Генерируем классы фикстур в Yii2

# Статья с сайта <http://www.elisdn.ru/blog/91/gii-fixture-generator>

# Генерируем классы фикстур в Yii2

При входе в чужой проект (или при доработке своего старого) часто сталкиваемся с отсутствием хоть каких-то автоматических тестов. А без них весьма неприятно ковыряться в исходниках, так как есть постоянный страх что-то сломать. Поэтому первым делом приходится внедрять автотесты. Посмотрим, чем Gii может нам помочь.

Если в проекте уже используются миграции для изменения структуры базы, то можно спокойно подключить FixtureHelper (как мы это делали ранее при подготовке [своих тестов](http://www.elisdn.ru/blog/84/seo-service-on-yii2-application-testing)). Этот модуль позволяет использовать фикстуры в виде PHP-массивов не только в интеграционных, но и в функциональных (по мнению Codeception) и приёмочных тестах вместо использования просто SQL-дампа.

Если же миграций нет, то либо делаем их, либо одновременно используем голый дамп со структурой и наборы фикстур для заполнения таблиц данными.

Для каждой таблицы в базе нужно создать отдельный класс фикстуры:

namespace tests\codeception\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture

{

public $modelClass = 'app\models\User';

public $dataFile = '@tests/codeception/fixtures/data/user.php';

}

и приложить файл с данными data/user.php:

return [

[

'id' => 1,

'username' => 'admin',

'email' => 'admin@example.com',

'auth\_key' => 'eckb2DLY9uv6r1hM6D73eoHPvv6BfnXc',

'password\_hash' => '$2y$13$D8are...',

'password\_reset\_token' => null,

'created\_at' => 1439635619,

'updated\_at' => 1439635619,

'status' => 10,

],

[

'id' => 2,

'username' => 'user',

...

],

];

или нагенерировать данные с помощью расширения yii2-faker.

Если в проекте около пятидесяти таблиц, то копипаст классов UserFixture, CategoryFixture и т.п. с вдумчивым и аккуратным заполнением всех data-файлов может занять несколько часов.

Но, по сути, всю эту информацию можно нагенерировать также, как мы создаём ActiveRecord-модели в Gii на основе таблиц. При этом получаем нужные классы с полями на основе списка колонок в таблице.

Это хороший повод разобраться во внутренностях генераторов и посмотреть, чем в этом плане может быть полезен Gii.

## Исследуем генераторы Gii

Расширение yii2-gii по умолчанию включено в конфигурацию стандартных приложений, поэтому устанавливать его отдельно не требуется.

В директории vendor/yiisoft/yii2-gii проекта нас интересует только поддиректория generators:

generators

├── controller

├── crud

├── extension

├── form

├── model

└── module

Рассмотрим, что из себя представляет генератор контроллеров:

generators

└── controller

   ├── default

   │   ├── controller.php

   │   └── view.php

   ├── form.php

   └── Generator.php

Ключевым здесь является класс Generator, хранящий всю информацию, необходимую модулю Gii. Вверху имеется название и описание:

namespace yii\gii\generators\controller;

...

class Generator extends \yii\gii\Generator

{

public $controllerClass;

public $viewPath;

public $baseClass = 'yii\web\Controller';

public $actions = 'index';

public function getName()

{

return 'Controller Generator';

}

public function getDescription()

{

return 'This generator helps you to quickly generate a new controller class with

one or several controller actions and their corresponding views.';

}

...

}

Именно эти данные выводятся на главной странице Gii:

Каждый генератор наследуется от абстрактного класса yii\gii\Generator, который, по традициям Yii2, по совместительству представляет из себя и генератор, и модель:

namespace yii\gii;

...

use yii\base\Model;

abstract class Generator extends Model

{

...

}

И, соответственно, содержит в себе поля и методы модели rules() и attributeLabels() и ещё кучу либо важных вещей, либо вспомогательной непонятной инфраструктуры.

Интерфейс у генератора не определён, нет разнесения на форму и генерацию, а также все методы публичные. Поэтому методом тыка и изучения существующих наследников класса можем предположить, что нам интересна только половина из них:

namespace yii\gii;

...

use yii\base\Model;

abstract class Generator extends Model

{

public $templates = [];

public $template = 'default';

public $enableI18N = false;

public $messageCategory = 'app';

abstract public function getName();

public function getDescription()

{

return '';

}

abstract public function generate();

public function rules()

{

return [

[['template'], 'required', 'message' => 'A code template must be selected.'],

[['template'], 'validateTemplate'],

];

}

public function attributeLabels()

{

return [

'enableI18N' => 'Enable I18N',

'messageCategory' => 'Message Category',

];

}

public function requiredTemplates()

{

return [];

}

public function stickyAttributes()

{

return ['template', 'enableI18N', 'messageCategory'];

}

public function hints()

{

return [

'enableI18N' => 'This indicates whether...',

'messageCategory' => 'This is the category...',

];

}

public function autoCompleteData()

{

return [];

}

public function successMessage()

{

return 'The code has been generated successfully.';

