Yii 2.0 Cookbook

by Yii community
and Alexander Makarov

Contents

0.1 Preface

Yii is a high-performance PHP framework designed from the ground up to develop modern web applications and APIs that can run in a multi-device environment.

Yii comes with rich set of features including MVC, ActiveRecord, I18N/L10N, caching, authentication and role-based access control, code generation, testing, and REST based API framework. Together with a comprehensive set of documentation and a enthusiastic user community Yii can reduce your development time significantly compared with other frameworks.

0.1.1 What's the book about?

This book is for you if you're familiar with Yii 2.0 and building Yii applications plus familiar with the official Yii guide. It covers fundamentally important Yii concepts and dives into subjects that will help you develop even faster, more reliable Yii applications.

The book consists of individual recipes gathered from Yii experts that you can apply in your applications. Recipes are grouped by chapter but you are free to read them in any order as there is no dependency between them.

0.1.2 Prerequisites

- You should have Yii 2.0 installed[1].
- You should be familiar with the framework basics and the official guide[2].

0.1.3 How to participate

If you've found any errata, incorrect information, know how to improve something or have a good recipe for the book, either create an issue or make a pull request in the book github repository[3].

[1] http://www.yiiframework.com/doc-2.0/guide-start-installation.html
[2] http://www.yiiframework.com/doc-2.0/guide-README.html
[3] https://github.com/samdark/yii2-cookbook

Chapter 1

Unnoticed basics

Chapter 2

Logging and error handling

2.1 Logging: problems and solutions

Logging in Yii is really flexible. Basics are easy but sometimes it takes time to configure everything to get what you want. There are some ready to use solutions collected below. Hopefully you'll find what you're looking for.

2.1.1 Write 404 to file and send the rest via email

404 not found happens too often to email about it. Still, having 404s logged to a file could be useful. Let's implement it.

```
'components' => [
    'log' => [
        'targets' => [
            'file' => [
                'class' => 'yii\log\FileTarget',
                'categories' => ['yii\web\HttpException:404'],
                'levels' => ['error', 'warning'],
                'logFile' => '@runtime/logs/404.log',
            ],
            'email' => [
                'class' => 'yii\log\EmailTarget',
                'except' => ['yii\web\HttpException:404'],
                'levels' => ['error', 'warning'],
                'message' => ['from' => 'robot@example.com', 'to' => '
    admin@example.com'],
            ],
        ],
    ],
],
```

When there's unhandled exception in the application Yii logs it additionally to displaying it to end user or showing customized error page. Exception message is what actually to be written and the fully qualified exception class name is the category we can use to filter messages when configuring targets. 404 can be triggered by throwing `yii\web\NotFoundHttpException` or

automatically. In both cases exception class is the same and is inherited from `\yii\web\HttpException` which is a bit special in regards to logging. The speciality is the fact that HTTP status code prepended by : is appended to the end of the log message category. In the above we're using `categories` to include and `except` to exclude 404 log messages.

2.1.2 Immediate logging

By default Yii accumulates logs till the script is finished or till the number of logs accumulated is enough which is 1000 messages by default for both logger itself and log target. It could be that you want to log messages immediately. For example, when running an import job and checking logs to see the progress. In this case you need to change settings via application config file:

```
'components' => [
    'log' => [
        'flushInterval' => 1, // <-- here
        'targets' => [
            'file' => [
                'class' => 'yii\log\FileTarget',
                'levels' => ['error', 'warning'],
                'exportInterval' => 1, // <-- and here
            ],
        ],
    ]
]
```

2.1.3 Write different logs to different files

Usually a program has a lot of functions. Sometimes it is necessary to control these functions by logging. If everything is logged in one file this file becomes too big and too difficult to maintain. Good solution is to write different functions logs to different files.

For example you have two functions: catalog and basket. Let's write logs to catalog.log and basket.log respectively. In this case you need to establish categories for your log messages. Make a connection between them and log targets by changing application config file:

```
'components' => [
    'log' => [
        'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'categories' => ['catalog'],
                    'logFile' => '@app/runtime/logs/catalog.log',
                ],
                [
                    'class' => 'yii\log\FileTarget',
```

```
                    'categories' => ['basket'],
                    'logFile' => '@app/runtime/logs/basket.log',
                ],
        ],
    ]
]
```

After this you are able to write logs to separate files adding category name to log function as second parameter. Examples:

```
\Yii::info('catalog info', 'catalog');
\Yii::error('basket error', 'basket');
\Yii::beginProfile('add to basket', 'basket');
\Yii::endProfile('add to basket', 'basket');
```

2.1.4 Disable logging of youself actions

Problem

You want to receive email when new user signs up. But you don't want to trace yourself sign up tests.

Solution

At first mark logging target inside `config.php` by key: ‘php ‘log’ => [

```
// ...
'targets' => [
        // email is a key for our target
        'email' => [
            'class' => 'yii\log\EmailTarget',
            'levels' => ['info'],
            'message' => ['from' => 'robot@example.com', 'to' => '
    admin@example.com'],
        ],
// ...
```

```
Then, for example, inside ‘Controller‘ ‘beforeAction‘ you can create a
    condition:
‘‘‘php
    public function beforeAction($action)
    {
        // '127.0.0.1' - replace by your IP address
        if (in_array(@$_SERVER['REMOTE_ADDR'], ['127.0.0.1'])) {
            Yii::$app->log->targets['email']->enabled = false; // Here we
    disable our log target
        }
        return parent::beforeAction($action);
    }
```

2.1.5 Log everything but display different error messages

Problem

You want to log concrete server error but display only broad error explanation to the end user.

Solution

If you catch an error appropriate log target doesn't work. Let's say you have such log target configuration: ‘php ‘components’ => [

```
'log' => [
    'targets' => [
        'file' => [
            'class' => 'yii\log\FileTarget',
            'levels' => ['error', 'warning'],
            'logFile' => '@runtime/logs/error.log',
        ],
    ],
],
```

], ‘

As an example let's add such code line inside `actionIndex`: ‘php

```
public function actionIndex()
{
    throw new ServerErrorHttpException('Hey! Coding problems!');
    // ...
```

```
Go to ‘index‘ page and you will see this message in the browser and in the ‘
    error.log‘ file.

Let's modify our ‘actionIndex‘:

‘‘‘php
    public function actionIndex()
    {
        try {
            throw new ServerErrorHttpException('Hey! Coding problems!'); //
    here is our code line now
        }
        catch(ServerErrorHttpException $ex) {
            Yii::error($ex->getMessage()); // concrete message for us
            throw new ServerErrorHttpException('Server problem, sorry.'); //
     broad message for the end user
        }
    // ..
```

As the result in the browser you will see `Server problem, sorry.`. But in the `error.log` you will see **both** error messages. In our case second message is not necessary to log.

Let's add `category` for our log target and for logging command.

For `config`: ‘php ‘file’ => [

```
'class' => 'yii\log\FileTarget',
'levels' => ['error', 'warning'],
'categories' => ['serverError'], // category is added
'logFile' => '@runtime/logs/error.log',
```

], For ‘`actionIndex`‘:php catch(ServerErrorHttpException $ex) {

```
Yii::error($ex->getMessage(), 'serverError'); // category is added
throw new ServerErrorHttpException('Server problem, sorry.');
```

} ‘

As the result in the `error.log` you will see only the error related to `Hey! Coding problems!`.

Even more

If there is an bad request (user side) error you may want to display error message ‘as is’. You can easily do it because our catch block works only for `ServerErrorHttpException` error types. So you are able to throw something like this: ‘php throw new BadRequestHttpException('Email address you provide is invalid'); ‘ As the result end user will see the message ‘as is’ in his browser.

2.1.6 See also

- Yii2 guide - handling Errors[1].
- Yii2 guide - logging[2].

[1] http://www.yiiframework.com/doc-2.0/guide-runtime-handling-errors.html
[2] http://www.yiiframework.com/doc-2.0/guide-runtime-logging.html

Chapter 3

Web essentials

3.1 URLs with variable number of parameters

There are many cases when you need to get variable number of parameters via URL. For example one may want URLs such as `http://example.com/products/cars/sport` to lead to `ProductController::actionCategory` where it's expected to get an array containing `cars` and `sport`.

3.1.1 Get Ready

First of all, we need to enable pretty URLs. In the application config file add the following:

```
$config = [
    // ...
    'components' => [
        // ...
        'urlManager' => [
            'showScriptName' => false,
            'enablePrettyUrl' => true,
            'rules' => require 'urls.php',
        ],
    ]
```

Note that we're including separate file instead of listing rules directly. It is helpful when application grows large.

Now in `config/urls.php` add the following content:

```
<?php
return [
    [
        'pattern' => 'products/<categories:.*>',
        'route' => 'product/category',
        'encodeParams' => false,
    ],
];
```

Create `ProductController`:

```
namespace app\controllers;

use yii\web\Controller;

class ProductController extends Controller
{
    public function actionCategory($categories)
    {
        $params = explode('/', $categories);
        print_r($params);
    }
}
```

That's it. Now you can try `http://example.com/products/cars/sport`. What you'll get is

```
Array ( [0] => cars [1] => sport)
```

3.2 Working with different response types

Web and mobile applications are more than just rendered HTML nowadays. Modern architecture moves the UI to the client, where all user interactions are handled by the client-side, utilizing server APIs to drive the frontend. The JSON and XML formats are often used for serializing and transmitting structured data over a network, so the ability to create such responses is a must for any modern server framework.

3.2.1 Response formats

As you probably know, in Yii2 you need to `return` the result from your action, instead of echoing it directly:

```
// returning HTML result
return $this->render('index', [
    'items' => $items,
]);
```

Good thing about it is now you can return different types of data from your action, namely:

- an array
- an object implementing `Arrayable` interface
- a string
- an object implementing `__toString()` method.

Just don't forget to tell Yii what format do you want as result, by setting `\Yii::$app->response->format` before `return`. For example: ‘php \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON; ‘

Valid formats are:

- FORMAT_RAW
- FORMAT_HTML

- FORMAT_JSON
- FORMAT_JSONP
- FORMAT_XML

Default is `FORMAT_HTML`.

3.2.2 JSON response

Let's return an array:

```
public function actionIndex()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    $items = ['some', 'array', 'of', 'data' => ['associative', 'array']];
    return $items;
}
```

And - voila! - we have JSON response right out of the box:

Result

```
{
    "0": "some",
    "1": "array",
    "2": "of",
    "data": ["associative", "array"]
}
```

Note: you'll get an exception if response format is not set.

As we already know, we can return objects too.

```
public function actionView($id)
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    $user = \app\models\User::find($id);
    return $user;
}
```

Now $user is an instance of `ActiveRecord` class that implements `Arrayable` interface, so it can be easily converted to JSON:

Result ‘json {

```
"id": 1,
"name": "John Doe",
"email": "john@example.com"
```

} ‘

We can even return an array of objects:

```
public function actionIndex()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    $users = \app\models\User::find()->all();
    return $users;
}
```

Now `$users` is an array of ActiveRecord objects, but under the hood Yii uses `\yii\helpers\Json::encode()` that traverses and converts the passed data, taking care of types by itself:

Result

```
[
    {
        "id": 1,
        "name": "John Doe",
        "email": "john@example.com"
    },
    {
        "id": 2,
        "name": "Jane Foo",
        "email": "jane@example.com"
    },
    ...
]
```

3.2.3 XML response

Just change response format to `FORMAT_XML` and that't it. Now you have XML:

```
public function actionIndex()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_XML;
    $items = ['some', 'array', 'of', 'data' => ['associative', 'array']];
    return $items;
}
```

Result

```
<response>
    <item>some</item>
    <item>array</item>
    <item>of</item>
    <data>
        <item>associative</item>
        <item>array</item>
    </data>
</response>
```

And yes, we can convert objects and array of objects the same way as we did before.

```
public function actionIndex()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_XML;
    $users = \app\models\User::find()->all();
    return $users;
}
```

Result:

```
<response>
    <User>
        <id>1</id>
        <name>John Doe</name>
        <email>john@example.com</email>
    </User>
    <User>
        <id>2</id>
        <name>Jane Foo</name>
        <email>jane@example.com</email>
    </User>
</response>
```

3.2.4 Custom response format

Let's create a custom response format. To make example a bit fun and crazy we'll respond with PHP arrays.

