO PHP extension and the PDO driver for the database you are using (e.g. pdo_mysql for MySQL). This is a basic requirement if your application uses a relational database. [1]

Updated 25.5.2016:

Instead of editing `common/config/main-local.php` file we will edit `environments/dev/common/config/main-local.php` configuration file [2]. It is better to do it this way because we will use these credentials only in local development. For production you will need to use another credentials.

Now we need to connect our blog to the MySQL database. To do this, we will edit the `environments/dev/common/config/main-local.php` file. Change the parameters to be correct for your database and MySQL server:

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

After this, run

```
php init
```

command again, choose 0, type *yes* and *All* to overwrite old config files and then press RETURN to confirm.

Testing connection

We will test our database connection by running Yii migration command trough console:

```
php yii migrate
```

Yii Migration Tool will ask you: Apply the above migration? Type “yes” and press RETURN. You should see:

```
AiruzivJaroslav:blog jpulik$ php yii migrate
Yii Migration Tool (based on Yii v2.0.6)

Creating migration history table "migration"...Done.
Total 1 new migration to be applied:
        m130524_201442_init

Apply the above migration? (yes|no) [no]:yes
*** applying m130524_201442_init
    > create table {{%user}} ... done (time: 0.042s)
*** applied m130524_201442_init (time: 0.098s)


Migrated up successfully.
```

If your migration was done successfully, you have just created User table in your `blog` database.

Congratulations! Your blog is now ready to Sign up and Log in users.

We will continue building our blog in next episode. If do you have any questions regarding to this episode, please write them below to the comments section.

Download files from this episode: episode_01.zip.

Episode 2 - Creating Post, Category, Tag, PostTag and RBAC tables

Hello,
In the previous episode, we have successfully installed and set up our Yii application. At the end we also run our first migration to test database connection and create `User` table. In this episode, we will learn more about migrations, create our own migrations and run RBAC migrations. Also, we will prepare all other tables which we will need to create our Yii blog application.

Database migrations

During the course of developing and maintaining a database-driven application, the structure of the database being used evolves just like the source code does. For example, during the development of an application, a new table may be found necessary; after the application is deployed to production, it may be discovered that an index should be created to improve the query performance; and so on. Because a database structure change often requires some source code changes, Yii supports the so-called database migration feature that allows you to keep track of database changes in terms of database migrations which are version-controlled together with the source code. [1]

Creating migrations

Tag

First, we will create new migration for `Tag` table. To generate new migration we have to run this command:

```
php yii migrate/create create_tag_table
```

You should see:

```
AiruzivJaroslav:blog jpulik$ php yii migrate/create create_tag_table
Yii Migration Tool (based on Yii v2.0.6)

Create new migration '/Users/jpulik/Sites/blog/console/migrations/m150904_094837_create_tag_table.php'? (yes|no) [no]:yes
New migration created successfully.
```

This will generate the new create_tag_table migration file in `console/migrations/` directory with this name:

`m<YYMMDD_HHMMSS>_create_tag_table` [2]

```
<?php
```

```
use yii\db\Migration;



class m160525_190407_create_tag_table extends Migration
{

    public function up()
    {
        $this->createTable('tag_table', [
            'id' => $this->primaryKey(),
        ]);
    }



    public function down()
    {
        $this->dropTable('tag_table');
    }
}
```

There are also `safeUp()` and `safeDown()` methods, but we won't use them because we don't actually need a transaction migration.

As the official documentation describes: [3]

In the migration class, you are expected to write code in the up() method that makes changes to the database structure. You may also want to write code in the down() method to revert the changes made by up(). The up() method is invoked when you upgrade the database with this migration, while the down() method is invoked when you downgrade the database.

We will use the new migration syntax introduced in Yii 2.0.6. For our tag table we need only few columns like id, name, created_at and updated_at. Our migration file will look like:

```
<?php
```

```
use yii\db\Migration;

class m150904_094837_create_tag_table extends Migration
{
    public function up()
    {
        $this->createTable('tag', [
            'id' => $this->primaryKey(),
            'name' => $this->string(64)->notNull()->unique(),
            'created_at' => $this->datetime()->notNull(),
            'updated_at' => $this->datetime(),
        ]);
    }

    public function down()
    {
        $this->dropTable('tag');
    }
}
```

Category

Run

```
php yii migrate/create create_category_table
```

to generate new migration for Category table.

Our `Category` migration will have this content:

```
<?php

use yii\db\Migration;
```

```
class m150904_102410_create_category_table extends Migration
{
    public function up()
    {
        $this->createTable('category', [
            'id' => $this->primaryKey(),
            'name' => $this->string(64)->notNull()->unique(),
            'slug' => $this->string(64)->notNull()->unique(),
            'meta_description' => $this->string(160),
            'created_at' => $this->datetime()->notNull(),
            'updated_at' => $this->datetime(),
        ]);
    }

    public function down()
    {
        $this->dropTable('category');
    }
}
```

Post

Run

```
php yii migrate/create create_post_table
```

to generate new migration for Post table.