}

public function formView() { ... }

public function defaultTemplate() { ... }

public function loadStickyAttributes() { ... }

public function saveStickyAttributes() { ... }

public function getStickyDataFile() { ... }

public function save($files, $answers, &$results) { ... }

public function getTemplatePath() { ... }

public function render($template, $params = []) { ... }

public function validateTemplate() { ... }

public function validateClass($attribute, $params) { ... }

public function validateNewClass($attribute, $params) { ... }

public function validateMessageCategory() { ... }

public function isReservedKeyword($value) { ... }

public function generateString($string = '', $placeholders = []) { ... }

}

От такого класса всё у нас и наследуется. Например, тот самый генератор контроллера успешно добавляет свои поля и переопределяет методы для построения формы:

namespace yii\gii\generators\controller;

class Generator extends \yii\gii\Generator

{

public $controllerClass;

public $viewPath;

public $baseClass = 'yii\web\Controller';

public $actions = 'index';

public function rules()

{

return array\_merge(parent::rules(), [

[['controllerClass', 'actions', 'baseClass'], 'filter', 'filter' => 'trim'],

[['controllerClass', 'baseClass'], 'required'],

['controllerClass', 'match', 'pattern' => '/^[\w\\\\]\*Controller$/'],

['controllerClass', 'validateNewClass'],

['baseClass', 'match', 'pattern' => '/^[\w\\\\]\*$/'],

['actions', 'match', 'pattern' => '/^[a-z][a-z0-9\\-,\\s]\*$/'],

['viewPath', 'safe'],

]);

}

public function attributeLabels()

{

return [

'baseClass' => 'Base Class',

'controllerClass' => 'Controller Class',

'viewPath' => 'View Path',

'actions' => 'Action IDs',

];

}

public function stickyAttributes()

{

return ['baseClass'];

}

public function hints()

{

return [

'controllerClass' => 'This is the name of the controller class to be generated...',

'actions' => 'Provide one or multiple action IDs to generate...',

'viewPath' => 'Specify the directory for storing the view scripts for the controller...',

'baseClass' => 'This is the class that the new controller class will extend from...',

];

}

...

}

Ещё в той же папке:

generators

└── controller

   ├── default

   │   ├── controller.php

   │   └── view.php

   ├── form.php

   └── Generator.php

имеется файл form.php, выводящий нужные поля формы:

<?php

/\* @var $this yii\web\View \*/

/\* @var $form yii\widgets\ActiveForm \*/

/\* @var $generator yii\gii\generators\controller\Generator \*/

echo $form->field($generator, 'controllerClass');

echo $form->field($generator, 'actions');

echo $form->field($generator, 'viewPath');

echo $form->field($generator, 'baseClass');

На основе этих полей и правил их валидации Gii и формирует полноценный интерфейс:

Мы видим, что поле Base Class выведено с жёлтым фоном. Это из-за того, что в коде это поле добавлено в список так называемых «липких» атрибутов:

public function stickyAttributes()

{

return ['baseClass'];

}

Этот список используется в методах базового класса loadStickyAttributes() и saveStickyAttributes(), где значения этих полей сохраняются в файл в недрах папки runtime и загружаются оттуда же:

abstract class Generator extends Model

{

public function loadStickyAttributes()

{

$stickyAttributes = $this->stickyAttributes();

$path = $this->getStickyDataFile();

if (is\_file($path)) {

$result = json\_decode(file\_get\_contents($path), true);

if (is\_array($result)) {

foreach ($stickyAttributes as $name) {

if (isset($result[$name])) {

$this->$name = $result[$name];

}

}

}

}

}

public function saveStickyAttributes()

{

$stickyAttributes = $this->stickyAttributes();

...

$path = $this->getStickyDataFile();

@mkdir(dirname($path), 0755, true);

file\_put\_contents($path, json\_encode($values));

}

public function getStickyDataFile()

{

return Yii::$app->getRuntimePath() . '/gii-' . Yii::getVersion() . '/' . str\_replace('\\', '-', get\_class($this)) . '.json';

}

...

}

Соответственно, введённое в такое поле значение сохранится в файле и так и будет выводится в форме. Его не придётся каждый раз набирать вручную.

Продолжим наше исследование. В подпапке default помещены шаблоны для получаемых файлов. Прямо там рендерится класс сгенерированного контроллера в default/controller.php:

<?php

/\*\*

\* This is the template for generating a controller class file.

\*/

use yii\helpers\Inflector;

use yii\helpers\StringHelper;

/\* @var $this yii\web\View \*/

/\* @var $generator yii\gii\generators\controller\Generator \*/

echo "<?php\n";

?>

namespace <?= $generator->getControllerNamespace() ?>;

class <?= StringHelper::basename($generator->controllerClass) ?> extends <?= '\\' . trim($generator->baseClass, '\\') . "\n" ?>

{

<?php foreach ($generator->getActionIDs() as $action): ?>

public function action<?= Inflector::id2camel($action) ?>()

{

return $this->render('<?= $action ?>');

}

<?php endforeach; ?>

}

и аналогично формируется представление в default/view.php:

<?php

/\*\*

\* This is the template for generating an action view file.