First of all, we need formatter itself. Create `components/PhpArrayFormatter.php`:

```
<?php
namespace app\components;

use yii\helpers\VarDumper;
use yii\web\ResponseFormatterInterface;

class PhpArrayFormatter implements ResponseFormatterInterface
{
    public function format($response)
    {
        $response->getHeaders()->set('Content-Type', 'text/php; charset=UTF-
    8');
        if ($response->data !== null) {
            $response->content = "<?php\nreturn " . VarDumper::export(
    $response->data) . ";\n";
        }
    }
}
```

Now we need to registed it in application config (usually it's `config/web.php`):

```
return [
    // ...
    'components' => [
        // ...
        'response' => [
            'formatters' => [
                'php' => 'app\components\PhpArrayFormatter',
            ],
        ],
    ],
];
```

Now it's ready to be used. In `controllers/SiteController` create a new method `actionTest`:

```
public function actionTest()
{
    Yii::$app->response->format = 'php';
    return [
        'hello' => 'world!',
    ];
}
```

That's it. After executing it, Yii will respond with the following:

```
<?php
return [
    'hello' => 'world!',
];
```

3.2.5 Choosing format based on content type requested

You can use the `ContentNegotiator` controller filter in order to choose format based on what is requested. In order to do so you need to implement `behaviors` method in controller:

```
public function behaviors()
{
    return [
        // ...
        'contentNegotiator' => [
            'class' => \yii\filters\ContentNegotiator::className(),
            'only' => ['index', 'view'],
            'formatParam' => '_format',
            'formats' => [
                'application/json' => \yii\web\Response::FORMAT_JSON,
                'application/xml' => \yii\web\Response::FORMAT_XML,
            ],
        ],
    ];
}

public function actionIndex()
{
    $users = \app\models\User::find()->all();
    return $users;
}

public function actionView($id)
{
    $user = \app\models\User::findOne($id);
    return $user;
}
```

That's it. Now you can test it via the following URLs:

- `/index.php?r=user/index&_format=xml`

- `/index.php?r=user/index&_format=json`

3.3 Managing cookies

Managing HTTP cookies isn't that hard using plain PHP but Yii makes it a bit more convenient. In this recipe we'll describe how to perform typical cookie actions.

3.3.1 Setting a cookie

To set a cookie i.e. to create it and schedule for sending to the browser you need to create new `\yii\web\Cookie` class instance and add it to response cookies collection:

```
$cookie = new Cookie([
    'name' => 'cookie_monster',
    'value' => 'Me want cookie!',
    'expire' => time() + 86400 * 365,
]);
\Yii::$app->getResponse()->getCookies()->add($cookie);
```

In the above we're passing parameters to cookie class constructor. These basically the same as used with native PHP setcookie[1] function:

- `name` - name of the cookie.
- `value` - value of the cookie. Make sure it's a string. Browsers typically aren't happy about binary data in cookies.
- `domain` - domain you're setting the cookie for.
- `expire` - unix timestamp indicating time when the cookie should be automatically deleted.
- `path` - the path on the server in which the cookie will be available on.
- `secure` - if `true`, cookie will be set only if HTTPS is used.
- `httpOnly` - if `true`, cookie will not be available via JavaScript.

3.3.2 Reading a cookie

In order to read a cookie use the following code:

```
$value = \Yii::$app->getRequest()->getCookies()->getValue('my_cookie');
```

3.3.3 Where to get and set cookies?

Cookies are part of HTTP request so it's a good idea to do both in controller which responsibility is exactly dealing with request and response.

[1] http://php.net/manual/en/function.setcookie.php

3.3.4 Cookies for subdomains

Because of security reasons, by default cookies are accessible only on the same domain from which they were set. For example, if you have set a cookie on domain `example.com`, you cannot get it on domain `www.example.com`. So if you're planning to use subdomains (i.e. admin.example.com, profile.example.com), you need to set `domain` explicitly:

```
$cookie = new Cookie([
  'name' => 'cookie_monster',
  'value' => 'Me want cookie everywhere!',
  'expire' => time() + 86400 * 365,
  'domain' => '.example.com' // <<<=== HERE
]);
\Yii::$app->getResponse()->getCookies()->add($cookie);
```

Now cookie can be read from all subdomains of `example.com`.

3.3.5 Cross-subdomain authentication and identity cookies

In case of autologin or "remember me" cookie, the same quirks as in case of subdomain cookies are applying. But this time you need to configure user component, setting `identityCookie` array to desired cookie config.

Open you application config file and add `identityCookie` parameters to user component configuration:

```
$config = [
    // ...
    'components' => [
        // ...
        'user' => [
            'class' => 'yii\web\User',
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
            'loginUrl' => '/user/login',
            'identityCookie' => [ // <---- here!
                'name' => '_identity',
                'httpOnly' => true,
                'domain' => '.example.com',
            ],
        ],
        'request' => [
            'cookieValidationKey' => 'your_validation_key'
        ],
        'session' => [
            'cookieParams' => [
                'domain' => '.example.com',
                'httpOnly' => true,
            ],
        ],

    ],
];
```

Note that `cookieValidationKey` should be the same for all sub-domains.

Note that you have to configure the `session::cookieParams` property to have the samedomain as your `user::identityCookie` to ensure the `login` and `logout` work for all subdomains. This behavior is better explained on the next section.

3.3.6 Session cookie parameters

Session cookies parameters are important both if you have a need to maintain session while getting from one subdomain to another or when, in contrary, you host backend app under `/admin` URL and want handle session separately.

```
$config = [
    // ...
    'components' => [
        // ...
        'session' => [
            'name' => 'admin_session',
            'cookieParams' => [
                'httpOnly' => true,
                'path' => '/admin',
            ],
        ],
    ],
];
```

3.3.7 See also

- API reference[2]
- PHP documentation[3]
- RFC 6265[4]

3.4 Handling incoming third party POST requests

By default Yii uses CSRF protection that verifies that POST requests could be made only by the same application. It enhances overall security significantly but there are cases when CSRF should be disabled i.e. when you expect incoming POST requests from a third party service.

Additionally, if third party is posting via XMLHttpRequest (browser AJAX), we need to send additional headers to allow CORS (cross-origin resource sharing[5]).

[2] http://stuff.cebe.cc/yii2docs/yii-web-cookie.html
[3] http://php.net/manual/en/function.setcookie.php
[4] http://www.faqs.org/rfcs/rfc6265.html
[5] https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

3.4.1 How to do it

First of all, never disable CSRF protection altogether. If you need it to be disabled, do it for specific controller or even controller action.

Disabling CSRF for a specific controller

Disabling protection for specific controller is easy:

```
class MyController extends Controller
{
    public $enableCsrfValidation = false;
```

We've added a public property `enableCsrfValidation` and set it to false.

Disabling CSRF for specific controller action

In case of disabling only a single action it's a bit more code:

```
class MyController extends Controller
{
    public function beforeAction($action)
    {
        if (in_array($action->id, ['incoming'])) {
            $this->enableCsrfValidation = false;
        }
        return parent::beforeAction($action);
    }
```

We've implemented `beforeAction` controller method. It is invoked right before an action is executed so we're checking if executed action id matches id of the action we want to disable CSRF protection for and, if it's true, disabling it. Note that it's important to call parent method and call it last.

Sending CORS headers

Yii has a special Cors filter[6] that allows you sending headers required to allow CORS.

To allow AJAX requests to the whole controller you can use it like that:

```
class MyController extends Controller
{
    public function behaviors()
    {
        return [
            'corsFilter' => [
                'class' => \yii\filters\Cors::className(),
            ],
        ];
    }
```

[6] http://www.yiiframework.com/doc-2.0/yii-filters-cors.html

In order to do it for a specific action, use the following:

```
class MyController extends Controller
{
    public function behaviors()
    {
        return [
            'corsFilter' => [
                'class' => \yii\filters\Cors::className(),
                'cors' => [],
                'actions' => [
                    'incoming' => [
                        'Origin' => ['*'],
                        'Access-Control-Request-Method' => ['GET', 'POST', '
    PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS'],
                        'Access-Control-Request-Headers' => ['*'],
                        'Access-Control-Allow-Credentials' => null,
                        'Access-Control-Max-Age' => 86400,
                        'Access-Control-Expose-Headers' => [],
                    ],
                ],
            ],
        ];
    }
```

Chapter 4

SEO essentials

4.1 Enable pretty URLs

Sometimes users want to share your site URLs via social networks. For example, by default your *about page* URL looks like `http://webproject.ru/index.php?r=site%2Fabout`. Let's imagine this link on Facebook page. Do you want to click on it? Most of users have no idea what is `index.php` and what is `%2`. They trust such link less, so will click less on it. Thus web site owner would lose traffic.

URLs such as the following is better: `http://webproject.ru/about`. Every user can understand that it is a clear way to get to *about page*.

Let's enable pretty URLs for our Yii project.

4.1.1 Apache Web server configuration

If you're using Apache you need an extra step. Inside your `.htaccess` file in your webroot directory or inside location section of your main Apache config add the following lines:

```
RewriteEngine on
# If a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
# Otherwise forward it to index.php
RewriteRule . index.php
```

4.1.2 URL manager configuration

Configure `urlManager` component in your Yii config file:

```
'components' => [
    // ...
    'urlManager' => [
        'class' => 'yii\web\UrlManager',
        // Hide index.php
```

```
        'showScriptName' => false,
        // Use pretty URLs
        'enablePrettyUrl' => true,
        'rules' => [
        ],
    ],
    // ...
],
```

Remove site parameter from URL

After previous steps you will get `http://webproject.ru/site/about` link. `site` parameter tells nothing helpful to your users. So remove it by additional `urlManager` rule:

```
    'rules' => [
        '<alias:\w+>' => 'site/<alias>',
    ],
```

As a result your URL will looks like `http://webproject.ru/about`.

4.2 Pagination pretty URLs

For example we can render our site content by GridView. If there are a lot of content rows we use pagination. And it is necessary to provide GET request for every pagination page. Thus search crawlers can index our content. We also need our URLs to be pretty. Let's do it.

4.2.1 Initial state

For example we have a page with such URL `http://example.com/schools/schoolTitle`. Parameter `schoolTitle` is a `title` parameter for our request. For details see this recipe[1].

In the application config file we have: ‘php $config = [

```
// ...
'components' => [
    // ...
    'urlManager' => [
        'showScriptName' => false,
        'enablePrettyUrl' => true,
        'rules' => [
          'schools/<title:\w+>' => 'site/schools',
        ],
    ],
    // ...
],
// ...
```

[1] https://github.com/samdark/yii2-cookbook/blob/master/book/urls-variable-number-of-parameters.md

```
We decided to add a GridView on the page ‘http://example.com/schools/
    schoolTitle‘.

Pagination pretty URLs
---------
When we click on pagination link our URL is transformed to ‘http://example.
    com/schools/schoolTitle?page=2‘.
We want our pagination link looks like ‘http://example.com/schools/
    schoolTitle/2‘.

Let’s add new urlManager rule **higher than** existed rule. Here it is:
‘‘‘php
$config = [
        // ...
        ’urlManager’ => [
            // ...
            ’rules’ => [
              ’schools/<title:\w+>/<page:\d+>’ => ’site/schools’, // new
    rule
              ’schools/<title:\w+>’ => ’site/schools’,
            ],
        ],
        // ...
```

4.3 Adding SEO tags

Organic search is an excellent traffic source. In order to get it you have to make a lot of small steps to improve your project.

One of such steps is to provide different meta tags for different pages. It will improve your site organic search appearance and may result in better ranking.

Let’s review how to add SEO-related metadata to your pages.

4.3.1 Title

It is very simple to set title. Inside controller action:

```
\Yii::$app->view->title = ’title set inside controller’;
```

Inside a view:

```
$this->title = ’Title from view’;
```

> **Note:** Setting `$this->title` in layout will override value which is set for concrete view so don’t do it.

It’s a good idea to have default title so inside layout you can have something like the following:

```
$this->title = $this->title ? $this->title : ’default title’;
```

4.3.2 Description and Keywords

There are no dedicated view parameters for `keywords` or `description`. Since these are meta tags and you should set them by `registerMetaTag()` method.

Inside controller action:

```
\Yii::$app->view->registerMetaTag([
    'name' => 'description',
    'content' => 'Description set inside controller',
]);
\Yii::$app->view->registerMetaTag([
    'name' => 'keywords',
    'content' => 'Keywords set inside controller',
]);
```

Inside a view:

```
$this->registerMetaTag([
    'name' => 'description',
    'content' => 'Description set inside view',
]);
$this->registerMetaTag([
    'name' => 'keywords',
    'content' => 'Keywords set inside view',
]);
```

All registered meta tags will be rendered inside layout in place of `$this->head()` call.

Note that when the same tag is registered twice it's rendered twice. For example, description meta tag that is registered both in layout and a view is rendered twice. Usually it's not good for SEO. In order to avoid it you can specify key as the second argument of `registerMetaTag`:

```
$this->registerMetaTag([
    'name' => 'description',
    'content' => 'Description 1',
], 'description');

$this->registerMetaTag([
    'name' => 'description',
    'content' => 'Description 2',
], 'description');
```

In this case later second call will overwrite first call and description will be set to "Description 2".

4.4 Canonical URLs

Because of many reasons the same or nearly the same page content often is accessible via multiple URLs. There are valid cases for it such as viewing an article within a category and not so valid ones. For end user it doesn't really matter much but still it could be a problem because of search engines

because either you might get wrong URLs preferred or, in the worst case, you might get penalized.