Our `Post` migration will have this content:

```
<?php

use yii\db\Migration;
```

```
class m150904_102648_create_post_table extends Migration
{
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(128)->notNull()->unique(),
            'slug' => $this->string(128)->notNull()->unique(),
            'lead_photo' => $this->string(128),
            'lead_text' => $this->text(),
            'content' => $this->text()->notNull(),
            'meta_description' => $this->string(160),
            'created_at' => $this->datetime()->notNull(),
            'updated_at' => $this->datetime(),
            'created_by' => $this->integer()->notNull(),
            'updated_by' => $this->integer(),
            'category_id' => $this->integer()->notNull()
        ]);

        $this->createIndex('post_index', 'post', ['created_by', 'updated_by']);
        $this->addForeignKey('fk_post_category', 'post', 'category_id', 'category', 'id', 'CASCADE', 'CASCADE');
        $this->addForeignKey('fk_post_user_created_by', 'post', 'created_by', 'user', 'id', 'CASCADE', 'CASCADE');
        $this->addForeignKey('fk_post_user_updated_by', 'post', 'updated_by', 'user', 'id', 'CASCADE', 'CASCADE');
    }

    public function down()
    {
        $this->dropForeignKey('fk_post_category', 'post');
        $this->dropForeignKey('fk_post_user_created_by', 'post');
        $this->dropForeignKey('fk_post_user_updated_by', 'post');
        $this->dropTable('post');
```

```
    }
}
```

PostTag

Run

```
php yii migrate/create create_post_tag_table
```

to generate new migration for PostTag table.

Our `PostTag` migration will have this content:

```
<?php

use yii\db\Migration;

class m150906_141330_create_post_tag_table extends Migration
{
    public function up()
    {
        $this->createTable('post_tag', [
            'id' => $this->primaryKey(),
            'post_id' => $this->integer()->notNull(),
            'tag_id' => $this->integer()->notNull()
        ]);

        $this->createIndex('post_tag_index', 'post_tag', ['post_id', 'tag_id']);
        $this->addForeignKey('fk_post_tag_post', 'post_tag', 'post_id', 'post', 'id', 'CASCADE', 'CASCADE');
        $this->addForeignKey('fk_post_tag_tag', 'post_tag', 'tag_id', 'tag', 'id', 'CASCADE', 'CASCADE');
    }

    public function down()
    {
```

```
        $this->dropForeignKey('fk_post_tag_post', 'post_tag');
        $this->dropForeignKey('fk_post_tag_tag', 'post_tag');
        $this->dropTable('post_tag');
    }
}
```

Applying migrations

Run command

```
php yii migrate/up
```

in your console to apply newly created migrations.

This command will list all migrations that have not been applied so far. If you confirm that you want to apply these migrations, it will run the up() or safeUp() method in every new migration class, one after another, in the order of their timestamp values. If any of the migrations fails, the command will quit without applying the rest of the migrations.

RBAC (Role-Based Access Control) tables migration [4]

Before executing RBAC tables migration we must setup Yii `authManager` component. Yii provides two types of authorization managers: `yii\rbac\PhpManager` and `yii\rbac\DbManager`. The former uses a PHP script file to store authorization data, while the latter stores authorization data in a database. In our blog application we will store our RBAC data in database - so we will use DbManager. To configure authManager to work with DbManager we must add this code to `common/config/main.php`:

```
'components' => [

    'authManager' => [
        'class' => 'yii\rbac\DbManager',
    ],

],
```

After setting up the `authManager` we can now run RBAC migration:

```
php yii migrate --migrationPath=@yii/rbac/migrations
```

This migration will create following tables:

1. **itemTable**: the table for storing authorization items. Defaults to `auth_item`.
2. **itemChildTable**: the table for storing authorization item hierarchy. Defaults to `auth_item_child`.
3. **assignmentTable**: the table for storing authorization item assignments. Defaults to `auth_assignment`.
4. **ruleTable**: the table for storing rules. Defaults to `auth_rule`.

If your migration was successful you should see:

```
AiruzivJaroslav:blog jpulik$ php yii migrate --migrationPath=@yii/rbac/migrations
Yii Migration Tool (based on Yii v2.0.6)

Total 1 new migration to be applied:
        m140506_102106_rbac_init

Apply the above migration? (yes|no) [no]:y
*** applying m140506_102106_rbac_init
    > create table {{%auth_rule}} ... done (time: 0.078s)
    > create table {{%auth_item}} ... done (time: 0.024s)
    > create index idx-auth_item-type on {{%auth_item}} (type) ... done (time: 0.025s)
    > create table {{%auth_item_child}} ... done (time: 0.034s)
    > create table {{%auth_assignment}} ... done (time: 0.021s)
*** applied m140506_102106_rbac_init (time: 0.209s)


Migrated up successfully.
```

The `authManager` can now be accessed via `Yii::$app->authManager`. We will set up our roles, permissions and rules in the next episode focused specially to RBAC. We have only created necessary RBAC tables with built in migrations for now.