\*/

/\* @var $this yii\web\View \*/

/\* @var $generator yii\gii\generators\controller\Generator \*/

/\* @var $action string the action ID \*/

echo "<?php\n";

?>

/\* @var $this yii\web\View \*/

<?= "?>" ?>

<h1><?= $generator->getControllerID() . '/' . $action ?></h1>

<p>

You may change the content of this page by modifying

the file <code><?= '<?=' ?> \_\_FILE\_\_; ?></code>.

</p>

Прямо так вместо HTML-разметки «печатаем» исходный код.

Путь до этой папки формируется в последнем поле Code Template выводимой формы. Для задания папки шаблонов имеются отдельные поля в базовом классе:

abstract class Generator extends Model

{

public $templates = [];

public $template = 'default';

...

}

которые наследуются во все генераторы. Поэтому можно к любому генератору добавить свою папку с шаблонами под именем Super Controller и, при желании, сделать свой шаблон главным:

$config['modules']['gii'] = [

'class' => 'yii\gii\Module',

'generators' => [

'controller' => [

'class' => 'yii\gii\generators\controller\Generator',

'templates' => [

'Super Controller' => '@app/templates/controller',

],

'template' => 'Super Controller',

],

],

];

И он будет выводиться рядом со стандартным:

Это полезно, например, если вы сделали специфический контроллер или вёрстку представлений для своих CRUD и хотите подменить стандартный шаблон.

Или можно полностью переопределить стандартный шаблон default на свой:

'controller' => [

'class' => 'yii\gii\generators\controller\Generator',

'templates' => [

'default' => '@app/templates/controller',

],

],

чтобы не было возможности воспользоваться ненужным стандартным.

Пойдём далее. Когда форма успешно отправлена и провалидирована, в действие вступает самый главный метод genarate():

namespace yii\gii\generators\controller;

class Generator extends \yii\gii\Generator

{

...

public function generate()

{

$files = [];

$files[] = new CodeFile(

$this->getControllerFile(),

$this->render('controller.php')

);

foreach ($this->getActionIDs() as $action) {

$files[] = new CodeFile(

$this->getViewFile($action),

$this->render('view.php', ['action' => $action])

);

}

return $files;

}

public function getActionIDs()

{

$actions = array\_unique(preg\_split('/[\s,]+/', $this->actions, -1, PREG\_SPLIT\_NO\_EMPTY));

sort($actions);

return $actions;

}

public function getControllerFile()

{

return Yii::getAlias('@' . str\_replace('\\', '/', $this->controllerClass)) . '.php';

}

public function getViewFile($action)

{

if (empty($this->viewPath)) {

return Yii::getAlias('@app/views/' . $this->getControllerID() . "/$action.php");

} else {

return Yii::getAlias($this->viewPath . "/$action.php");

}

}

}

Ему уже нужно отрендерить шаблоны и вернуть результат в виде массива объектов класса CodeFile:

$files[] = new CodeFile($fileName, $this->render('controller.php'));

А сам Gii потом займётся их сохранением. Или, если такие файлы уже есть, спросит, нужно ли их перезаписать.

## Подготовка структуры расширения

Как уже говорили в начале, генератор нужен для многих проектов. Поэтому сразу выложим его в публичный доступ.

Сделаем заготовку расширения по аналогии с нашим [yii2-hybrid-authmanager](http://www.elisdn.ru/blog/88/seo-service-on-yii2-composer-extensions) и с пустыми файлам в папке src:

generator

├── src

│   ├── default

│   │   ├── class.php

│   │   └── data.php

│   ├── form.php

│   └── Generator.php

├── tests

│   ├── runtime

│   │ └── .gitignore

│   ├── bootstrap.php

│   └── TestCase.php

├── .gitignore

├── composer.json

├── phpunit.xml.dist

├── LICENCE.md

└── README.md

В .gitignore поместим:

/vendor

/composer.lock

Файл phpunit.xml.dist оставим стандартным:

<?xml version="1.0" encoding="utf-8"?>

<phpunit bootstrap="./tests/bootstrap.php"

colors="true"

convertErrorsToExceptions="true"

convertNoticesToExceptions="true"

convertWarningsToExceptions="true"

stopOnFailure="false">

<testsuites>

<testsuite name="Test Suite">

<directory>./tests</directory>

</testsuite>

</testsuites>

<filter>

<whitelist>

<directory suffix=".php">./src/</directory>

</whitelist>

</filter>

</phpunit>

И изменим немного composer.json:

{

"name": "elisdn/yii2-gii-fixture-generator",

"description": "Fixture class generator for Gii module of Yii2 Framework.",

"type": "yii2-extension",

"keywords": ["yii2", "yii 2", "gii", "fixture"],

"license": "BSD-3-Clause",

"authors": [

{

"name": "Dmitriy Yeliseyev",

"email": "mail@elisdn.ru",

"homepage": "http://www.elisdn.ru"

}

],

"support": {

"issues": "https://github.com/ElisDN/yii2-gii-fixture-generator/issues?state=open",

"source": "https://github.com/ElisDN/yii2-gii-fixture-generator"

},

"require": {

"yiisoft/yii2-gii": "~2.0"

},

"require-dev": {

"phpunit/phpunit": "4.\*"

},

"autoload": {

"psr-4": {

"elisdn\\gii\\fixture\\": "src/",

"elisdn\\gii\\fixture\\tests\\": "tests/"

}

},

"extra": {

"asset-installer-paths": {

"npm-asset-library": "vendor/npm",

"bower-asset-library": "vendor/bower"

}

}

}

Здесь мы проставим зависимость от пакета yiisoft/yii2-gii, который своими зависимостями подтянет нам сам фреймворк.