One way to solve it is to mark one of URLs as a primary or, as it called, canonical, one you may use `<link rel="canonical"` tag in the page head.

> **Note:** In the above we assume that pretty URLs are enabled.

Let's imagine we have two pages with similar or nearly similar content:

- `http://example.com/item1`
- `http://example.com/item2`

Our goal is to mark first one as canonical. Another one would be still accessible to end user. The process of adding SEO meta-tags is descibed in "adding SEO tags" recipe. Adding `<link rel="canonical"` is very similar. In order to do it from controller action you may use the following code:

```
\Yii::$app->view->registerLinkTag(['rel' => 'canonical', 'href' => Url::to([
    'item1'], true)]);
```

In order to achieve the same from inside the view do it as follows:

```
$this->registerLinkTag(['rel' => 'canonical', 'href' => Url::to(['item1'],
    true)]);
```

> **Note:** It is necessary to use absolute paths instead of relative ones.

As an alternative to `Url::to()` you can use `Url::canonical()` such as

```
$this->registerLinkTag(['rel' => 'canonical', 'href' => Url::canonical()]);
```

The line above could be added to layout. `Url::canonical()` generates the tag based on current controller route and action parameters (the ones present in the method signature).

4.4.1 See also

- Google article about canonical URLs[2].

4.5 Using redirects

4.5.1 301

Let's imagine we had a page `http://example.com/item2` but then permanently moved content to `http://example.com/item1`. There is a good chance that some users (or search crawlers) have already saved `http://example.com/item2` via bookmarks, database, web site article, etc. Because of that we can't just remove `http://webproject.ru/item2`.

In this case use 301 redirect.

[2] `https://support.google.com/webmasters/answer/139066?hl=en`

```
class MyController extends Controller
{
    public function beforeAction($action)
    {
        if (in_array($action->id, ['item2'])) {
            Yii::$app->response->redirect(Url::to(['item1']), 301);
            Yii::$app->end();
        }
        return parent::beforeAction($action);
    }
```

For further convenience you can determine an array. So if you need to redirect another URL then add new key=>value pair:

```
class MyController extends Controller
{
    public function beforeAction($action)
    {
        $toRedir = [
            'item2' => 'item1',
            'item3' => 'item1',
        ];

        if (isset($toRedir[$action->id])) {
            Yii::$app->response->redirect(Url::to([$toRedir[$action->id]]),
    301);
            Yii::$app->end();
        }
        return parent::beforeAction($action);
    }
```

4.5.2 See also

- Handling trailing slash in URLs.

4.6 Using slugs

Even when pretty URLs are enabled, these often aren't looking too friendly:

```
http://example.com/post/42
```

Using Yii it doesn't take much time to make URLs look like the following:

```
http://example.com/post/hello-world
```

4.6.1 Preparations

Set up database to use with Yii, create the following table:

```
post
====

id
```

```
title
content
```

Generate `Post` model and CRUD for it using Gii.

4.6.2 How to do it

Add `slug` field to `post` table that holds our posts. Then add sluggable behavior to the model:

```
<?php
use yii\behaviors\SluggableBehavior;

// ...

class Post extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => SluggableBehavior::className(),
                'attribute' => 'title',
            ],
        ];
    }

    // ...
}
```

Now when post is created `slug` in database will be automatically filled.

We need to adjust controller. Add the following method:

```
protected function findModelBySlug($slug)
{
    if (($model = Post::findOne(['slug' => $slug])) !== null) {
        return $model;
    } else {
        throw new NotFoundHttpException();
    }
}
```

Now adjust `view` action:

```
public function actionView($slug)
{
    return $this->render('view', [
        'model' => $this->findModelBySlug($slug),
    ]);
}
```

Now in order to create a link to the post you have to pass slug into it like the following:

```
echo Url::to(['post/view', 'slug' => $post->slug]);
```

4.6.3 Handling title changes

There are be multiple strategies to deal with situation when title is changed. One of these is to include ID in the title and use it to find a post.

```
http://example.com/post/42/hello-world
```

4.7 Handling trailing slash in URLs

By default Yii handles URLs without trailing slash and gives out 404 for URLs with it. It is a good idea to choose either using or not using slash but handling both and doing 301 redirect from one variant to another.

For example,

```
/hello/world - 200
/hello/world/ - 301 redirect to /hello/world
```

4.7.1 Using UrlNormalizer

Since Yii 2.0.10 there's `UrlNormalizer` class you can use to deal with slash and no-slash URLs in a very convenient way. Check out the "URL normalization[3]" section in the official guide for details.

4.7.2 Redirecting via web server config

Besides PHP, there's a way to achieve redirection using web server.

4.7.3 Redirecting via nginx

Redirecting to no-slash URLs.

```
location / {
    rewrite ^(.*)/$ $1 permanent;
    try_files $uri $uri/ /index.php?$args;
}
```

Redirecting to slash URLs.

```
location / {
    rewrite ^(.*[^/])$ $1/ permanent;
    try_files $uri $uri/ /index.php?$args;
}
```

[3] http://www.yiiframework.com/doc-2.0/guide-runtime-routing.html#url-normalization

4.7.4 Redirecting via Apache

Redirecting to no-slash URLs.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)/$ /$1 [L,R=301]
```

Redirecting to slash URLs.

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*[^/])$ /$1/ [L,R=301]
```

Chapter 5

Forms

5.1 Using and customizing CAPTCHA

Accodring to Wikipedia[1] CAPTCHA means "Completely Automated Public Turing test to tell Computers and Humans Apart". In other words, CAPTCHA provides a problem human can solve easily but computer can't. The purpose of it is to prevent automated abuse such as posting comments containing links to malicious websites or voting for a particular candidate in an election.

Typical problem that is still quite tricky for computer algorithms is image recognition. That's why a common CAPTCHA shows an image with some text user should read and enter into input field.

5.1.1 How add CAPTCHA to a form

Yii provides a set of ready to use classes to add CAPTCHA to any form. Let's review how it could be done.

First of all, we need an action that will display an image containing text. Typical place for it is `SiteController`. Since there's ready to use action, it could be added via `actions()` method:

```
class SiteController extends Controller
{
    // ...
    public function actions()
    {
        return [
            // ...
            'captcha' => [
                'class' => 'yii\captcha\CaptchaAction',
                'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
            ],
        ];
```

[1] http://en.wikipedia.org/wiki/Captcha

```
    }

    // ...
}
```

In the above we're reusing `yii\captcha\CaptchaAction` as `site/captcha` route. `fixedVerifyCode` is set for test environment in order for the test to know which answer is correct.

Now in the form model (it could be either ActiveRecord or Model) we need to add a property that will contain user input for verification code and validation rule for it:

```
class ContactForm extends Model
{
    // ...
    public $verifyCode;

    // ...
    public function rules()
    {
        return [
            // ...
            ['verifyCode', 'captcha'],
        ];
    }

    // ...
}
```

Now we can actually display image and verification input box in a view containing a form:

```
<?php $form = ActiveForm::begin(['id' => 'contact-form']); ?>
    // ...
    <?= $form->field($model, 'verifyCode')->widget(Captcha::className()) ?>
    // ...
<?php ActiveForm::end(); ?>
```

That's it. Now robots won't pass. At least dumb ones.

If the image is not being displayed a good way to test if captcha requirements are installed is by accessing the captcha action directly. So for example if you are using a controller called site try typing in "http://blah.com/index.php/site/captcha" which should display an image. If not then turn on tracing and check for errors.

5.1.2 Simple math captcha

Nowadays CAPTCHA robots are relatively good at parsing image so while by using typical CAPTCHA you're significanly lowering number of spammy actions, some robots will still be able to parse the image and enter the verification code correctly.

In order to prevent it we have to increase the challenge. We could add extra ripple and special effects to the letters on the image but while it could make it harder for computer, it certainly will make it significantly harder for humans which isn't really what we want.

A good solution for it is to mix a custom task into the challenge. Example of such task could be a simple math question such as "2 + 1 = ?". Of course, the more unique this question is, the more secure is the CAPTCHA.

Let's try implementing it. Yii CAPTCHA is really easy to extend. The component itself doesn't need to be touched since both code generation, code verification and image generation happens in `CaptchaAction` which is used in a controller. In basic project template it's used in `SiteController`.

So, first of all, create `components/MathCaptchaAction.php`:

```
<?php
namespace app\components;

use yii\captcha\CaptchaAction;

class MathCaptchaAction extends CaptchaAction
{
    public $minLength = 0;
    public $maxLength = 100;

    /**
     * @inheritdoc
     */
    protected function generateVerifyCode()
    {
        return mt_rand((int)$this->minLength, (int)$this->maxLength);
    }

    /**
     * @inheritdoc
     */
    protected function renderImage($code)
    {
        return parent::renderImage($this->getText($code));
    }

    protected function getText($code)
    {
        $code = (int)$code;
        $rand = mt_rand(min(1, $code - 1), max(1, $code - 1));
        $operation = mt_rand(0, 1);
        if ($operation === 1) {
            return $code - $rand . '+' . $rand;
        } else {
            return $code + $rand . '-' . $rand;
        }
    }
}
```

In the code above we've adjusted code generation to be random number from 0 to 100. During image rendering we're generating simple math expression based on the current code.

Now what's left is to change default captcha action class name to our class name in `controllers/SiteController.php`, `actions()` method:

```
public function actions()
{
    return [
        // ...
        'captcha' => [
            'class' => 'app\components\MathCaptchaAction',
            'fixedVerifyCode' => YII_ENV_TEST ? '42' : null,
        ],
    ];
}
```

5.2 Working with ActiveForm via JavaScript

PHP side of ActiveForm, which is usually more than enough for majority of projects, is described well in the official Yii 2.0 guide[2]. It is getting a bit more tricky when it comes to advanced things such as adding or removing form fields dynamically or triggering individual field validation using unusual conditions.

In this recipe you'll be introduced to ActiveForm JavaScript API.

5.2.1 Preparations

We're going to use basic project template contact form for trying things out so install it first[3].

5.2.2 Triggering validation for individual form fields

```
$('#contact-form').yiiActiveForm('validateAttribute', 'contactform-name');
```

5.2.3 Trigger validation for the whole form

```
$('#contact-form').yiiActiveForm('validate', true);
```

The second passed argument `true` forces validation of the whole form.

5.2.4 Using events

```
$('#contact-form').on('beforeSubmit', function (e) {
  if (!confirm("Everything is correct. Submit?")) {
    return false;
  }
```

[2] http://www.yiiframework.com/doc-2.0/guide-input-forms.html
[3] http://www.yiiframework.com/doc-2.0/guide-start-installation.html

```
  return true;
});
```

Available events are:

- `beforeValidate`[4].
- `afterValidate`[5].
- `beforeValidateAttribute`[6].
- `afterValidateAttribute`[7].
- `beforeSubmit`[8].
- `ajaxBeforeSend`[9].
- `ajaxComplete`[10].

5.2.5 Adding and removing fields dynamically

To add a field to validation list:

```
$('#contact-form').yiiActiveForm('add', {
    id: 'address',
    name: 'address',
    container: '.field-address',
    input: '#address',
    error: '.help-block',
    validate:  function (attribute, value, messages, deferred, $form) {
        yii.validation.required(value, messages, {message: "Validation
    Message Here"});
    }
});
```

To remove a field so it's not validated:

```
$('#contact-form').yiiActiveForm('remove', 'address');
```

5.2.6 Updating error of a single attribute

In order to add error to the attribute:

```
$('#contact-form').yiiActiveForm('updateAttribute', 'contactform-subject', [
    "I have an error..."]);
```

[4] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L39

[5] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L50

[6] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L64

[7] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L74

[8] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L83

[9] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L93

[10] https://github.com/yiisoft/yii2/blob/master/framework/assets/yii.activeForm.js#L103

In order to remove it:

```
$('#contact-form').yiiActiveForm('updateAttribute', 'contactform-subject',
    '');
```

5.2.7 Update error messages and, optionally, summary

```
$('#contact-form').yiiActiveForm('updateMessages', {
    'contactform-subject': ['Really?'],
    'contactform-email': ['I don\'t like it!']
}, true);
```

The last argument in the above code indicates if we need to update summary.

5.2.8 Listening for attribute changes

To attach events to attribute changes like Select, Radio Buttons, etc.. you can use the following code

```
$("#attribute-id").on('change.yii',function(){
        //your code here
});
```

5.2.9 Getting Attribute Value

In order to be compatible with third party widgets like (Kartik), the best option to retrieve the actual value of an attribute is:

```
$('#form_id').yiiActiveForm('find', '#attribute').value
```

5.2.10 Custom Validation

In case you want to change the validation of an attribute in JS based on a new condition, you can do it with the rule property whenClient, but in the case you need a validation that doesn't depends on rules (only client side), you can try this:

```
$('#form_id').on('beforeValidate', function (e) {
            $('#form_id').yiiActiveForm('find', '#attribute').validate =
    function (attribute, value, messages, deferred, $form) {
                //Custom Validation
            }
        return true;
    });
```

5.3 Uploading files

Uploading files is explained in the guide[11] but it won't hurt to elaborate it a bit more because often, when Active Record model is reused as a form it causes confusion when a field storing file path is reused for file upload.