Now check your database, you should have these 10 tables: `auth_assignment`, `auth_item`, `auth_item_child`, `auth_rule`, `category`, `migration`, `post`, `post_tag`, `tag` and `user`.

Perfect, we are done for this episode! We have successfully learned how to create an apply database migrations. We have also run migration for Yii `authManager` component. We will use `authManager` to provide Role-Baces Access Control in the next episode.

We will continue building our blog in next episode. If do you have any questions regarding to this episode, please write them below to the comments section.

Download files from this episode: episode_02.zip.

Episode 3 - Setting up the Role Based Access Control for our blog

Hello,
We have already set up our authManager component in previous episode. So now we can start creating roles, permissions and theirs mutual connections.

We want to have 3 roles: user, author and admin. The “user” role is the default User role after registration. “Author” is the privileged User role which can add new Posts or edit own Posts. And “admin” is the User role which can do everything what “author” can but also can update ALL Posts.

Roles and Permissions

To create roles, permissions and theirs mutual connections we need to create new `console/controllers/RbacController.php` class. With this class we well later initialize our RBAC rules. Create new RbacController.php file with this content:

```
<?php
namespace console\controllers;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;




        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'User can create a post';
```

```
$auth->add($createPost);


$updatePost = $auth->createPermission('updatePost');
$updatePost->description = 'User can update post';
$auth->add($updatePost);




$user = $auth->createRole('user');
$auth->add($user);


$author = $auth->createRole('author');
$auth->add($author);


$admin = $auth->createRole('admin');
$auth->add($admin);




$auth->addChild($author, $createPost);


$auth->addChild($admin, $author);


$auth->addChild($admin, $updatePost);
```

```
    }
}
```

If do you want more informations about RBAC Configuration, I recommend you to take a look to official Yii guide [1].

Now we can initialize our RBAC configuration by running this command:

```
php yii rbac/init
```

After this, we should check our database if `auth_item` and `auth_item_child` tables are filled with rules. You should see (`auth_item` table):

←T→			name	type	description	rule_name	data	created_at	updated_at
Edit	Copy	Delete	admin	1	NULL	NULL	NULL	1441374017	1441374017
Edit	Copy	Delete	author	1	NULL	NULL	NULL	1441374017	1441374017
Edit	Copy	Delete	createPost	2	User can create a post	NULL	NULL	1441374017	1441374017
Edit	Copy	Delete	updatePost	2	User can update post	NULL	NULL	1441374017	1441374017
Edit	Copy	Delete	user	1	NULL	NULL	NULL	1441374017	1441374017

Also, we need to automatically assign “user” role to every new User whose registered to our blog. To do this, we need to update `frontend\models\SignupForms` action `signup()`. We just need to add 3 new lines:

```
$auth = \Yii::$app->authManager;
$userRole = $auth->getRole('user');
$auth->assign($userRole, $user->getId());
```

Entire signup() method now should looks like:

```
public function signup()
{
        if (!$this->validate()) {
            return null;
        }

        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
```

```
        $user->setPassword($this->password);
        $user->generateAuthKey();

        if ($user->save()) {



            $auth = \Yii::$app->authManager;
            $userRole = $auth->getRole('user');
            $auth->assign($userRole, $user->getId());

            return $user;
        }

        return null;
}
```

Rules

Rules add additional constraint to roles and permissions. A rule is a class extending from `yii\rbac\Rule`. It must implement the `execute()` method. In the hierarchy we've created previously author cannot edit his own post. Let's fix it. First we need a rule to verify that the user is the post author. To do this, we need to create `console/rbac/AuthorRule.php` file. Also, you will need to create the rbac folder in the console directory. AuthorRule.php should contain:

```
<?php

namespace console\rbac;

use yii\rbac\Rule;



class AuthorRule extends Rule
{
    public $name = 'isAuthor';
```

```
    public function execute($user, $item, $params)
    {
        return isset($params['model']) ? $params['model']->createdBy->id == $user : false;
    }
}
```

The rule above checks if the post is created by $user. We'll create new permission `updateOwnPost` and associate the new rule with it. To do this, we will create new `actionCreateAuthorRule()` method in our RbacController class (`console\controllers\RbacController.php`).

```
public function actionCreateAuthorRule()
{
    $auth = Yii::$app->authManager;



    $rule = new \console\rbac\AuthorRule();
    $auth->add($rule);



    $updateOwnPost = $auth->createPermission('updateOwnPost');
    $updateOwnPost->description = 'Update own post';
    $updateOwnPost->ruleName = $rule->name;
    $auth->add($updateOwnPost);



    $updatePost = $auth->getPermission('updatePost');



    $author = $auth->getRole('author');
```

```
    $auth->addChild($updateOwnPost, $updatePost);



    $auth->addChild($author, $updateOwnPost);
}
```