В tests/bootstrap.php впишем инициализацию окружения:

<?php

defined('YII\_DEBUG') or define('YII\_DEBUG', true);

defined('YII\_ENV') or define('YII\_ENV', 'test');

require(\_\_DIR\_\_ . '/../vendor/autoload.php');

require(\_\_DIR\_\_ . '/../vendor/yiisoft/yii2/Yii.php');

Создадим папку tests/runtime с файлом .gitignore:

\*

!.gitignore

Она пригодится для тестовых нужд.

В базовом классе для тестов tests/TestCase.php будем запускать тестовое приложение:

namespace elisdn\gii\fixture\tests;

use yii\console\Application;

abstract class TestCase extends \PHPUnit\_Framework\_TestCase

{

protected function setUp()

{

parent::setUp();

$this->mockApplication();

}

protected function tearDown()

{

$this->destroyApplication();

parent::tearDown();

}

protected function mockApplication()

{

new Application([

'id' => 'testapp',

'basePath' => \_\_DIR\_\_,

'vendorPath' => dirname(\_\_DIR\_\_) . '/vendor',

'runtimePath' => \_\_DIR\_\_ . '/runtime',

'aliases' => [

'@tests' => \_\_DIR\_\_,

],

]);

}

protected function destroyApplication()

{

\Yii::$app = null;

}

}

Теперь начнём, собственно, делать генератор.

## Написание своего генератора

Первым делом, добавим название и описание для выводя на стартовой странице и в меню Gii:

namespace elisdn\gii\fixture;

class Generator extends \yii\gii\Generator

{

public function getName()

{

return 'Fixture Class Generator';

}

public function getDescription()

{

return 'This generator generates fixture class for existing model class and prepares fixture data file.';

}

}

Теперь определимся, какие поля в форме нам нужны.

Нам будет необходимо на основе существующей ActiveRecord-модели вроде app\models\User сгенерировать класс фикстуры и её набор данных.

Соответственно, можно добавить наши поля и просить пользователя вводить пути в них:

class Generator extends \yii\gii\Generator

{

public $modelClass;

public $fixtureClass;

public $dataFile;

}

Тогда человеку нужно будет заполнить три поля значениями:

app\models\User

tests\codeception\fixtures\UserFixture

tests/codeception/fixtures/data/user.php

Но вбивать полные пути и полные пространства имён классов жутко неудобно. Для удобства это можно разбить на пространство имён и имя класса; на путь и имя файла – сделать всё, чтобы он мог вбивать только имя класса и файла:

app\models\User

UserFixture

user.php

И даже вообще сделать эти поля необязательными для заполнения, а пространства имён и пути снабдить значениями по умолчанию и сделать их «липкими» атрибутами, чтобы они запоминались:

class Generator extends \yii\gii\Generator

{

public $modelClass;

public $fixtureClass;

public $fixtureNs = 'tests\codeception\fixtures';

public $dataFile;

public $dataPath = '@tests/codeception/fixtures/data';

public function stickyAttributes()

{

return array\_merge(parent::stickyAttributes(), ['fixtureNs', 'dataPath']);

}

}

Ещё было бы неплохо сделать так, чтобы можно было считывать тестовые данные прямо из имеющихся записей в базе. Добавим для этого флаг $grabData:

class Generator extends \yii\gii\Generator

{

...

public $grabData = false;

}

Далее воспользуемся своей фантазией (и имеющимися в базовом классе валидаторами) и напишем правила валидации, имена полей и подсказки при наведении мыши:

class Generator extends \yii\gii\Generator

{

public function rules()

{

return array\_merge(parent::rules(), [

[['modelClass', 'fixtureClass', 'fixtureNs', 'dataPath'], 'filter', 'filter' => 'trim'],

[['modelClass', 'fixtureNs', 'dataPath'], 'required'],

[['modelClass', 'fixtureNs'], 'match', 'pattern' => '/^[\w\\\\]\*$/', 'message' => 'Only word characters and backslashes are allowed.'],

[['fixtureClass'], 'match', 'pattern' => '/^\w+$/', 'message' => 'Only word characters are allowed.'],

[['dataFile'], 'match', 'pattern' => '/^\w+\.php$/', 'message' => 'Only php files are allowed.'],

[['modelClass'], 'validateClass', 'params' => ['extends' => ActiveRecord::className()]],