[11] http://www.yiiframework.com/doc-2.0/guide-input-file-upload.html

5.3.1 Objective

We'll have a posts manager with a form. In the form we'll be able to upload an image, enter title and text. Image is not mandatory. Existing image path should not be set to null when saving without image uploaded.

5.3.2 Preparations

We'll need a database table with the following structure:

```
CREATE TABLE post
(
    id INT(11) PRIMARY KEY NOT NULL AUTO_INCREMENT,
    title VARCHAR(255) NOT NULL,
    text TEXT NOT NULL,
    image VARCHAR(255)
);
```

Next, let's generate `Post` model with Gii and a CRUD in `PostController`.

Now we're ready to start.

5.3.3 Post model adjustments

Post model's `image` stores a path to the image uploaded it should not be confused with the actual file uploaded so we'll need a separate field for that purpose. Since the file isn't saved to database we don't need to store it. Let's just add a public field called `upload`:

```
class Post extends \yii\db\ActiveRecord
{
    public $upload;
```

Now we need to adjust validation rules:

```
/**
 * @inheritdoc
 */
public function rules()
{
    return [
        [['title', 'text'], 'required'],
        [['text'], 'string'],
        [['title'], 'string', 'max' => 255],
        [['upload'], 'file', 'extensions' => 'png, jpg'],
    ];
}
```

In the above we removed everything concerning `image` because it's not user input and added file validation for `upload`.

5.3.4 A form

A form in the `views/post/_form.php` needs two things. First, we should remove a field for `image`. Second, we should add a file upload field for `upload`:

```
<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-
    data']]); ?>

    <?= $form->field($model, 'title')->textInput(['maxlength' => true]) ?>

    <?= $form->field($model, 'text')->textarea(['rows' => 6]) ?>

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

    <div class="form-group">
        <?= Html::submitButton($model->isNewRecord ? 'Create' : 'Update', ['
    class' => $model->isNewRecord ? 'btn btn-success' : 'btn btn-primary'])
     ?>
    </div>

<?php ActiveForm::end(); ?>
```

5.3.5 Processing upload

The handling, for simplicity sake, is done in `PostController`. Two actions: `actionCreate` and `actionUpdate`. Both are repetitive so the first step is to extract handling into separate method:

```
public function actionCreate()
{
    $model = new Post();
    $this->handlePostSave($model);

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

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    $this->handlePostSave($model);

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

Now let's implement `handlePostSave()`:

```
protected function handlePostSave(Post $model)
{
    if ($model->load(Yii::$app->request->post())) {
        $model->upload = UploadedFile::getInstance($model, 'upload');

        if ($model->validate()) {
            if ($model->upload) {
```

```
                $filePath = 'uploads/' . $model->upload->baseName . '.' .
    $model->upload->extension;
                if ($model->upload->saveAs($filePath)) {
                    $model->image = $filePath;
                }
            }

            if ($model->save(false)) {
                return $this->redirect(['view', 'id' => $model->id]);
            }
        }
    }
}
```

In the code above right after filling model from POST we're also filling its `upload` field with an instance of the file uploaded. The important thing is that it should be done before validation. After validating a form, if there's file uploaded we're saving it and writing path into model's `image` field. Last, regular `save()` is called with a `false` argument which means "don't validate". It's done this way because we've just validated above and sure it's OK.

That's it. Objective achieved.

5.3.6 A note on forms and Active Record

For the sake of simplicity and laziness, Active Record is often reused for forms directly. There are scenarious making it doable and usually it doesn't cause any problems. However, there are situations when what's in the form actually differs from what is saved into database. In this case it is preferrable to create a separate form model which is not Active Record. Data saving should be done in this model instead of controller directly.

5.4 Custom validator for multiple attributes

Which creating custom validator is well covered in the official guide[12], there are cases when you need to validate multiple attributes at once. For example, it can be hard to choose which one is more relevant or you consider it misleading in rules. In this recipe we'll implement `CustomValidator` which supports for validating multiple attributes at once.

5.4.1 How to do it

By default if multiple attributes are used for validation, the loop will be used to apply the same validation to each of them. Let's use a separate trait and override `yii\base\Validator:validateAttributes()`:

[12] https://github.com/yiisoft/yii2/blob/master/docs/guide/input-validation.md#creating-validator-

```php
<?php

namespace app\components;

trait BatchValidationTrait
{
    /**
     * @var bool whether to validate multiple attributes at once
     */
    public $batch = false;

    /**
     * Validates the specified object.
     * @param \yii\base\Model $model the data model being validated.
     * @param array|null $attributes the list of attributes to be validated.
     * Note that if an attribute is not associated with the validator, or is
     is prefixed with '!' char - it will be
     * ignored. If this parameter is null, every attribute listed in [[
    attributes]] will be validated.
     */
    public function validateAttributes($model, $attributes = null)
    {
        if (is_array($attributes)) {
            $newAttributes = [];
            foreach ($attributes as $attribute) {
                if (in_array($attribute, $this->attributes) || in_array('!'
    . $attribute, $this->attributes)) {
                    $newAttributes[] = $attribute;
                }
            }
            $attributes = $newAttributes;
        } else {
            $attributes = [];
            foreach ($this->attributes as $attribute) {
                $attributes[] = $attribute[0] === '!' ? substr($attribute,
    1) : $attribute;
            }
        }

        foreach ($attributes as $attribute) {
            $skip = $this->skipOnError && $model->hasErrors($attribute)
                || $this->skipOnEmpty && $this->isEmpty($model->$attribute);
            if ($skip) {
                // Skip validation if at least one attribute is empty or
    already has error
                // (according skipOnError and skipOnEmpty options must be
    set to true
                return;
            }
        }

        if ($this->batch) {
            // Validate all attributes at once
```

```
            if ($this->when === null || call_user_func($this->when, $model,
    $attribute)) {
                // Pass array with all attributes instead of one attribute
                $this->validateAttribute($model, $attributes);
            }
        } else {
            // Validate each attribute separately using the same validation
    logic
            foreach ($attributes as $attribute) {
                if ($this->when === null || call_user_func($this->when,
    $model, $attribute)) {
                    $this->validateAttribute($model, $attribute);
                }
            }
        }
    }
}
```

Then we need to create custom validator and use the created trait:

```
<?php

namespace app\components;

use yii\validators\Validator;

class CustomValidator extends Validator
{
    use BatchValidationTrait;
}
```

To support inline validation as well we can extend default inline validator and also use this trait:

```
<?php

namespace app\components;

use yii\validators\InlineValidator;

class CustomInlineValidator extends InlineValidator
{
    use BatchValidationTrait;
}
```

Couple more changes are needed.

First to use our `CustomInlineValidator` instead of default `InlineValidator` we need to override `\yii\validators\Validator::createValidator()` method in `CustomValidator`:

```
public static function createValidator($type, $model, $attributes, $params =
     [])
{
    $params['attributes'] = $attributes;
```

```
    if ($type instanceof \Closure || $model->hasMethod($type)) {
        // method-based validator
        // The following line is changed to use our CustomInlineValidator
        $params['class'] = __NAMESPACE__ . '\CustomInlineValidator';
        $params['method'] = $type;
    } else {
        if (isset(static::$builtInValidators[$type])) {
            $type = static::$builtInValidators[$type];
        }
        if (is_array($type)) {
            $params = array_merge($type, $params);
        } else {
            $params['class'] = $type;
        }
    }

    return Yii::createObject($params);
}
```

And finally to support our custom validator in model we can create the trait and override `\yii\base\Model::createValidators()` like this:

```
<?php

namespace app\components;

use yii\base\InvalidConfigException;

trait CustomValidationTrait
{
    /**
     * Creates validator objects based on the validation rules specified in
    [[rules()]].
     * Unlike [[getValidators()]], each time this method is called, a new
    list of validators will be returned.
     * @return ArrayObject validators
     * @throws InvalidConfigException if any validation rule configuration
    is invalid
     */
    public function createValidators()
    {
        $validators = new ArrayObject;
        foreach ($this->rules() as $rule) {
            if ($rule instanceof Validator) {
                $validators->append($rule);
            } elseif (is_array($rule) && isset($rule[0], $rule[1])) { //
    attributes, validator type
                // The following line is changed in order to use our
    CustomValidator
                $validator = CustomValidator::createValidator($rule[1],
    $this, (array) $rule[0], array_slice($rule, 2));
                $validators->append($validator);
            } else {
                throw new InvalidConfigException('Invalid validation rule: a
```

```
     rule must specify both attribute names and validator type.');
            }
        }
        return $validators;
    }
}
```

Now we can implement custom validator by extending from `CustomValidator`:

```
<?php

namespace app\validators;

use app\components\CustomValidator;

class ChildrenFundsValidator extends CustomValidator
{
    public function validateAttribute($model, $attribute)
    {
        // $attribute here is not a single attribute, it's an array
    containing all related attributes
        $totalSalary = $this->personalSalary + $this->spouseSalary;
        // Double the minimal adult funds if spouse salary is specified
        $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 :
    self::MIN_ADULT_FUNDS;
        $childFunds = $totalSalary - $minAdultFunds;
        if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
            $this->addError('*', 'Your salary is not enough for children.');
        }
    }
}
```

Because `$attribute` contains the list of all related attributes, we can use loop in case of adding errors for all attributes is needed:

```
foreach ($attribute as $singleAttribute) {
    $this->addError($attribute, 'Your salary is not enough for children.');
}
```

Now it's possible to specify all related attributes in according validation rule:

```
[
    ['personalSalary', 'spouseSalary', 'childrenCount'],
    \app\validators\ChildrenFundsValidator::className(),
    'batch' => `true`,
    'when' => function ($model) {
        return $model->childrenCount > 0;
    }
],
```

For inline validation the rule will be:

```
[
    ['personalSalary', 'spouseSalary', 'childrenCount'],
    'validateChildrenFunds',
    'batch' => `true`,
    'when' => function ($model) {
```

```
        return $model->childrenCount > 0;
    }
],
```

And here is according validation method:

```
public function validateChildrenFunds($attribute, $params)
{
    // $attribute here is not a single attribute, it's an array containing
    all related attributes
    $totalSalary = $this->personalSalary + $this->spouseSalary;
    // Double the minimal adult funds if spouse salary is specified
    $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 : self
    ::MIN_ADULT_FUNDS;
    $childFunds = $totalSalary - $minAdultFunds;
    if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
        $this->addError('childrenCount', 'Your salary is not enough for
    children.');
    }
}
```

5.4.2 Summary

The advantages of this approach:

- It better reflects all attributes that participate in validation (the rules become more readable);
- It respects the options `yii\validators\Validator::skipOnError` and `yii\validators\Validator::skipOnEmpty` for **each** used attribute (not only for that you decided to choose as more relevant).

If you have problems with implementing client validation, you can:

- combine `yii\widgets\ActiveForm::enableAjaxValidation` and `yii\widgets\ActiveForm::enableAjaxValidation` options, so multiple attributes will be validated with AJAX without page reload;
- implement validation outside of `yii\validators\Validator::clientValidateAttribute` because it's designed to work with single attribute.

Chapter 6

Security

6.1 SQL injection

A SQL injection exploit can modify a database data. Please, always validate all input on the server. The following examples shows how to build parameterized queries:

```
$user = Yii::$app->db->createCommand('SELECT * FROM user WHERE id = :id')
           ->bindValue(':id', 123, PDO::PARAM_INT)
           ->queryOne();
```

```
$params = [':id' => 123];

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

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

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

$user = $command->bindValue(':id', 123)->queryOne();
```

```
// Wrong: don't do this!
$user = Yii::$app->db->createCommand('SELECT * FROM user WHERE id = ' .
    $_GET['id'])->queryOne();
```

6.2 XSS

Cross-site scripting (XSS) is a web application vulnerability caused by insufficient output escaping. It allows attacker to inject JavaScript code into your site pages. For example, if your website has comments, an attacker may add the following text as a comment:

```
<script>alert('Hello from hacker ;)');</script>
```

If there's no filtering and comment is published as is, every user who visits the page will get "Hello from hacker" alert box which means JavaScript is executed. And with JavaScript attacker can do virtually anything valid user can.

That's how it typically looks in Yii. In controller we're getting data and passing it to view:

```
public function actionIndex()
{
    $data = '<script>alert("injection example")</script>';
        return $this->render('index', [
        'data' => $data,
    ]);
}
```

And in `index.php` view we output data without any escaping:

```
echo $data;
```

That's it. We're vulnerable. Visit your website main page you will see "injection example" alert.

Next you'll learn how to prevent it.

6.2.1 Basic output escaping

If you're sure you'll have just text in your data, you can escape it in the view with `Html::encode()` while outputting it:

'php echo Html::encode($data); '

6.2.2 Dealing with HTML output

In case you need to output HTML entered by user it's getting a bit more complicated. Yii has a built in HtmlPurifier helper[1] which cleans up everything dangerous from HTML. In a view you may use it as the following:

```
echo HtmlPurifier::process($data);
```

> **Note:** HtmlPurifier isn't fast so consider caching what's produced by `HtmlPurifier` not to call it too often.

[1] http://www.yiiframework.com/doc-2.0/yii-helpers-basehtmlpurifier.html

6.2.3 See also

- OWASP article about XSS[2].
- HtmlPurifier helper class[3].
- HtmlPurifier website[4].

6.3 RBAC

RBAC which stands for Role Based Access Control is an access management system built into Yii. Despite being described well in official guide[5] there's no complete example on how to use it. Let's fill the gap.

As an example we'll take article publishing system such as YiiFeed[6].

6.3.1 Configuring RBAC component

Initial configuration of authentication manager component follows the same pattern as any other component configuration[7]: in the application config under `components` section we're adding section called `authManager` specifying a class and options for the object created. There are two backends available for authentication manager: PHP files and database. Both are using the same API so there's no difference besides how RBAC data is stored.

PHP backend

In order to configure PHP backend add the following to your config file:

```
return [
    // ...
    'components' => [
        // ...
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
    ],
    // ...
];
```

[2] https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29
[3] http://www.yiiframework.com/doc-2.0/yii-helpers-basehtmlpurifier.html
[4] http://htmlpurifier.org
[5] http://www.yiiframework.com/doc-2.0/guide-security-authorization.html#rbac
[6] http://yiifeed.com/
[7] http://www.yiiframework.com/doc-2.0/guide-structure-application-components.html

> **Note:** If you are using `yii2-basic-app` template, there is a `config/console.php` configuration file where the `authManager` needs to be declared additionally to `config/web.php`. In case of `yii2-advanced-app` the `authManager` should be declared only once in `common/config/main.php`.

By default PHP file backend stores RBAC data under `@app/rbac` directory. That means `rbac` directory should be created directly in your application directory and web server process should have permissions to write files into this directory.

Database backend

Setting up database backend is a bit more complex. First of all, add the following to your config file:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
        ],
        // ...
    ],
];
```

> **Note:** If you are using `yii2-basic-app` template, there is a `config/console.php` configuration file where the `authManager` needs to be declared additionally to `config/web.php`. In case of `yii2-advanced-app` the `authManager` should be declared only once in `common/config/main.php`.

Make sure you have database configured for both web and console applications then open console and run migration that would create all the tables necessary to store RBAC data:

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

6.3.2 Planning roles and permissions hierarchy

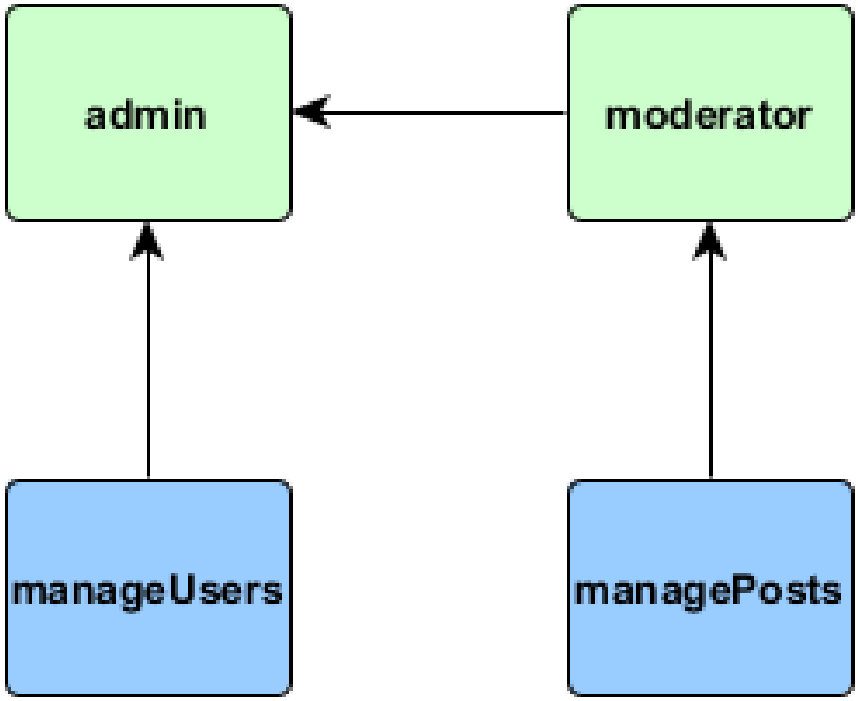
As an example we've chosen a publishing application. There are three types of users:

- Regular users who are reading and suggesting articles. They can edit their own articles as well.
- Moderators who are editing, approving, deleting or denying articles. They have access to moderation queue.

- Administrators who can do everything moderator can plus view list of users and edit their profiles.

At this point it's good to get pen and paper or software like yEd[8] and draw the hierarchy.

The first rule of successfully using RBAC is to use as simple hierarchy as possible. In our case what regular user can do isn't really needed to be associated with any role since we can make it work by default. Editing, approving, deleting or denying articles could be expressed as "managing articles". Viewing list of users and editing their profiles could be expressed as "managing users". This simplification leads us to the following hierarchy:



6.3.3 Filling hierarchy

If your project uses database and you're already familiar with migrations[9] it's better to build our hierarchy in migration.

Open your console and type

```
./yii migrate/create rbac_init
```

[8] https://www.yworks.com/products/yed
[9] http://www.yiiframework.com/doc-2.0/guide-db-migrations.html

That would create new migration class with `up()` method in which we'd build the hierarchy and `down()` in which we'll destroy it.

```
use yii\db\Migration;

class m141204_121823_rbac_init extends Migration
{
    public function up()
    {
        $auth = Yii::$app->authManager;

        $manageArticles = $auth->createPermission('manageArticles');
        $manageArticles->description = 'Manage articles';
        $auth->add($manageArticles);

        $manageUsers = $auth->createPermission('manageUsers');
        $manageUsers->description = 'Manage users';
        $auth->add($manageUsers);

        $moderator = $auth->createRole('moderator');
        $moderator->description = 'Moderator';
        $auth->add($moderator);
        $auth->addChild($moderator, $manageArticles);

        $admin = $auth->createRole('admin');
        $admin->description = 'Administrator';
        $auth->add($admin);
        $auth->addChild($admin, $moderator);
        $auth->addChild($admin, $manageUsers);
    }

    public function down()
    {
        Yii::$app->authManager->removeAll();
    }
}
```

In the above `createPermission()` and `createRole()` are creating new hierarchy objects but not yet saving them. In order to save them `add()` should be called. `addChild()` method is used to connect child object to their parents. When called this method saves connections immediately.

> **Note:** It doesn't matter which backend you're using: PHP files or database. Authentication manager exposes exactly the same methods so hierarchy is built using exactly the same code.

In case your application isn't using database at all or you don't want to use migrations, you can do the same in a console command. For basic project template that would be `commands\RbacController.php`:

```
<?php
namespace app\commands;
```

```
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        if (!$this->confirm("Are you sure? It will re-create permissions
    tree.")) {
            return self::EXIT_CODE_NORMAL;
        }

        $auth = Yii::$app->authManager;
        $auth->removeAll();

        $manageArticles = $auth->createPermission('manageArticles');
        $manageArticles->description = 'Manage articles';
        $auth->add($manageArticles);

        $manageUsers = $auth->createPermission('manageUsers');
        $manageUsers->description = 'Manage users';
        $auth->add($manageUsers);

        $moderator = $auth->createRole('moderator');
        $moderator->description = 'Moderator';
        $auth->add($moderator);
        $auth->addChild($moderator, $manageArticles);

        $admin = $auth->createRole('admin');
        $admin->description = 'Administrator';
        $auth->add($admin);
        $auth->addChild($admin, $moderator);
        $auth->addChild($admin, $manageUsers);
    }
}
```

The command above could be called as `./yii rbac/init`.

6.3.4 Assigning role to user

Since our default user doesn't have any role we don't need to worry about assigning it. User role management could be implemented either in admin panel or in console. Since our admins are cool guys, we'll create console contoller `commands\RbacController.php`:

```
<?php
namespace app\commands;

use yii\console\Controller;

class RbacController extends Controller
{
```

```
    public function actionAssign($role, $username)
    {
        $user = User::find()->where(['username' => $username])->one();
        if (!$user) {
            throw new InvalidParamException("There is no user \"$username\".
    ");
        }

        $auth = Yii::$app->authManager;
        $roleObject = $auth->getRole($role);
        if (!$roleObject) {
            throw new InvalidParamException("There is no role \"$role\".");
        }

        $auth->assign($roleObject, $user->id);
    }
}
```

In the code above we're finding a user by username specified. Then getting role object by its name and assigning role to a user by ID. Again, it doesn't matter if PHP backend or database backend is used. It would look exactly the same.

Also it would be exactly the same assignment in case of implementing admin UI or in case when you need role right away and assigning it right after user is successfully singed up.

Sign up three new users and assign two of them `admin` and `moderator` roles respectively:

```
./yii rbac/assign admin qiang
./yii rbac/assign moderator alex
```

6.3.5 Checking access

Now we have RBAC in place and three users: regular user, moderator and admin. Let's start using what we've created.

Access filter

The very basic access checks could be done via access control filter which is covered well in the official guide[10]:

```
namespace app\controllers;

use yii\web\Controller;
use yii\filters\AccessControl;

class ArticleController extends Controller
```

[10] http://www.yiiframework.com/doc-2.0/guide-security-authorization.html#access-control-filter

```
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['suggest', 'queue', 'delete', 'update'], //only
    be applied to
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['suggest', 'update'],
                        'roles' => ['@'],
                    ],
                    [
                        'allow' => true,
                        'actions' => ['queue', 'delete'],
                        'roles' => ['manageArticles'],
                    ],
                ],
            ],
            'verbs' => [
                'class' => VerbFilter::className(),
                'actions' => [
                    'delete' => ['post'],
                ],
            ],
        ];
    }

    // ...
```

We're allowing any authenticated user to suggest articles. Same applied to editing articles (it's explained in the next section). Viewing moderation queue and deleting articles are available only to roles which have `manageArticles` permission. In our case it's both `admin` and `moderator` since `admin` inherits all `moderator` permissions.

Same simple checks via access control filter could be applied to `UserController` which handles admin actions regarding users.

Doing manual checks

In some cases it's necessary to do manual checks. In our case it's checking if users is allowed to edit an article. We can't do it via access control filter because we need to allow editing for regular users owning an article and moderators at the same time:

```
namespace app\controllers;

use app\models\Article;
use yii\web\Controller;
```

```
use yii\filters\AccessControl;

class ArticleController extends Controller
{
    // ...
    public function actionUpdate($id)
    {
        $model = $this->findModel($id);
        if (Yii::$app->user->id == $model->user_id || \Yii::$app->user->can(
    'manageArticles')) {
            // ...
        } else {
            throw new ForbiddenHttpException('You are not allowed to edit
    this article.');
        }
    }
}
```

In the code above we're checking if current user is either article owner or is allowed to manage articles. If either one is true, we're proceeding normally. Otherwise denying access.

6.4 CSRF

Cross-site request Forgery (CSRF) is one of a typical web application vulnerabilities. It's based on the assumption that user may be authenticated at some legitimate website. Then he's visiting attacker's website which issues requests to legitimate website using JavaScript code, a form, `<img src="` tag or any other means. This way attacker could, for example, reset victim's password or transfer funds from his bank account (in case bank website isn't secure, of course).

One of the ways[11] to prevent this type of attacks is to use a special token to validate request origin. Yii does it by default. You can try it by adding a simple form to you view and submitting it:

```
<form method="POST">
    <input type="submit" value="ok!">
</form>
```

You'll get the following error:

```
Bad Request (#400)
Unable to verify your data submission.
```

This is because for every request Yii generates a special unique token that you have to send with your request data. So when you make a request Yii compares generated and received tokens. If they match Yii continues to

[11] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet

handle request. If they don't match or in case CSRF token is missing in request data, an error is raised.

The token generated isn't known to attacker website so it can't make requests on your behalf.

An important thing to note is about request methods such as `GET`. Let's add the following form and submit it:

```
<form method="GET">
    <input type="submit" value="ok!">
</form>
```

No error has occurred. This is because CSRF protection works only for unsafe request methods such `PUT`, `POST` or `DELETE`. That's why in order to stay safe your application should never ever use `GET` requests to change application state.