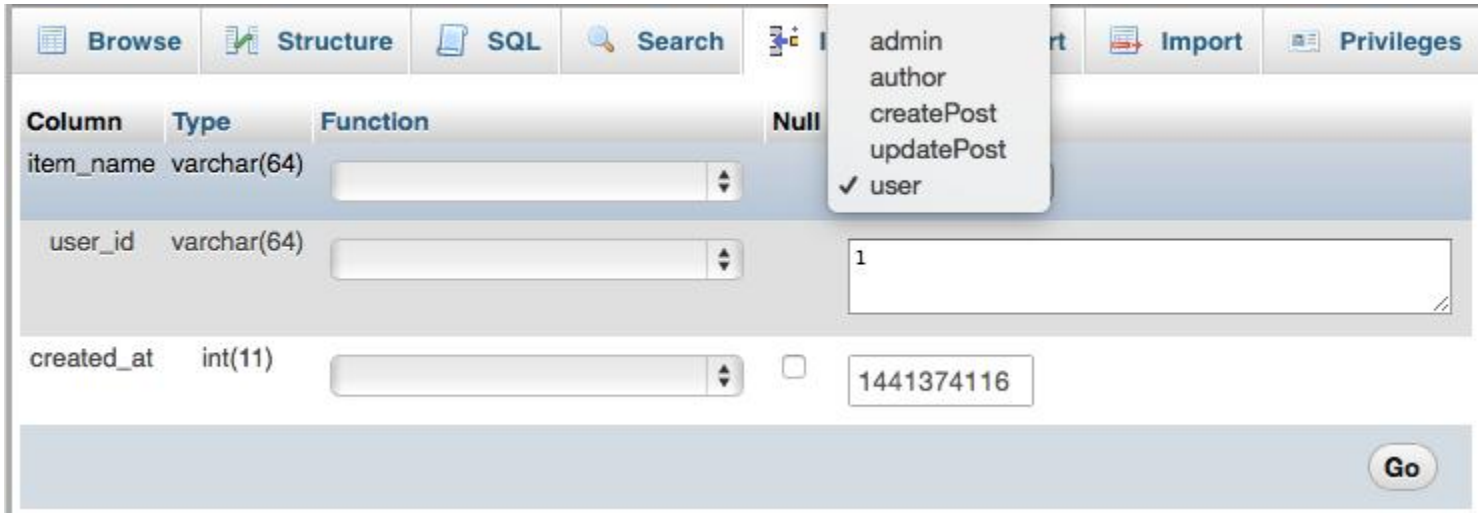
Then run:

```
php yii rbac/create-author-rule
```

Now we are done!

Trying it out

Now is the good time to sign up on your blog. After registration you should be automatically logged in and you should have assigned the “user” role in the `auth_assignment` table. Manually change your role to “admin”. You can do it by changing value “user” to “admin” in `item_name` column.



Access Check

To check if user is able to create new Post:

```
if (\Yii::$app->user->can('createPost')) {
```

```
}
```

To check if a user can update a post, we need to pass an extra parameter that is required by `AuthorRule` described before:

```
if (\Yii::$app->user->can('updatePost', ['model' => $post])) {


}
```

We will continue building our blog in next episode. If do you have any questions regarding to this episode, please write them below to the comments section.

Download files from this episode: episode_03.zip.

Episode 4 - Generating Models with Gii

Hello,
In the previous episode, we have successfully created all the tables which we will need for our blog. Now we need to generate models which will represent these tables in our application.

What is Gii?