[['dataPath'], 'match', 'pattern' => '/^@?\w+[\\-\\/\w]\*$/', 'message' => 'Only word characters, dashes, slashes and @ are allowed.'],

[['dataPath'], 'validatePath'],

[['grabData'], 'boolean'],

]);

}

public function validatePath($attribute)

{

$path = Yii::getAlias($this->$attribute, false);

if ($path === false || !is\_dir($path)) {

$this->addError($attribute, 'Path does not exist.');

}

}

public function attributeLabels()

{

return array\_merge(parent::attributeLabels(), [

'modelClass' => 'Model Class',

'fixtureClass' => 'Fixture Class Name',

'fixtureNs' => 'Fixture Class Namespace',

'dataFile' => 'Fixture Data File',

'dataPath' => 'Fixture Data Path',

'grabData' => 'Grab Existing DB Data',

]);

}

public function hints()

{

return array\_merge(parent::hints(), [

'modelClass' => 'This is the model class...',

'fixtureClass' => 'This is the name for fixture class..',

'fixtureNs' => 'This is the namespace for fixture class file..',

'dataFile' => 'This is the name for the generated fixture data file..',

'dataPath' => 'This is the root path to keep the generated fixture data files...',

'grabData' => 'If checked, the existed data from database will be grabbed into data file.',

]);

}

}

Ну и, до кучи, определим, какие файлы шаблонов нам будут нужны, если вдруг кто-то подсунет нам свою папку шаблонов в $templates:

class Generator extends \yii\gii\Generator

{

...

public function requiredTemplates()

{

return ['class.php', 'data.php'];

}

}

В файле form.php соберём форму с нашими полями:

<?php

/\* @var $this yii\web\View \*/

/\* @var $form yii\widgets\ActiveForm \*/

/\* @var $generator elisdn\gii\fixture\Generator \*/

echo $form->field($generator, 'modelClass');

echo $form->field($generator, 'fixtureClass');

echo $form->field($generator, 'fixtureNs');

echo $form->field($generator, 'dataFile');

echo $form->field($generator, 'dataPath');

echo $form->field($generator, 'grabData')->checkbox();

Теперь перейдём к самому методу generate(). На основе введённых данных или по умолчанию на основе имени модели:

class Generator extends \yii\gii\Generator

{

...

public function generate()

{

$files = [];

$files[] = new CodeFile(

Yii::getAlias('@' . str\_replace('\\', '/', $this->fixtureNs)) . '/' . $this->getFixtureClassName() . '.php',

$this->render('class.php')

);

$files[] = new CodeFile(

Yii::getAlias($this->dataPath) . '/' . $this->getDataFileName(),

$this->render('data.php', ['items' => $this->getFixtureData()])

);

return $files;

}

public function getDataFileName()

{

if (!empty($this->dataFile)) {

return $this->dataFile;

} else {

return strtolower(pathinfo(str\_replace('\\', '/', $this->modelClass), PATHINFO\_BASENAME)) . '.php';

}

}

public function getFixtureClassName()

{

if (!empty($this->fixtureClass)) {

return $this->fixtureClass;

} else {

return pathinfo(str\_replace('\\', '/', $this->modelClass), PATHINFO\_BASENAME) . 'Fixture';

}

}

protected function getFixtureData()

{

...

return $items;

}

}

А в методе getFixtureData() будем формировать пустую заготовку при $grabData равном false:

[

[

'id' => '',

'username' => '',

'email' => '',

'password\_hash' => '',

'password\_reset\_token' => null,

'status' => '',

'created\_at' => '',

],

]

либо заполнять значениями из базы данных при true:

[

[

'id' => 1,

'username' => 'user',

'email' => 'user@example.com',

'password\_hash' => 'dsfg34656tgfs3...',

'password\_reset\_token' => null,

'status' => 1,

'created\_at' => 1439635619,

],

[

'id' => 2,

'username' => 'admin',

'email' => 'admin@example.com',

'password\_hash' => '47fy4d45345egg...',

'password\_reset\_token' => null,

'status' => 1,

'created\_at' => 1439635813,

],

]

Такие массивы можно сформировать примерно так:

class Generator extends \yii\gii\Generator

{

...

protected function getFixtureData()

{

/\*\* @var \yii\db\ActiveRecord $modelClass \*/

$modelClass = $this->modelClass;

$items = [];

if ($this->grabData) {

$orderBy = array\_combine($modelClass::primaryKey(), array\_fill(0, count($modelClass::primaryKey()), SORT\_ASC));

foreach ($modelClass::find()->orderBy($orderBy)->asArray()->each() as $row) {

$item = [];

foreach ($row as $name => $value) {

if (is\_null($value)) {

$encValue = 'null';

} elseif (preg\_match('/^(0|[1-9-]\d\*)$/s', $value)) {

$encValue = $value;

} else {

$encValue = var\_export($value, true);

}

$item[$name] = $encValue;

}

$items[] = $item;

}

} else {

$item = [];

foreach ($modelClass::getTableSchema()->columns as $column) {

$item[$column->name] = $column->allowNull ? 'null' : '\'\'';

}

$items[] = $item;

}

return $items;

}

}

Теперь можно сформировать шаблоны для сгенерированных файлов.