6.4.1 Disabling CSRF protection

In some cases you may need to disable CSRF validation. In order to do it set `$enableCsrfValidation` Controller property to `false`:

```
class MyController extends Controller
{
    public $enableCsrfValidation = false;
```

In order to do the same for a certain action use the following code:

```
class MyController extends Controller
{
    public function beforeAction($action)
    {
        if (in_array($action->id, ['incoming'])) {
            $this->enableCsrfValidation = false;
        }
        return parent::beforeAction($action);
    }
```

See handling incoming third party POST requests for details.

6.4.2 Adding CSRF protection

CSRF protection is enabled by default so what you need is to submit a token along with all your requests. Usually it's done via hidden field:

```
<form method="POST">
    <input id="form-token" type="hidden" name="<?=Yii::$app->request->
    csrfParam?>"
           value="<?=Yii::$app->request->csrfToken?>"/>
    <input type="submit" value="ok!">
</form>
```

In case `ActiveForm` is used, token is added automatically.

6.4.3 See also

- OWASP article about CSRF[12]
- Handling incoming third party POST requests.

[12] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet

Chapter 7

Structuring and organizing code

7.1 Structure: backend and frontend via modules

https://github.com/yiisoft/yii2/issues/3647

By default Yii comes with advanced application template that allows you to properly structure backend and frontend. It's a good way to deal with the problem except when you want to redistribute parts of your application. In this case a better way is to use modules.

7.1.1 Directory structure

```
common
  components
  models
backend
  controllers
  views
  Module
frontend
  controllers
  views
  Module
```

7.1.2 Namespaces

Root namespace is the same as in any extension i.e. `samdark\blog` (PSR-4 record required in `composer.json`). Common stuff is under `samdark\blog\common`. Backend module is `samdark\blog\backend\Module`, frontend module is `samdark\blog\frontend\Module`.

7.1.3 Using it

- Install via Composer.

- In your application config use modules:

```
'modules' => [
  'blogFrontend' => [
    'class' => 'samdark\blog\frontend\Module',
    'anonymousComments' => false,
  ],
  'blogBackend' => [
    'class' => 'samdark\blog\backend\Module',
  ],
]
```

- Access via browser:

```
http://example.com/blog-frontend/post/view?id=10
http://example.com/blog-backend/user/index
```

7.2 Asset processing with Grunt

Yii 2.0 has pretty good asset management out of the box[1]. It can handle publishing, mapping, format conversion, combining and compression. So far so good but if you're working with frontend team of your asset processing goes slightly beyond what Yii is capable of, it's a good idea to delegate the job to Grunt[2] which has lots of extensions capable of what Yii can do plus anything you can imagine about clientside development.

7.2.1 Get ready

We'll start with basic application template. Its installation is described in official guide[3].

If you haven't installed Node.js[4], do so. After it's done install TypeScript, Grunt and its required plugins by executing the following commands in project root directory:

```
npm install -g grunt-cli

npm install grunt --save-dev
npm install grunt-contrib-copy --save-dev
npm install grunt-contrib-less --save-dev
npm install grunt-contrib-uglify --save-dev
npm install grunt-contrib-watch --save-dev
npm install grunt-concat-sourcemap --save-dev
npm install typescript --save-dev
npm install grunt-typescript --save-dev
```

[1] http://www.yiiframework.com/doc-2.0/guide-structure-assets.html
[2] http://gruntjs.com/
[3] http://www.yiiframework.com/doc-2.0/guide-start-installation.html
[4] http://nodejs.org/

7.2.2 How to do it...

First of all, turn off built in Yii asset management via editing `config/web.php`:

```
$params = require(__DIR__ . '/params.php');

$config = [
    // ...
    'components' => [
        // ...
        'assetManager' => [
            'bundles' => false,
        ],
    ],
];

// ...

return $config;
```

Edit layout file `views/layouts/main.php`. After `<?= Html::csrfMetaTags() ?>` add:

```
<?= Html::cssFile(YII_DEBUG ? '@web/css/all.css' : '@web/css/all.min.css?v='
     . filemtime(Yii::getAlias('@webroot/css/all.min.css'))) ?>
```

It adds a link to `http://example.com/css/all.css` in debug mode and a link to `http://example.com/css/all.min.css` with modification time (cache busting) in production mode. The file itself will be published by Grunt.

Right before `<?php $this->endBody() ?>` add:

```
<?= Html::jsFile(YII_DEBUG ? '@web/js/lib.js' : '@web/js/lib.min.js?v=' .
    filemtime(Yii::getAlias('@webroot/js/lib.min.js'))) ?>
<?= Html::jsFile(YII_DEBUG ? '@web/js/all.js' : '@web/js/all.min.js?v=' .
    filemtime(Yii::getAlias('@webroot/js/all.min.js'))) ?>
```

Same as with CSS, it adds a link for JS that is published via Grunt.

Now create `Gruntfile.js` in the root of the project. The file describes what grunt will do with your assets:

```
module.exports = function (grunt) {
    grunt.initConfig({
        less: {
            dev: {
                options: {
                    compress: false,
                    sourceMap: true,
                    outputSourceFiles: true
                },
                files: {
                    "web/css/all.css": "assets/less/all.less"
                }
            },
            prod: {
                options: {
                    compress: true
```

```
            },
            files: {
                "web/css/all.min.css": "assets/less/all.less"
            }
        }
    },
    typescript: {
        base: {
            src: ['assets/ts/*.ts'],
            dest: 'web/js/all.js',
            options: {
                module: 'amd',
                sourceMap: true,
                target: 'es5'
            }
        }
    },
    concat_sourcemap: {
        options: {
            sourcesContent: true
        },
        all: {
            files: {
                'web/js/all.js': grunt.file.readJSON('assets/js/all.json
')
            }
        }
    },
    copy: {
        main: {
            files: [
                {expand: true, flatten: true, src: ['vendor/bower/
bootstrap/fonts/*'], dest: 'web/fonts/', filter: 'isFile'}
            ]
        }
    },
    uglify: {
        options: {
            mangle: false
        },
        lib: {
            files: {
                'web/js/lib.min.js': 'web/js/lib.js'
            }
        },
        all: {
            files: {
                'web/js/all.min.js': 'web/js/all.js'
            }
        }
    },
    watch: {
        typescript: {
            files: ['assets/ts/*.ts'],
```

```
                tasks: ['typescript', 'uglify:all'],
                options: {
                    livereload: true
                }
            },
            js: {
                files: ['assets/js/**/*.js', 'assets/js/all.json'],
                tasks: ['concat_sourcemap', 'uglify:lib'],
                options: {
                    livereload: true
                }
            },
            less: {
                files: ['assets/less/**/*.less'],
                tasks: ['less'],
                options: {
                    livereload: true
                }
            },
            fonts: {
                files: [
                    'vendor/bower/bootstrap/fonts/*'
                ],
                tasks: ['copy'],
                options: {
                    livereload: true
                }
            }
        }
    });

    // Plugin loading
    grunt.loadNpmTasks('grunt-typescript');
    grunt.loadNpmTasks('grunt-concat-sourcemap');
    grunt.loadNpmTasks('grunt-contrib-watch');
    grunt.loadNpmTasks('grunt-contrib-less');
    grunt.loadNpmTasks('grunt-contrib-uglify');
    grunt.loadNpmTasks('grunt-contrib-copy');

    // Task definition
    grunt.registerTask('build', ['less', 'typescript', 'copy', '
    concat_sourcemap', 'uglify']);
    grunt.registerTask('default', ['watch']);
};
```

Now Grunt will look in `assets/js`, `assets/less` and `assets/ts` for clientside source files.

Create `assets/js/all.json`:

```
[
    "vendor/bower/jquery/dist/jquery.js",
    "vendor/bower/bootstrap/dist/js/bootstrap.js",
    "vendor/yiisoft/yii2/assets/yii.js",
    "vendor/yiisoft/yii2/assets/yii.validation.js",
```

```
    "vendor/yiisoft/yii2/assets/yii.activeForm.js"
]
```

`all.json` lists JavaScript files to process into `lib.js`. In the above we're doing the same things standard Yii asset management does: adding jQuery, bootstrap and Yii's JavaScript.

Now create `assets/less/all.less`:

```
@import "../../vendor/bower/bootstrap/less/bootstrap.less";
@import "site.less";
```

and `assets/less/site.less`. Its content should be copied from `web/css/site.css`.

7.2.3 How to use it

- Run `grunt build` to process assets.
- During development you could run `grunt` and the process will watch for changes and rebuild files necessary.
- In order to add JavaScript files, put these into `assets/js` and list their names in `assets/js/all.json`.
- In order to add LESS files[5], put these into `assets/less` and list their names in `assets/less/all.less`.
- In order to add TypeScript files[6] just put these into `assets/ts`.

7.3 Using global functions

Although it looks like a weird idea at the first glance, using just functions in PHP is actually nice. Code looks much shorter and, with a good naming choices, much simpler.

7.3.1 How to do it

First of all, create a file that will contain functions. Let it be `functions.php` right in the root of the application. In order to be used it should in `required`. The best place to do it is `index.php`:

```
// ...
require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

$config = require(__DIR__ . '/../config/web.php');

$app = new yii\web\Application($config);
require(__DIR__ . '/../functions.php');
$app->run();
```

[5] http://lesscss.org/
[6] http://www.typescriptlang.org/

Note that we're requiring it after including config and creating application instance. That allows you to use config and application which is needed for many functions.

Also you can do it in `composer.json`:

```
"autoload": {
    "files": [
        "functions.php"
    ]
},
```

Note that after add this section you need run `composer update`.

7.3.2 Function ideas

Below are some function ideas. You can add more if needed:

```
use yii\helpers\Url;
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
use yii\helpers\ArrayHelper;

function url($url = '', $scheme = false)
{
    return Url::to($url, $scheme);
}

function h($text)
{
    return Html::encode($text);
}

function ph($text)
{
    return HtmlPurifier::process($text);
}

function t($message, $params = [], $category = 'app', $language = null)
{
    return Yii::t($category, $message, $params, $language);
}

function param($name, $default = null)
{
    return ArrayHelper::getValue(Yii::$app->params, $name, $default);
}
```

7.4 Processing text

When implementing post or article publishing, it's very important to choose right tool for the job. Common approach is to use WYSIWYG (what you see is what you get) editor that produces HTML but that has significant

cons. The most prominent con is that it's easy to break website design and to produce excessive and ugly HTML. The pro is that it's quite natural for people worked with MS Word or alike text processors.

Luckily, we have simple markups such as markdown nowadays. While being very simple, it has everything to do basic text formatting: emphasis, hyperlinks, headers, tables, code blocks etc. For tricky cases it still accepts HTML.

7.4.1 Converting markdown to HTML

How to do it

Markdown helper is very easy to use:

```
$myHtml = Markdown::process($myText); // use original markdown flavor
$myHtml = Markdown::process($myText, 'gfm'); // use github flavored markdown
$myHtml = Markdown::process($myText, 'extra'); // use markdown extra
```

How to secure output

Since markdown allows pure HTML as well, it's not secure to use it as is. Thus we'll need to post-process output via HTMLPurifier:

```
$safeHtml = HtmlPurifier::process($unsafeHtml);
```

Where to do it

- The library used to convert markdown to HTML is fast so processing right in the view could be OK.
- Result could be saved into separate field in database or cached for extra performance. Both could be done in `afterSave` method of the model. Note that in case of database field we can't save processed HTML to the same field because of the need to edit original.

7.4.2 Alternatives

Markdown is not the only simple markup available. A good overview exists in Wikipedia[7].

7.5 Implementing typed collections

For stricter type hinting and iterfaces it could be useful to implement typed collections to put your models and other same-typed classes into.

As an example, we'll assume we have a `Post` class:

[7] https://en.wikipedia.org/wiki/Lightweight_markup_language

```
class Post
{
    private $title;

    public function __construct($title)
    {
        $this->title = $title;
    }

    public function getTitle()
    {
        return $this->title;
    }
}
```

A simple typed immutable collection could be implemented like the following:

```
class PostCollection implements \Countable, \IteratorAggregate
{
    private $data;

    public function __construct(array $data)
    {
        foreach ($data as $item) {
            if (!$item instanceof Post) {
                throw new \InvalidArgumentException('All items should be of
    Post class.');
            }
        }
        $this->data = $data;
    }

    public function count()
    {
        return count($this->data);
    }

    public function getIterator()
    {
        return new \ArrayIterator($this->data);
    }
}
```

That's it. Now you can use it like the following:

```
$data = [new Post('post1'), new Post('post 2')];
$collection = new PostCollection($data);

foreach ($collection as $post) {
    echo $post->getTitle();
}
```

The main pro besides type checking in the constructor is about interfaces. With typed collections you may explicitly require a set of items of a certain class:

```
interface FeedGenerator
{
    public function generateFromPosts(PostsCollection $posts);
}

// instead of

interface FeedGenerator
{
    public function generateFromPosts(array $posts);
}
```

7.6 MVC

If you've been working in web development you've most probably heard of the term "MVC" and know that it refers to "Model", "View", and "Controller". The "MVC" software pattern was introduced in the nineties to explain the principles of building desktop applications. In particular to describe how a user interface interacts with a program and its underlying data. Later it was adjusted and adapted by web developers and has become a core foundation of many popular web frameworks including PHP frameworks. The name of a new pattern is still "MVC" and is often causing confusion.

