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?

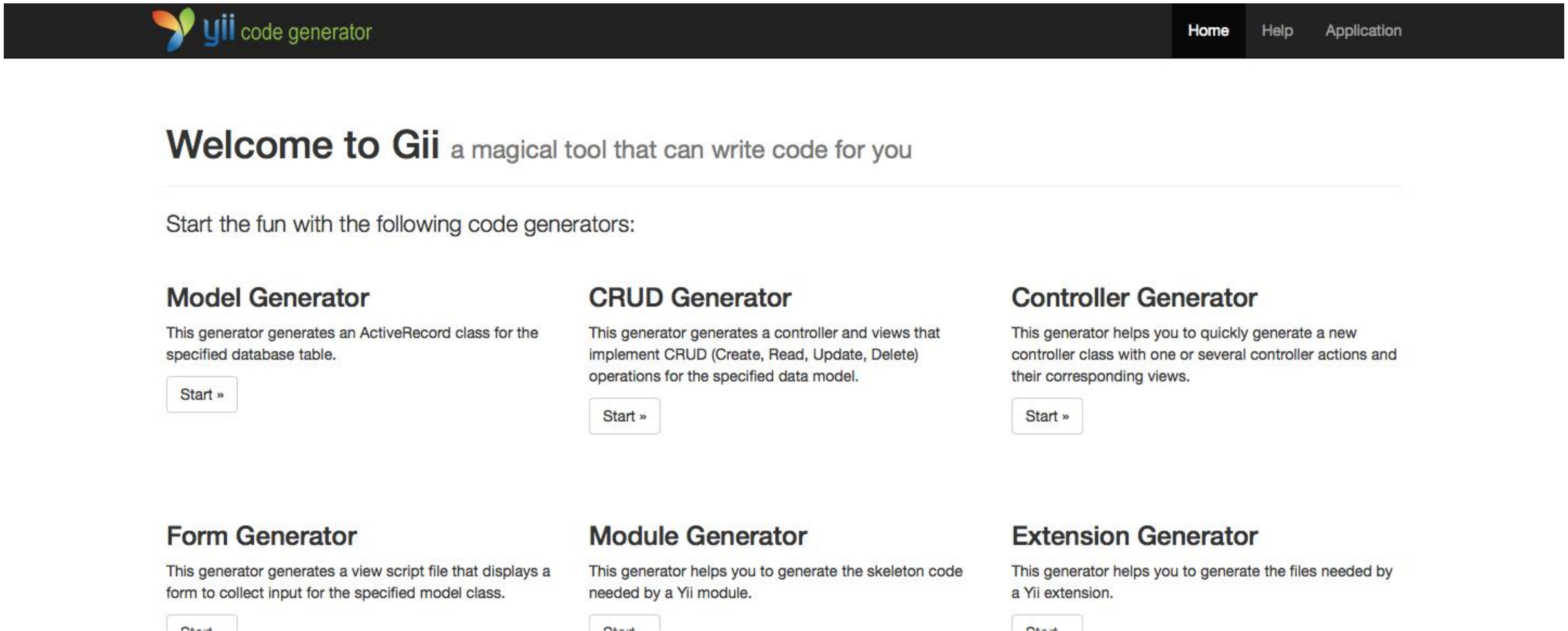
Gii is web-based code generator. With Gii we can quickly generate models, forms, modules, CRUD, etc.. Gii also provides command line interface for people who prefer to work with their console.

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:



Welcome to Gii a magical tool that can write code for you

Start the fun with the following code generators:

Model Generator

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

CRUD Generator

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

Controller Generator

This generator helps you to quickly generate a new controller class with one or several controller actions and their corresponding views.

Form Generator

This generator generates a view script file that displays a form to collect input for the specified model class.

Module Generator

This generator helps you to generate the skeleton code needed by a Yii module.

Extension Generator

This generator helps you to generate the files needed by a Yii extension.

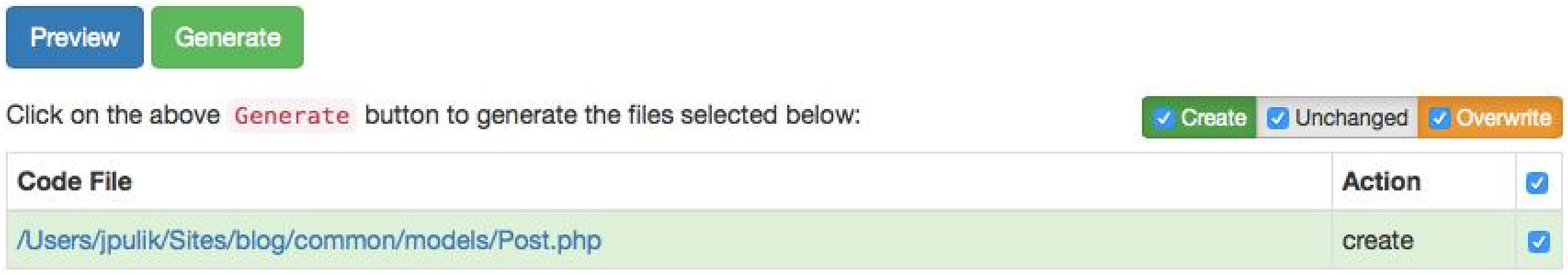
Generating Models

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:



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

Code File	Action	✓
/Users/jpulik/Sites/blog/common/models/Post.php	create	✓

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

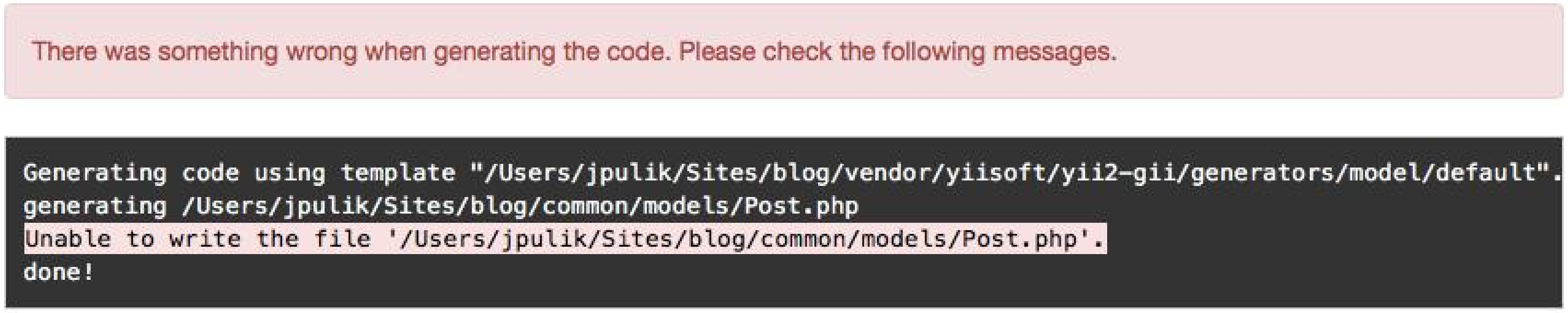
```
The code has been generated successfully.
```

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:



There was something wrong when generating the code. Please check the following messages.

```
Generating code using template "/Users/jpulik/Sites/blog/vendor/yiisoft/yii2-gii/generators/model/default".
generating /Users/jpulik/Sites/blog/common/models/Post.php
Unable to write the file '/Users/jpulik/Sites/blog/common/models/Post.php'.
done!
```

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::className(), ['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;
```

```php
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::className(),
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                'value' => new Expression('NOW()'),
            ],
            [
                'class' => SluggableBehavior::className(),
                '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::className(), ['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.