В default/class.php у нас будет производиться создание класса:

<?php

/\* @var $this yii\web\View \*/

/\* @var $generator elisdn\gii\fixture\Generator \*/

echo "<?php\n";

?>

namespace <?= $generator->fixtureNs ?>;

use yii\test\ActiveFixture;

class <?= $generator->getFixtureClassName() ?> extends ActiveFixture

{

public $modelClass = '<?= $generator->modelClass ?>';

public $dataFile = '<?= $generator->dataPath . '/' . $generator->getDataFileName() ?>';

}

А в default/data.php – создание файла тестовых данных:

<?php

/\* @var $this yii\web\View \*/

/\* @var $generator elisdn\gii\fixture\Generator \*/

/\* @var $items array \*/

echo "<?php\n";

?>

return [

<?php foreach ($items as $item): ?>

[

<?php foreach ($item as $name => $value): ?> '<?= $name ?>' => <?= $value ?>,

<?php endforeach; ?>

],

<?php endforeach; ?>

];

И, для красоты, переопределим сообщение об успешной генерации:

class Generator extends \yii\gii\Generator

{

...

public function successMessage()

{

$output = <<<EOD

<p>The fixture has been generated successfully.</p>

<p>To access the data, you need to add this to your test class:</p>

EOD;

$id = $this->getFixtureId();

$class = $this->fixtureNs . '\\' . $this->getFixtureClassName();

$file = $this->dataPath . '/' . $this->getDataFileName();

$code = <<<EOD

<?php

public function fixtures()

{

return [

'{$id}' => [

'class' => \\{$class}::className(),

'dataFile' => '{$file}',

],

];

}

EOD;

return $output . '<pre>' . highlight\_string($code, true) . '</pre>';

}

}

Теперь подключим генератор в конфигурационном файле в любой наш проект. Это ещё не Composer-пакет и мы его не устанавливали, так что внесём класс в автозагрузку явно через $classMap:

if (YII\_ENV\_DEV) {

...

Yii::setAlias('@tests', dirname(\_\_DIR\_\_) . '/tests');

Yii::$classMap['elisdn\gii\fixture\Generator'] = dirname(\_\_DIR\_\_) . '/generator/src/Generator.php';

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

$config['modules']['gii'] = [

'class' => 'yii\gii\Module',

'generators' => [

'fixture' => [

'class' => 'elisdn\gii\fixture\Generator',

],

],

];

}

Мы ещё указали, куда должен вести псевдоним @tests, чтобы наш компонент знал, в какую папку всё сохранять.

И попробуем открыть Gii и что-нибудь сгенерировать:

Ещё остался один небольшой момент. Имя класса UserFixture и имя файла user.php можно либо вбить вручную, либо оставить пустым для автоматической генерации. Но можно дополнить интерфейс по примеру генератора ActiveRecord-моделей, когда после введения имени таблицы имя модели и ActiveQuery-класса заполнялись автоматически.

Добавим в структуру свой JavaScript-файл и класс GeneratorAsset:

generator

├── src

│   ├── assets

│   │   └── generator.js

│   ├── default

│   │   ├── class.php

│   │   └── data.php

│   ├── form.php

│   ├── Generator.php

│   └── GeneratorAsset.php

└── ...

В скрипте assets/generator.js сделаем автоподстановку имени класса и имени файла в соответствующие поля:

(function ($) {

$('#generator-modelclass').on('blur', function () {

var modelClass = $(this).val();

if (modelClass !== '') {

var fixtureClassInput = $('#generator-fixtureclass');

var fixtureClass = fixtureClassInput.val();

if (fixtureClass === '') {

fixtureClass = modelClass.split('\\').slice(-1)[0] + 'Fixture';

fixtureClassInput.val(fixtureClass);

}

var dataFileInput = $('#generator-datafile');

var dataFile = dataFileInput.val();

if (dataFile === '') {

dataFile = modelClass.split('\\').slice(-1)[0].toLowerCase() + '.php';

dataFileInput.val(dataFile);

}

}

});

})(jQuery);

И в GeneratorAsset сконфигурируем комплект ресурсов:

namespace elisdn\gii\fixture;

use yii\web\AssetBundle;

class GeneratorAsset extends AssetBundle

{

public $sourcePath = '@elisdn/gii/fixture/assets';

public $js = [

'generator.js',

];

public $depends = [

'yii\web\JqueryAsset',

];

}

Его мы будем подключать на странице формы:

<?php

use elisdn\gii\fixture\GeneratorAsset;

/\* @var $this yii\web\View \*/

/\* @var $form yii\widgets\ActiveForm \*/

/\* @var $generator elisdn\gii\fixture\Generator \*/

GeneratorAsset::register($this);

echo $form->field($generator, 'modelClass');

echo $form->field($generator, 'fixtureClass');

echo $form->field($generator, 'fixtureNs');

echo $form->field($generator, 'dataFile');

echo $form->field($generator, 'dataPath');

echo $form->field($generator, 'grabData')->checkbox();

Компонент практически готов. Осталось удостовериться в правильности его работы.