A key issue for beginners with MVC is that despite its simplicity, it's not always easy to identify what a "Model", "View", and "Controller" really mean. Let's find out by starting with the simplest one.

7.6.1 Controller

The role of the controller is to accept input and convert it to commands for the model or view. Essentially the controller works with external data and environment such as:

- GET, POST and other requests in web
- user input in console

7.6.2 View

The view layer processes and formats data from the controller before sending it to the user. It generates HTML, JSON, or whatever format is needed.

> **Note:** It is strictly forbidden to work with any environment, database or user input directly in the view. It should be in controller.

7.6.3 Model

The model is the most interesting part of the MVC pattern and the most misunderstood one. A MVC model is often confused with the Yii Model class

and even Active Record. These are not the same thing. The MVC model is not a single class, it's the whole domain layer (also called the problem layer or application logic). Given data from the controller it's doing actual work of the application and passing results back to controller.

> **Note:** Its possible the model can be totally disconnected from the database structure.

ActiveRecord classes should not contain any significant business logic. It deserves to be in separate classes which are built according to SOLID and Dependency Inversion. Don't be afraid to create your own classes which are not inherited from anything from the framework.

> **Note:** The Model should never deal with formatting i.e. it should not produce any HTML. This is the job of the view layer. Also, same as in the view, it is strictly forbidden to work with any environment, database or user input directly in the view. It should be in controller.

7.7 SOLID

SOLID is a set of principles that you should follow if you want to get pure object oriented code which is easy to test and extend.

These stand for:

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Let's check what these mean.

7.7.1 Single responsibility

A class should be responsible for a single task.

7.7.2 Open-closed

A class or module (a set of related classes) should hide its implementation details (i.e. how exactly things are done) but have a well defined interface that both allows its usage by other classes (public methods) and extension via inheritance (protected and public methods).

7.7.3 Liskov substitution

LSP, when defined classically, is the most complicated principle in SOLID. In fact, it's not that complicated.

It is about class hierarchies and inheritance. When you implement a new class extending from base one it should have the same interface and behave exactly the same in same situations so it's possible to interchange these classes.

For more see a good set of answers at StackOverflow[8].

7.7.4 Interface segregation

The principle points that an interface should not define more functionality that is actually used at the same time. It is like single responsibiltiy but for interfaces. In other words: if an interface does too much, break it into multiple more focused interfaces.

7.7.5 Dependency inversion

Dependency inversion basically states that a class should tell what it needs via interfaces but never get what it needs itself.

See dependencies.md for more information.

7.8 Dependencies

Class A depends on class B when B is used within A i.e. when B is required for A to function.

7.8.1 Bad and good dependencies

There are two metrics of dependencies:

- Cohesion.
- Coupling.

Cohesion means dependency on a class with related functionality.

Coupling means dependency on a class with not really related functionality.

It's preferrable to have high cohesion and low coupling. That means you should get related functionality together into a group of classes usually called a module (which is *not* Yii module and not an actual class, just a logical boundary). Within that module it's a good idea not to over-abstract things and use interconnected classes directly. As for classes which aren't part of the module's purpose but are used by the module, such as general purpose utilities, these should not be used directly but through interface. Via this

[8] http://stackoverflow.com/questions/56860/what-is-the-liskov-substitution-principle

interface module states what is needed for it to function and from this point it doesn't care how these dependencies are satisfied.

7.8.2 Achieving low coupling

You can't eliminate dependencies altogether but you can make them more flexible.

7.8.3 Inversion of control

7.8.4 Dependency injection

7.8.5 Dependency container

Injecting basic dependencies is simple and easy. You're choosing a place where you don't care about dependencies, which is usually controller which you aren't going to unit-test ever, create instances of dependencies needed and pass these to dependent classes.

It works well when there aren't many dependencies overall and when there are no nested dependencies. When there are many and each dependency has dependencies itself, instantiating the whole hierarchy becomes tedious process which requires lots of code and may lead to hard to debug mistakes.

Additionally, lots of dependencies, such as certain third party API wrapper, are the same for any class using it. So it makes sense to:

- Define how to instantiate such API wrapper once.
- Instantiate it when required and only once per request.

That's what dependency containers are for.

See official guide[9] for more information about Yii's dependency container.

[9] http://www.yiiframework.com/doc-2.0/guide-concept-di-container.html

Chapter 8

View

8.1 Reusing views via partials

One of the main developing principles is DRY - don't repeat yourself. Duplication happens everywhere during development of the project including views. In order to fix it let's create reusable views.

8.1.1 Creating partial view

Here's a part of a standard `views/site/index.php` code:

```
<?php
/* @var $this yii\web\View */
$this->title = 'My Yii Application';
?>
<div class="site-index">
<div class="jumbotron">
    <h1>Congratulations!</h1>
    <p class="lead">You have successfully created your Yii-powered
    application.</p>
    <p><a class="btn btn-lg btn-success" href="http://www.yiiframework.com">
    Get started with Yii</a></p>
</div>
<div class="body-content">
//...
```

For example, we want to show `<div class="jumbotron">` HTML block both on the front page and inside `views/site/about.php` view which is for *about* page. Let's create a separate view file `views/site/_jumbotron.php` and place the following code inside:

```
<div class="jumbotron">
    <h1>Congratulations!</h1>
    <p class="lead">You have successfully created your Yii-powered
    application.</p>
    <p><a class="btn btn-lg btn-success" href="http://www.yiiframework.com">
    Get started with Yii</a></p>
</div>
```

8.1.2 Using partial view

Replace `<div class="jumbotron">` HTML block inside `views/site/index.php` with the following code:

```
<?php
/* @var $this yii\web\View */
$this->title = 'My Yii Application';
?>
<div class="site-index">
<?=$this->render('_jumbotron.php')?>; // this line replaces standard block
<div class="body-content">
//...
```

Let's add the same code line inside `views/site/about.php` (or inside another view):

```
<?php
use yii\helpers\Html;
/* @var $this yii\web\View */
$this->title = 'About';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="site-about">
    <?=$this->render('_jumbotron.php')?>; // our line
```

In the code above we're relying on `View::render()`[1] method which renders a view specified and returns its output which we're echoing immediately.

8.1.3 Adding variables

Let's customize message displayed in jumbotron. By default it will be the same message but user should be able to pass custom message via `message` parameter.

First of all, customize `views/site/_jumbotron.php`:

```
<?php
$message = isset($message) ? $message : 'You have successfully created your
    Yii-powered application.';
?>
<div class="jumbotron">
    <h1>Congratulations!</h1>
    <p class="lead"><?= $message ?></p>
    <p><a class="btn btn-lg btn-success" href="http://www.yiiframework.com">
    Get started with Yii</a></p>
</div>
```

Now let's pass custom message for about page:

```
<?php
use yii\helpers\Html;
/* @var $this yii\web\View */
```

[1] http://www.yiiframework.com/doc-2.0/yii-base-view.html#render%28%29-detail

```
$this->title = 'About';
$this->params['breadcrumbs'][] = $this->title;
?>
<div class="site-about">
    <?=$this->render('_jumbotron.php', [
        'message' => 'This is about page!',
    ])?>; // our line
```

8.2 Switching themes dynamically

View themes are useful for overriding extension views and making special view versions. Official Yii guide[2] describes static usage and configuration of views well so in this recipe we'll learn how to switch themes dynamically.

8.2.1 Preparations

We'll start with the basic project template[3]. Make sure it is installed and works well.

8.2.2 The goal

For simplicity, let's switch theme based on a GET parameter i.e. `themed=1`.

8.2.3 How to do it

Theme could be switched at any moment before view template is rendered. For clarity sake let's do it right in index action of `controllers/SiteController.php`:

```
public function actionIndex()
{
    if (Yii::$app->request->get('themed')) {
        Yii::$app->getView()->theme = new Theme([
            'basePath' => '@app/themes/basic',
            'baseUrl' => '@web/themes/basic',
            'pathMap' => [
                '@app/views' => '@app/themes/basic',
            ],
        ]);
    }
    return $this->render('index');
}
```

If there's a `themed` GET parameter we're configuring current a theme which takes view templates from `themes/basic` directory. Let's add customized template itself in `themes/basic/site/index.php`:

[2] http://www.yiiframework.com/doc-2.0/guide-output-theming.html
[3] http://www.yiiframework.com/doc-2.0/guide-start-installation.html

```
Hello, I'm a custom theme!
```

That's it. Now try accessing homepage with and without `themed` GET parameter.

8.3 Post-processing response

Sometimes there's a need to post-process repsonse which is to be sent to browser. A good example is short tags like the ones in Wordpress engine. You use it like the following:

```
This is [username]. We have [visitor_count] visitors on website.
```

And both are automatically replaced by corresponding content.

8.3.1 How to do it

Yii is very flexible so it's easy to achieve:

```
Yii::$app->getResponse()->on(Response::EVENT_AFTER_PREPARE, function($event)
     {
    /** @var User $user */
    $user = Yii::$app->getUser()->getIdentity();
    $replacements = [
        '[username]' => $user->username,
        '[visitor_count]' => 42,
    ];

    $event->sender->content = str_replace(array_keys($replacements),
    array_values($replacements), $event->sender->content);
});
```

In the code above we're using `Response::EVENT_AFTER_PREPARE` which is triggered right before sending content to a browser. In the callback `$event->sender` is our response object which keeps data to be sent in `content` property. So we are finding and replacing short tags there.

Chapter 9

Models

Chapter 10

Active Record

10.1 Single table inheritance

There is no native inheritance support in most relational databases so it should be implemented manually if needed. One of approaches to the problem is called single table inheritance[1], described well by Martin Fowler.

According to the pattern in the entity table we add an additional column called `type` that determines which class will be instantiated from a row of data.

Let's implement simple car types inheritance with the following class structure:

```
Car
|- SportCar
|- HeavyCar
```

10.1.1 Get ready

We'll use a basic application. After creating and setting up a database, execute the following SQL to create the table and insert some data:

```
CREATE TABLE `car` (
    `id` int NOT NULL AUTO_INCREMENT,
    `name` varchar(255) NOT NULL,
    `type` varchar(255) DEFAULT NULL,
    PRIMARY KEY (`id`)
);

INSERT INTO car (id, NAME, TYPE) VALUES (1, 'Kamaz', 'heavy'), (2, 'Ferrari'
    , 'sport'), (3, 'BMW', 'city');
```

Now use Gii to generate `Car` model.

[1] http://martinfowler.com/eaaCatalog/singleTableInheritance.html

10.1.2 How to do it...

We'll need a quite simple custom query class in order to always apply car type to query condition. Create `models/CarQuery.php`:

```
namespace app\models;

use yii\db\ActiveQuery;

class CarQuery extends ActiveQuery
{
    public $type;
    public $tableName;

    public function prepare($builder)
    {
        if ($this->type !== null) {
            $this->andWhere(["$this->tableName.type" => $this->type]);
        }
        return parent::prepare($builder);
    }
}
```

Now let's create models for car classes for different types. First `models/SportCar.php`:

```
namespace app\models;

class SportCar extends Car
{
    const TYPE = 'sport';

    public function init()
    {
        $this->type = self::TYPE;
        parent::init();
    }

    public static function find()
    {
        return new CarQuery(get_called_class(), ['type' => self::TYPE, '
    tableName' => self::tableName()]);
    }

    public function beforeSave($insert)
    {
        $this->type = self::TYPE;
        return parent::beforeSave($insert);
    }
}
```

Then `models/HeavyCar.php`:

```
namespace app\models;

class HeavyCar extends Car
```

```
{
    const TYPE = 'heavy';

    public function init()
    {
        $this->type = self::TYPE;
        parent::init();
    }

    public static function find()
    {
        return new CarQuery(get_called_class(), ['type' => self::TYPE, '
    tableName' => self::tableName()]);
    }

    public function beforeSave($insert)
    {
        $this->type = self::TYPE;
        return parent::beforeSave($insert);
    }
}
```

Now we need to override `instantiate` method in the `Car` model:

```
public static function instantiate($row)
{
    switch ($row['type']) {
        case SportCar::TYPE:
            return new SportCar();
        case HeavyCar::TYPE:
            return new HeavyCar();
        default:
           return new self;
    }
}
```

Also we need to override `tableName` method in the `Car` model in order for all models involved to use a single table:

```
public static function tableName()
{
    return '{{%car%}}';
}
```

That's it. Let's try it. Create the following `actionTest` in `SiteController` and run it:

```
// finding all cars we have
$cars = Car::find()->all();
foreach ($cars as $car) {
    echo "$car->id $car->name " . get_class($car) . "<br />";
}

// finding any sport car
$sportCar = SportCar::find()->limit(1)->one();
echo "$sportCar->id $sportCar->name " . get_class($sportCar) . "<br />";
```

The output should be:

```
1 Kamaz app\models\HeavyCar
2 Ferrari app\models\SportCar
3 BMW app\models\Car
2 Ferrari app\models\SportCar
```

That means models are now instantiated according to `type` field and the search is performed as expected.