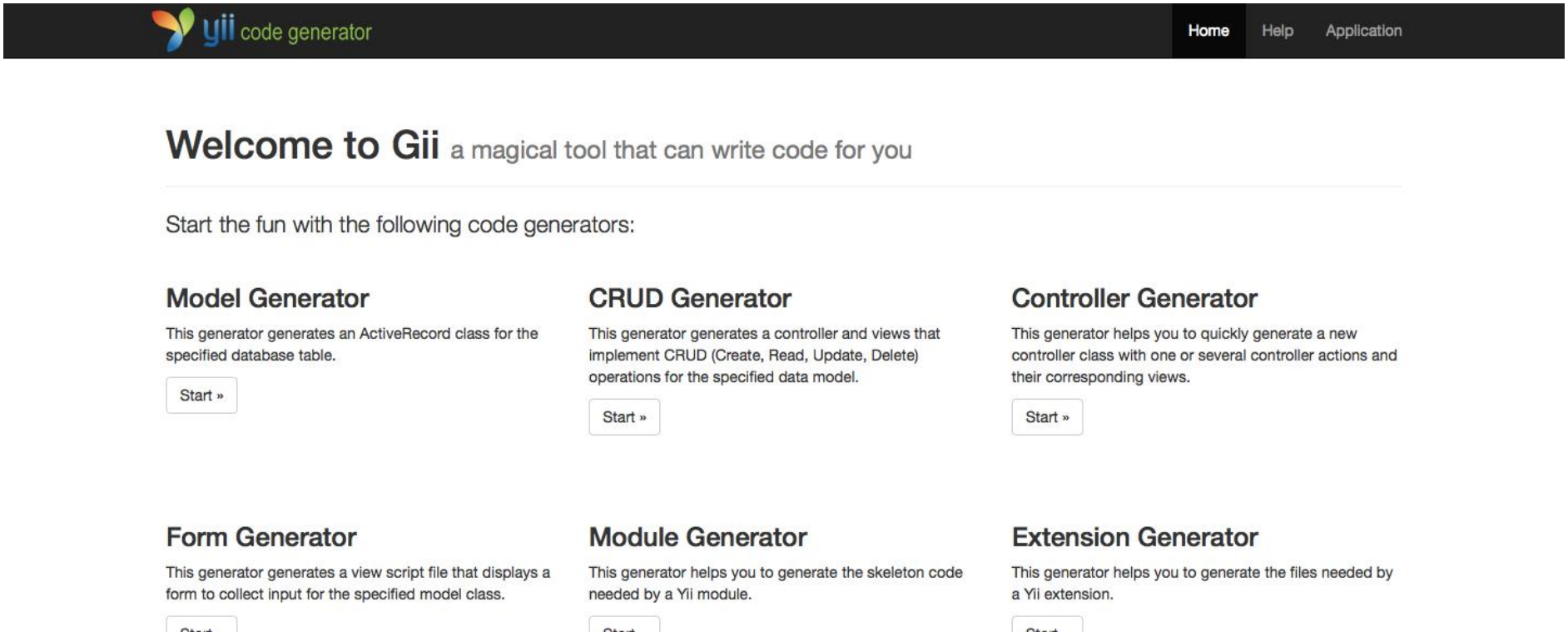
Gii is web-based code generator. With Gii we can quickly generate models, forms, modules, CRUD, etc.. Gii also provides command line interface for people who prefer to work with their console.

How to use?

In our environment we can use Gii web-based interface by going to this address:

```
http://localhost/blog/frontend/web/gii
```

You should see:



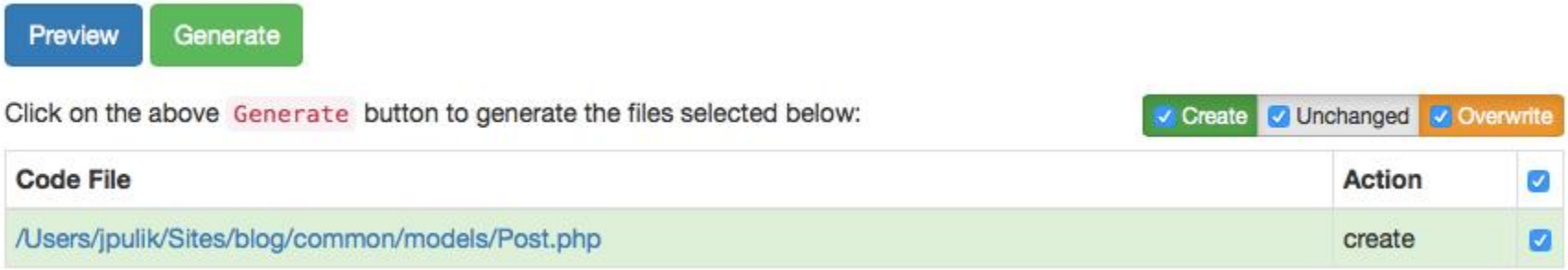
Generating Models

First, we need to generate Models for all tables we have created. To generate model click on the “Start” button under the Model Generator section. Model Generator generates an ActiveRecord class for the specified database table.

Post Model

To generate Post Model fill the Model Generator form with following:
- Table name: `post`
- Model class: `Post`
- Namespace: change `app\models` to `common\models`
- Base class: leave the default value
- Database connection ID: leave the default value

Make sure that “Generate Relations” option is set to `All relations` and click on the “Preview” button. After clicking you should see:



Make sure that action “create” is checked and click on the “Generate” button. You should see:

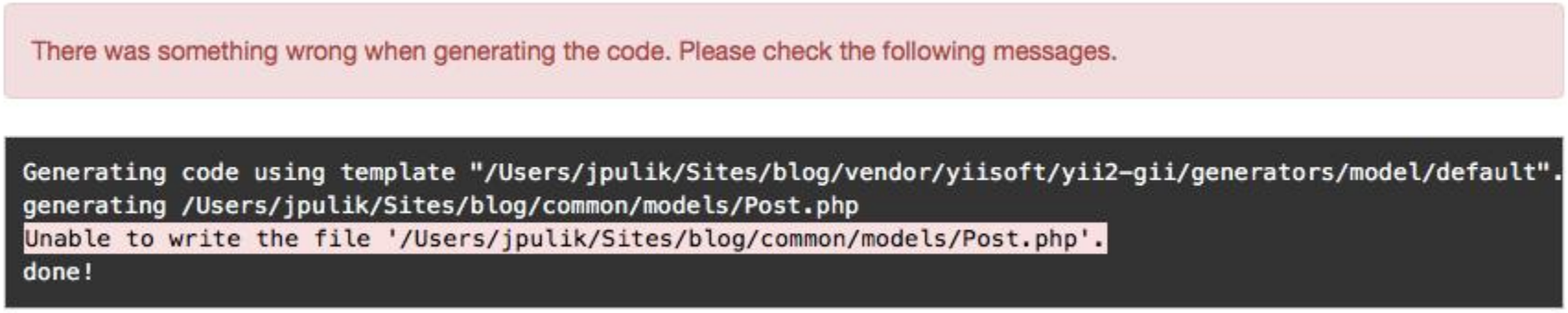
```
The code has been generated successfully.
```