## Написание тестов

Первоначальную подготовку тестового окружения мы уже произвели при создании структуры директорий расширения.

Нам потребуется протестировать генератор на некой ActiveRecord-модели и попробовать с помощью него спарсить данные из базы. Соответственно, нам потребуется модель и тестовая база данных для неё. Также нам нужны будут папки для указания их в качестве путей для результирующего класса и для файла данных фикстуры.

Сейчас в папку tests добавим модель Post, подпапку data в runtime и пустую заготовку тестового скрипта GeneratorTest:

generator

├── src

│   └── ...

├── tests

│   ├── runtime

│   │ ├── .gitignore

│   │   └── data

│   │   └── .gitignore

│   ├── bootstrap.php

│   ├── GeneratorTest.php

│   ├── Post.php

│   └── TestCase.php

└── ...

В модель добавим минимальное содержимое:

namespace elisdn\gii\fixture\tests;

use yii\db\ActiveRecord;

class Post extends ActiveRecord

{

public static function tableName()

{

return 'post';

}

}

Сейчас можно добавить тесты на валидацию, которые будут проверять, что генератор не принимает несуществующие модели и не позволяет вписывать несуществующие пути:

namespace elisdn\gii\fixture\tests;

use elisdn\gii\fixture\Generator as FixtureGenerator;

class GeneratorTest extends TestCase

{

public function testValidateIncorrect()

{

$generator = new FixtureGenerator();

$generator->modelClass = 'tests\Fake';

$generator->fixtureNs = 'tests\runtime';

$generator->dataPath = '@tests/runtime/fake';

$generator->grabData = true;

$this->assertFalse($generator->validate());

$this->assertEquals($generator->getFirstError('dataPath'), 'Path does not exist.');

$this->assertEquals($generator->getFirstError('modelClass'), 'Class \'tests\\Fake\' does not exist or has syntax error.');

}

public function testValidateCorrect()

{

$generator = new FixtureGenerator();

$generator->modelClass = 'elisdn\gii\fixture\tests\Post';

$generator->fixtureNs = 'tests\runtime';

$generator->dataPath = '@tests/runtime/data';

$generator->grabData = true;

$this->assertTrue($generator->validate(), 'Validation failed: ' . print\_r($generator->getErrors(), true));

}

}

Ещё можно добавить проверку на правильность получения имён сгенерированных файлов:

namespace elisdn\gii\fixture\tests;

use elisdn\gii\fixture\Generator as FixtureGenerator;

class GeneratorTest extends TestCase

{

...

public function testDefaultNames()

{

$generator = new FixtureGenerator();

$generator->modelClass = 'elisdn\gii\fixture\tests\Post';

$generator->fixtureNs = 'tests\runtime';

$generator->dataPath = '@tests/runtime/data';

$generator->grabData = false;

$this->assertEquals('PostFixture', $generator->getFixtureClassName());

$this->assertEquals('post.php', $generator->getDataFileName());

}

public function testSpecificNames()

{

$generator = new FixtureGenerator();

$generator->modelClass = 'elisdn\gii\fixture\tests\Post';

$generator->fixtureClass = 'PostCustomFixture';

$generator->fixtureNs = 'tests\runtime';

$generator->dataFile = 'post-custom.php';

$generator->dataPath = '@tests/runtime/data';

$generator->grabData = false;

$this->assertEquals('PostCustomFixture', $generator->getFixtureClassName());

$this->assertEquals('post-custom.php', $generator->getDataFileName());

}

}

И, что самое важное, добавить тесты на проверку самих получившихся файлов.

Чтобы не возиться с большими текстовыми фрагментами, добавим папку expected и создадим в ней проверочные образцы, с которыми будем сравнивать результаты генерации в тестах:

generator

├── src

│   └── ...

├── tests

│   ├── expected

│   │ ├── class.php

│   │   ├── data-empty.php

│   │   └── data-full.php

│   ├── runtime

│   │ ├── .gitignore

│   │   └── data

│   │   └── .gitignore

│   ├── bootstrap.php

│   ├── GeneratorTest.php

│   ├── Post.php

│   └── TestCase.php

└── ...

В tests/expected/class.php будет содержаться образцовый код класса фикстуры:

<?php

namespace tests\runtime;

use yii\test\ActiveFixture;

class PostFixture extends ActiveFixture

{

public $modelClass = 'elisdn\gii\fixture\tests\Post';

public $dataFile = '@tests/runtime/data/post.php';

}

В tests/expected/data-empty.php будет пример файла с пустыми данными:

<?php

return [

[

'id' => '',

'title' => '',

'content' => null,

'status' => '',

'created\_at' => '',

],

];

И в tests/expected/data-full.php будут вшиты существующие данные из базы:

<?php

return [

[

'id' => 1,

'title' => 'First Title',

'content' => null,

'status' => 0,

'created\_at' => 1459672035,

],

[

'id' => 2,

'title' => 'Second Title',

'content' => 'Second Content',

'status' => 1,

'created\_at' => 1459672036,

],

];