and something similar to this:

```
Generating code using template "/Users/jpulik/Sites/blog/vendor/yiisoft/yii2-gii/generators/model/default"...
```

```
generated /Users/jpulik/Sites/blog/common/models/Post.php
done!
```

Tip: If do you see error similar to this one:



You should set the right access rights to your project files. Do it with this command (OSX and Linux from console): `sudo chmod -R a+w ~/Sites/blog/` Replace `~/Sites/blog/` with path to the root folder of your Yii2 project.

Now check your `common/models` directory and you should be able to see generated `Post` model. Generated file look like this (comments are deleted to save some space):

```
<?php

namespace common\models;

use Yii;

class Post extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'post';
    }

    public function rules()
    {
        return [
```

```
            [['title', 'slug', 'content', 'created_at', 'created_by', 'category_id'], 'required'],
            [['lead_text', 'content'], 'string'],
            [['created_at', 'updated_at'], 'safe'],
            [['created_by', 'updated_by', 'category_id'], 'integer'],
            [['title', 'slug', 'lead_photo'], 'string', 'max' => 128],
            [['meta_description'], 'string', 'max' => 160],
            [['title'], 'unique'],
            [['slug'], 'unique'],
            [['category_id'], 'exist', 'skipOnError' => true, 'targetClass' => Category::className(),
'targetAttribute' => ['category_id' => 'id']],
            [['created_by'], 'exist', 'skipOnError' => true, 'targetClass' => User::className(), 'targetAttribute' =>
['created_by' => 'id']],
            [['updated_by'], 'exist', 'skipOnError' => true, 'targetClass' => User::className(), 'targetAttribute' =>
['updated_by' => 'id']],
        ];
    }

    public function attributeLabels()
    {
        return [
            'id' => 'ID',
            'title' => 'Title',
            'slug' => 'Slug',
            'lead_photo' => 'Lead Photo',
            'lead_text' => 'Lead Text',
            'content' => 'Content',
            'meta_description' => 'Meta Description',
            'created_at' => 'Created At',
            'updated_at' => 'Updated At',
            'created_by' => 'Created By',
            'updated_by' => 'Updated By',
```

```
            'category_id' => 'Category ID',
        ];
    }

    public function getCategory()
    {
        return $this->hasOne(Category::className(), ['id' => 'category_id']);
    }

    public function getCreatedBy()
    {
        return $this->hasOne(User::className(), ['id' => 'created_by']);
    }

    public function getUpdatedBy()
    {
        return $this->hasOne(User::className(), ['id' => 'updated_by']);
    }

    public function getPostTags()
    {
        return $this->hasMany(PostTag::class, ['post_id' => 'id']);
    }
}
```

Explaining generated model

`rules()` method (Validation rules)

When the data for a model is received from end users, it should be validated to make sure it satisfies certain rules (called validation rules, also known as business rules). For example, given a ContactForm model, you may want to make sure all attributes are not empty and the email attribute contains a valid email address. If the values for some attributes do not satisfy the corresponding

business rules, appropriate error messages should be displayed to help the user to fix the errors. [1]

To declare validation rules associated with a model, we will override the `yii\base\Model::rules()` method by returning the rules that the model attributes should satisfy.

`attributeLabels()` method

In this method we are declaring attribute labels. For applications supporting multiple languages, we can translate them here with `Yii::t()` component.

Other methods (`getCategory()`, `getCreatedBy()`, ...)

Remaining model methods are generated on the basis of the relations in database. To stay simple I won't explain how exactly this works. We will take a closer look to this problematics in the another episode. For now we only need to know that `getCategory()` method will give us category which is associated with the current `Post` model. Example:

```
$post->category;
$post->getCategory();
```

And `getPostTags()` method returns all 'PostTag' models associated with the current `Post` model

```
$post->postTags;
$post->getPostTags();
```

Generating other models

We will generate other models the same way as generating the `Post` model.

For generating `Tag` model fill the Model Generator form with:
- Table name: `tag`
- Model class: `Tag`
- Namespace: `common\models`
- Base class: leave the default value
- Database connection ID: leave the default value

For generating `PostTag` model fill the Model Generator form with:

- Table name: `post_tag`
- Model class: `PostTag`
- Namespace: `common\models`
- Base class: leave the default value
- Database connection ID: leave the default value

For generating `Category` model fill the Model Generator form with:
- Table name: `category`
- Model class: `Category`
- Namespace: `common\models`
- Base class: leave the default value
- Database connection ID: leave the default value

Enhancing and finalizing generated models

Attaching Behaviors

We will attach the behaviors to our models statically. To attach a behavior statically, we will override the behaviors() method of the component class to which the behavior is being attached. The behaviors() method returns a list of behavior configurations.

Basic types of Yii 2 behaviors:

1. `yii\behaviors\AttributeBehavior` - Automatically assigns a specified value to one or multiple attributes of an ActiveRecord object when certain events happen.
2. `yii\behaviors\BlameableBehavior` - Automatically fills the specified attributes with the current user ID.
3. `yii\behaviors\SluggableBehavior` - Automatically fills the specified attribute with a value that can be used a slug in a URL.
4. `yii\behaviors\TimestampBehavior` - Automatically fills the specified attributes with the current timestamp.

Post model

In our blog, we will use `BlameableBehavior`, `SluggableBehavior` and `TimestampBehavior`. We will describe it best on the example. So let's finalize our `Post` model. After the `tableName()` method we will add our new `behaviors()` method:

```
public function behaviors()
{
    return [
        'timestamp' => [
            'class' => TimestampBehavior::className(),
            'attributes' => [
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ],
        [
            'class' => BlameableBehavior::className(),
            'createdByAttribute' => 'created_by',
            'updatedByAttribute' => 'updated_by',
        ],
        [
            'class' => SluggableBehavior::className(),
            'attribute' => 'title',
            'slugAttribute' => 'slug',
        ],
    ];
}
```

Do not forget to add these lines to your use section:

```
use yii\db\ActiveRecord;
use yii\behaviors\BlameableBehavior;
use yii\behaviors\SluggableBehavior;
use yii\behaviors\TimestampBehavior;
use yii\db\Expression;
```

Also, we must edit our `rules()` method. We have to remove `created_at`, `created_by` and slug attributes from `required` array because

they are filled automatically before saving model, so we don't want to validate them on the users side:

```
public function rules()
    {
        return [
            [['title', 'content', 'category_id'], 'required'],
            [['lead_text', 'content'], 'string'],
            [['created_at', 'updated_at'], 'safe'],
            [['created_by', 'updated_by', 'category_id'], 'integer'],
            [['title', 'slug', 'lead_photo'], 'string', 'max' => 128],
            [['meta_description'], 'string', 'max' => 160],
            [['title'], 'unique'],
            [['slug'], 'unique'],
            [['category_id'], 'exist', 'skipOnError' => true, 'targetClass' => Category::className(),
'targetAttribute' => ['category_id' => 'id']],
            [['created_by'], 'exist', 'skipOnError' => true, 'targetClass' => User::className(), 'targetAttribute' =>
['created_by' => 'id']],
            [['updated_by'], 'exist', 'skipOnError' => true, 'targetClass' => User::className(), 'targetAttribute' =>
['updated_by' => 'id']],
        ];
    }
```

Category model

Our Category model should look like this after enhancement:

```
<?php

namespace common\models;

use Yii;
use yii\db\ActiveRecord;
use yii\behaviors\SluggableBehavior;
```

```
use yii\behaviors\TimestampBehavior;
use yii\db\Expression;


class Category extends ActiveRecord
{

    public static function tableName()
    {
        return 'category';
    }


    public function behaviors()
    {
        return [
            'timestamp' => [
                'class' => TimestampBehavior::class,
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                'value' => new Expression('NOW()'),
            ],
            [
                'class' => SluggableBehavior::class,
                'attribute' => 'name',
                'slugAttribute' => 'slug',
            ],
        ];
    }
```

```
public function rules()
{
    return [
        [['name'], 'required'],
        [['created_at', 'updated_at'], 'safe'],
        [['name', 'slug'], 'string', 'max' => 64],
        [['meta_description'], 'string', 'max' => 160],
        [['name'], 'unique'],
        [['slug'], 'unique']
    ];
}


public function attributeLabels()
{
    return [
        'id' => 'ID',
        'name' => 'Name',
        'slug' => 'Slug',
        'meta_description' => 'Meta Description',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
    ];
}


public function getPosts()
{
    return $this->hasMany(Post::class\Name(), ['category_id' => 'id']);
```

```
    }
}
```

Tag model

Our Tag model should look like this after enhancement:

```
<?php

namespace common\models;

use Yii;
use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;
use yii\db\Expression;



class Tag extends ActiveRecord
{

    public static function tableName()
    {
        return 'tag';
    }



    public function behaviors()
    {
        return [
            'timestamp' => [
                'class' => TimestampBehavior::className(),
                'attributes' => [
```

```
                ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
            ],
            'value' => new Expression('NOW()'),
        ]
    ];
}



public function rules()
{
    return [
        [['name'], 'required'],
        [['created_at', 'updated_at'], 'safe'],
        [['name'], 'string', 'max' => 64],
        [['name'], 'unique']
    ];
}



public function attributeLabels()
{
    return [
        'id' => 'ID',
        'name' => 'Name',
        'created_at' => 'Created At',
        'updated_at' => 'Updated At',
    ];
}
```

```
    public function getPostTags()
    {
        return $this->hasMany(PostTag::className(), ['tag_id' => 'id']);
    }
}
```

We will continue building our blog in next episode. If do you have any questions regarding to this episode, please write them below to the comments section.