И в тестах теперь попробуем выполнить метод generate() и проверить отрендеренное содержимое вернувшихся из этого метода набора объектов CodeFile. При этом не забудем создать тестовую SQLite-базу в файле tests/runtime/sqlite.db и заполнить её этими же тестовыми данными:

namespace elisdn\gii\fixture\tests;

use elisdn\gii\fixture\Generator as FixtureGenerator;

use Yii;

use yii\db\Connection;

use yii\db\Schema;

use yii\gii\CodeFile;

class GeneratorTest extends TestCase

{

...

public function testGenerateWithoutData()

{

$this->initDb();

$generator = new FixtureGenerator();

$generator->modelClass = 'elisdn\gii\fixture\tests\Post';

$generator->fixtureNs = 'tests\runtime';

$generator->dataPath = '@tests/runtime/data';

$generator->grabData = false;

/\*\* @var CodeFile[] $files \*/

$this->assertCount(2, $files = $generator->generate());

$this->assertStringEqualsFile(\_\_DIR\_\_ . '/expected/class.php', $files[0]->content);

$this->assertStringEqualsFile(\_\_DIR\_\_ . '/expected/data-empty.php', $files[1]->content);

}

public function testGenerateWithData()

{

$this->initDb();

$generator = new FixtureGenerator();

$generator->modelClass = 'elisdn\gii\fixture\tests\Post';

$generator->fixtureNs = 'tests\runtime';

$generator->dataPath = '@tests/runtime/data';

$generator->grabData = true;

/\*\* @var CodeFile[] $files \*/

$this->assertCount(2, $files = $generator->generate());

$this->assertStringEqualsFile(\_\_DIR\_\_ . '/expected/class.php', $files[0]->content);

$this->assertStringEqualsFile(\_\_DIR\_\_ . '/expected/data-full.php', $files[1]->content);

}

private function initDb()

{

@unlink(\_\_DIR\_\_ . '/runtime/sqlite.db');

$db = new Connection([

'dsn' => 'sqlite:' . Yii::$app->getRuntimePath() . '/sqlite.db',

'charset' => 'utf8',

]);

Yii::$app->set('db', $db);

$db->createCommand()->createTable('post', [

'id' => Schema::TYPE\_PK,

'title' => Schema::TYPE\_STRING . '(255) NOT NULL',

'content' => Schema::TYPE\_TEXT,

'status' => Schema::TYPE\_SMALLINT . '(1) NOT NULL DEFAULT 1',

'created\_at' => Schema::TYPE\_INTEGER . '(11) NOT NULL'

])->execute();

$db->createCommand()->insert('post', [

'id' => 1,

'title' => 'First Title',

'content' => null,

'status' => 0,

'created\_at' => 1459672035

])->execute();

$db->createCommand()->insert('post', [

'id' => 2,

'title' => 'Second Title',

'content' => 'Second Content',

'status' => 1,

'created\_at' => 1459672036,

])->execute();

}

}

И теперь для тестов в папке самого расширения в консоли устанавливаем Gii, фреймворк и PHPUnit:

composer install

После установки всего в vendor запускаем наши тесты:

php vendor/bin/phpunit

И видим, что все шесть тестов прошли успешно:

PHPUnit 4.8.26 by Sebastian Bergmann and contributors.

......

Time: 1.86 seconds, Memory: 12.00MB

OK (6 tests, 18 assertions)

Теперь коммитим все недокоммиченное, пишем инструкцию по использованию в README.md и [публикуем на Packagist](http://www.elisdn.ru/blog/88/seo-service-on-yii2-composer-extensions) как и раньше.

После публикации переходим в любой свой проект, загружаем это расширение:

composer require --dev elisdn/yii2-gii-fixture-generator

и в конфигурационном файле приложения подключаем новый генератор к своему gii-модулю как в инструкции [на странице расширения](https://github.com/ElisDN/yii2-gii-fixture-generator), не забыв указать путь до папки через Yii::setAlias('@tests', ...):

Yii::setAlias('@tests', dirname(\_\_DIR\_\_) . '/tests');

$config = [

...

];

if (YII\_ENV\_DEV) {

$config['bootstrap'][] = 'debug';

$config['modules']['debug'] = [

'class' => 'yii\debug\Module',

];

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

$config['modules']['gii'] = [

'class' => 'yii\gii\Module',

'generators' => [

'fixture' => [

'class' => 'elisdn\gii\fixture\Generator',

],

],

];

}

return $config;

И используем у себя:

Если модуль Gii подключен и в консольной конфигурации, то можно использовать эти же возможности в консоли:

php yii gii/fixture --modelClass=app\\models\\Post --grabData=1

На этом подключение завершено. Можно спокойно генерировать десятки и сотни фикстур для всех своих моделей и таблиц без возни с ручным копированием классов и полей. И, что особенно приятно, можно за пару минут воссоздать в фикстурах полную копию данных из уже существующих заполненных таблиц.