Download files from this episode: episode_04.zip.

Episode 5 - Generating CRUD code with Gii

What is CRUD

CRUD stands for Create, Read, Update, and Delete, representing the four common tasks taken with data on most Web sites. CRUD Generator generates a controller and views that implement CRUD operations for the specified data model.

Generating CRUD code

We can generate CRUD code directly in the Gii code generator. To go to CRUD generator click on the “CRUD Generator” button or go to this address:

```
http://localhost/blog/frontend/web/gii/crud
```

There are several form fields which we have to fill:
Model class - This is the ActiveRecord class associated with the table that CRUD will be built upon.
Search Model Class - This is the name of the search model class to be generated.
Controller Class - This is the name of the controller class to be generated.
View Path - Specify the directory for storing the view scripts for the controller.
Widget Used in Index Page - This is the widget type to be used in the index page to display list of the models.

Generating CRUD code for Posts

To generate CRUD code for Post model, fill the form with:
Model Class - `common\models\Post`
Search Model Class - `common\models\PostSearch`
Controller Class - `frontend\controllers\PostController`
View Path - `@frontend/views/post`
Widget Used in Index Page - **ListView**

After filling the fields, click on the “Preview” button and you should see:

Preview Generate

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

Create Unchanged Overwrite

Code File	Action	
controllers/PostController.php	create	✓
/Users/jpulik/Sites/blog/common/models/PostSearch.php	create	✓
views/post/_form.php	create	✓
views/post/_search.php	create	✓
views/post/create.php	create	✓
views/post/index.php	create	✓
views/post/update.php	create	✓
views/post/view.php	create	✓

Then click on the "Generate" button to generate CRUD code.

Now go to this address:

```
http://localhost/blog/frontend/web/post/index
```

and you should see our newly generated Posts index page:

My-Yii Blog Home About Contact Signup Login

Home / Posts

Posts

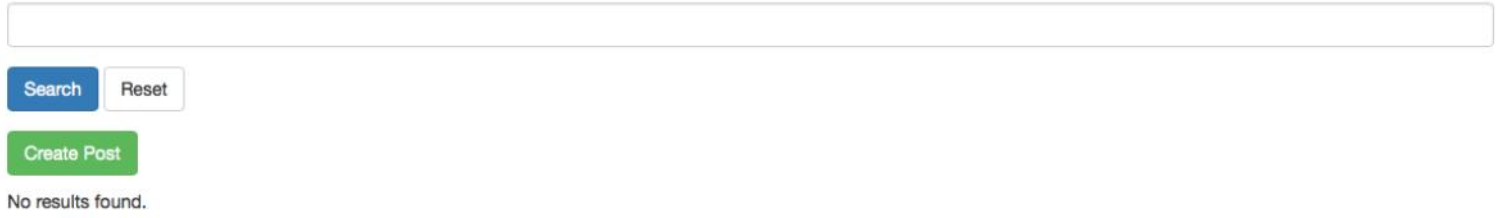
ID

Title

Slug

Lead Photo

Lead Text



We will customize generated code in the next episode. In this episode we will only **generate** necessary CRUD code for the models which we generated in the previous episode.

Before we will continue, create new `.htaccess` file in `backend\web\` to route backend requests properly:

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on



RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d



RewriteRule . index.php
```

We will do the same for the Category and Tag models.

Generating CRUD code for Categories

Fill the Gii form with: Model Class - `common\models\Category`
Search Model Class - `common\models\CategorySearch`
Controller Class - `backend\controllers\CategoryController`

View Path - `@backend/views/category`

Widget Used in Index Page - **GridView**

Categories index page:

```
http://localhost/blog/backend/web/index.php?r=category%2Findex
```

We have generated CRUD code for Categories to the `backend` because we want to manage categories from there.

Generating CRUD code for Tags

Fill the Gii form with: Model Class - `common\models\Tag`

Search Model Class - `common\models\TagSearch`

Controller Class - `backend\controllers\TagController`

View Path - `@backend/views/tag`

Widget Used in Index Page - GridView

Tags index page:

```
http://localhost/blog/backend/web/index.php?r=tag%2Findex
```

We have also generated CRUD code for Tags to the `backend` because we want to manage tags from there.

In this episode we have successfully generated all CRUD code which we will need in our blog. Generated CRUD code is just the skeleton (avoids DRY) of the real code which we want to have. So we will customize generated code in the next episodes to bring all the functionality we want. If do you have any questions regarding to this episode, please write them below to the comments section.

Download files from this episode: episode_05.zip.