10.1.3 How it works...

`SportCar` and `HeavyCar` models are quite similar. They both extend from `Car` and have two methods overridden. In `find` method we're instantiating a custom query class that stores car type and applies it in the `prepare` method that is called right before forming SQL for the database query. `SportCar` will only search for sport cars and `HeavyCar` will only search for heavy cars. In `beforeSave` we're making sure that the proper `type` is written to database when class is saved. `TYPE` constants are introduced just for convenience.

The `Car` model is pretty much what was generated by Gii except additional `instantiate` method. This method is called after data is retrieved from database and is about to be used to initialize class properties. Return value is uninitialized class instance and the only argument passed to the method is the row of data retrieved from the database. Exactly what we need. The implementation is a simple switch statement where we're checking if the `type` field matches type of the classes we suport. If so, an instance of the class is returned. If nothing matches, it falls back to returning a `Car` model instance.

10.1.4 Handling unique values

If you have a column marked as unique, to prevent breaking the `UniqueValidator` `you need to specify the` targetClass‘ property.

```
    public function rules()
    {
        return [
            [['MyUniqueColumnName'], 'unique', 'targetClass' => '\app\models
    \Car'],
        ];
    }
```

Chapter 11

i18n

11.1 Selecting application language

When developing applications or websites for global marked, supporting multiple languages is always a requirement. Yii has built in solution for handling message translations but doesn't provide anything about selecting a language because implementation depends of requirements.

In this recipe we'll describe some typical cases of language selection and provide ideas and code snippets so you'll be able to pick what's required and implement it in your project.

11.1.1 How to set application language

Setting application language is pretty simple. It can be done either via code like the following:

```
Yii::$app->language = 'ru_RU';
```

or via application config such as `config/main.php`:

```
return [
    // ...
    'language' => 'ru_RU',
];
```

Note that it should be done every request before any output in order for outputted content to be affected. Good places to consider are custom `UrlManager`, custom `UrlRule`, controller's or module's `beforeAction()` or application bootstrap.

11.1.2 Detecting language automatically

Detecting language automatically could help your application to conquer international markets if done properly. The following code shows selecting a language using information supplied by user's browser and a list of languages your application supports:

```
$supportedLanguages = ['en', 'ru'];
$languages = Yii::$app->request->getPreferredLanguage($supportedLanguages);
```

Note that language should be set prior to controller action so it's a good idea to create language selection component:

```
namespace app\components;
use yii\base\BootstrapInterface;
class LanguageSelector implements BootstrapInterface
{
    public $supportedLanguages = [];

    public function bootstrap($app)
    {
        $preferredLanguage = $app->request->getPreferredLanguage($this->
    supportedLanguages);
        $app->language = $preferredLanguage;
    }
}
```

In order to use the component you should specify it in the application config like the following:

```
return [
    'bootstrap' => [
        [
            'class' => 'app\components\LanguageSelector',
            'supportedLanguages' => ['en_US', 'ru_RU'],
        ],
    ],
    // ...
];
```

As was mentioned above, it could be implemented in custom `UrlManager`, custom `UrlRule` or controller's / module's `beforeAction()` instead.

11.1.3 Support selecting language manually

While it sounds like a great idea to always detect language, it's usually not enough. Detection could fail so user will get language he doesn't know, user may know many languages but prefer, for example, English for information about travelling. These problems could be solved by providing visible enough language selector that somehow remembers what was selected and uses it for the application further.

So the solution consists of three parts:

1. Language selector.
2. Storing what was selected.
3. Reusing what was stored.

Let's start with language selector widget. Overall it's a simple select widget pre-filled with an array of language code => language name pairs.

```
<?= Html::beginForm() ?>
<?= Html::dropDownList('language', Yii::$app->language, ['en-US' => 'English
    ', 'zh-CN' => 'Chinese']) ?>
<?= Html::submitButton('Change') ?>
<?= Html::endForm() ?>
```

Form handling should be done in controller. A good place to do it is `SiteController::actionLanguage`:

```
$language = Yii::$app->request->post('language');
Yii::$app->language = $language;

$languageCookie = new Cookie([
    'name' => 'language',
    'value' => $language,
    'expire' => time() + 60 * 60 * 24 * 30, // 30 days
]);
Yii::$app->response->cookies->add($languageCookie);
```

We're using cookie to store the language. But it could be, for example, database:

```
$user = Yii::$app->user;
$user->language = $language;
$user->save();
```

Now we can improve `LanguageSelector` a bit:

```
namespace app\components;
use yii\base\BootstrapInterface;

class LanguageSelector implements BootstrapInterface
{
    public $supportedLanguages = [];

    public function bootstrap($app)
    {
        $preferredLanguage = isset($app->request->cookies['language']) ? (
    string)$app->request->cookies['language'] : null;
        // or in case of database:
        // $preferredLanguage = $app->user->language;

        if (empty($preferredLanguage)) {
            $preferredLanguage = $app->request->getPreferredLanguage($this->
    supportedLanguages);
        }

        $app->language = $preferredLanguage;
    }
}
```

11.1.4 Language in URL / subdomain

So far we've found a way to detect language, select it manually and store it. For intranet applications and applications for which search engine indexing isn't important, it is already enough. For others you need to expose each application language to the world.

The best way to do it is to include language in the URL such as `http://example.com/ru/about` or subdomain such as `http://ru.example.com/about`.

The most straightforward implementation is about creating URL manager rules for each URL you have. In these rules you need to define language part i.e.:

```
'<language>/<page>' => 'site/page',
```

The con of this approach is that it is repetitive. You have to define it for all URLs you have and you have to put current language to parameters list each time you're creating an URL such as:

```
<?= Html::a('DE', ['post/view', 'language' => 'de']); ?>
```

Thanks to Yii we have an ability to replace default URL class with our own right from config file:

```
return [
    'components' => [
        'urlManager' => [
           'ruleConfig' => [
                'class' => 'app\components\LanguageUrlRule'
            ],
        ],
    ],
];
```

Here's what language aware URL rule class could look like:

```
class LanguageUrlRule extends UrlRule
{
    public function init()
    {
        if ($this->pattern !== null) {
            $this->pattern = '<language>/' . $this->pattern;
            // for subdomain it should be:
            // $this->pattern =  'http://<language>.example.com/' . $this->
    pattern,
        }
        $this->defaults['language'] = Yii::$app->language;
        parent::init();
    }
}
```

Ready to use extension

yii2-localeurls extension[1] implements reliable and quite customizable way of handling language in URLs.

11.2 Using IDs as translation source

Default way of translating content in Yii is to use English as a source language. It is quite convenient but there's another convenient way some developers prefer: using IDs such as `thank.you`.

11.2.1 How to do it

Using keys is very easy. In order to do it, specify `sourceLanguage` as `key` in your message bundle config and then set `forceTranslation` to `true`:

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                'sourceLanguage' => 'key', // <--- here
                'forceTranslation' => true, // <--- and here
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

Note that you have to provide translation for the application language (which is `en_US` by default) as well.

11.2.2 How it works

Setting `sourceLanguage` to `key` makes sure no language will ever match so translation will always happen no matter what.

[1] https://github.com/codemix/yii2-localeurls

Chapter 12

Performance

12.1 Implementing backgroud tasks (cronjobs)

There are at least two ways to run scheduled background tasks:

- Running console application command.
- Browser emulation.

Scheduling itself is the same for both ways. The difference is in actually running a command.

Under Linux and MacOS it's common to use cronjobs[1]. Windows users should check SchTasks or at[2].

12.1.1 Running console application command

Running console application command is preferred way to run code. First, implement Yii's console command[3]. Then add it to crontab file:

```
42 0 * * * php /path/to/yii.php hello/index
```

In the above `/path/to/yii` is full absolute path to `yii` console entry script. For both basic and advanced project templates it's right in the project root. Then follows usual command syntax.

`42 0 * * *` is crontab's configuration specifying when to run the command. Refer to cron docs and use crontab.guru[4] service to verify syntax.

Browser emulation

Browser emulation may be handy if you don't have access to local crontab, if you're triggering command externally or if you need the same environment as web application has.

[1] https://en.wikipedia.org/wiki/Cron
[2] http://technet.microsoft.com/en-us/library/cc725744.aspx
[3] http://www.yiiframework.com/doc-2.0/guide-tutorial-console.html
[4] http://crontab.guru/

The difference here is that instead of console controller you're putting your code into web controller and then fetching corresponding page via one of the following commands:

```
GET http://example.com/cron/
wget -O - http://example.com/cron/
lynx --dump http://example.com/cron/ >/dev/null
```

Note that you should take extra care about security since web controller is pretty much exposed. A good idea is to pass special key as GET parameter and check it in the code.

12.2 Running Yii 2.0 on HHVM

HHVM is an alternative PHP engine made by Facebook which is performing significantly better than current PHP 5.6 (and much better than PHP 5.5 or PHP 5.4). You can get 10–40for virtually any application. For processing-intensive ones it could be times faster than with usual Zend PHP.

12.2.1 Linux only

HHVM is linux only. It doesn't have anything for Windows and for MacOS it works in limited mode without JIT compiler.

12.2.2 Installing HHVM

Installing is easy. Here's how to do it for Debian:

```
sudo apt-key adv --recv-keys --keyserver hkp://keyserver.ubuntu.com:80 0
    x5a16e7281be7a449
echo deb http://dl.hhvm.com/debian wheezy main | sudo tee /etc/apt/sources.
    list.d/hhvm.list
sudo apt-get update
sudo apt-get install hhvm
```

Instructions for other distributions are available[5].

12.2.3 Nginx config

You can have both HHVM and php-fpm on the same server. Switching between them using nginx is easy since both are working as fastcgi. You can even have these side by side. In order to do it you should run regular PHP on one port and HHVM on another.

```
server {
    listen 80;
    root /path/to/your/www/root/goes/here;
```

[5] https://github.com/facebook/hhvm/wiki/Getting-Started

```
    index index.php;
    server_name hhvm.test.local;

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root/$fastcgi_script_name;
        fastcgi_pass   127.0.0.1:9001;
        try_files $uri =404;
    }
}

server {
    listen 80;
    root /path/to/your/www/root/goes/here;

    index index.php;
    server_name php.test.local;

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root/$fastcgi_script_name;
        fastcgi_pass   127.0.0.1:9000;
        try_files $uri =404;
    }
}
```

As you can see, configurations are identical except port number.

12.2.4 Test it first

HHVM is more or less tested to work with most frameworks out there[6]. Yii isn't exception. Except minor issues everything should work. Still, there are many incompatibilities compared to PHP[7] so make sure to test application well before going live.

12.2.5 Error reporting

HHVM behavior about errors is different than PHP one so by default you're getting nothing but a blank white screen instead of an error. Add the following to HHVM config to fix it:

```
hhvm.debug.server_error_message = true
```

[6] http://hhvm.com/frameworks/

[7] https://github.com/facebook/hhvm/labels/php5%20incompatibility

12.3 Caching

Yii supports different caching mechanisms. In this recipe we'll describe how to solve typical caching tasks.

12.3.1 PHP variables caching using APC extension

Configure cache component in Yii application config file:

```
'cache' => [
  'class' => 'yii\caching\ApcCache',
],
```

After that you can cache data by this way:

```
$key = 'cacheKey'
$data = Yii::$app->cache->get($key);

if ($data === false) {
    // $data is not found in cache, calculate it from scratch

    // store $data in cache so that it can be retrieved next time
    $cache->set($key, $data);
}
```

Error `Call to undefined function yii\caching\apc_fetch()` means that you have problems with APC extension. Refer PHP APC manual[8] for the details.

If cache doesn't work (`$data is always false`) check `apc.shm_size` property in `php.ini`. Probably your data size is more than allowed by this parameter.

12.3.2 HTTP caching for assets and other static resources

If expiration not specified for cacheable resources (`.js`, `.css`, .etc.) a speed of page loading process may be very slow. Such tool as `PageSpeed Insights for Chrome` determines `expiration not specified` problem as **crucial** for yii web page performance. It advices you to `Leverage browser caching`. You can do it by adding only one row to your application asset manager component:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'appendTimestamp' => true,
        ],
    ],
];
```

[8] http://php.net/manual/en/book.apc.php

12.4 Configuring a Yii2 Application for an Multiple Servers Stack

These instructions are focused on configuring Yii2 to be a stateless application. Stateless application means each of your servers should be the exact same copy as possible. They should not store data in the instance. Stateless instances means more scalable architecture. Current deployment method described is pretty basic. More efficient and complex deployment methods may be described in the future.

Generally in web development, scaling means horizontal scaling, adding more servers to handle more amount of traffics. This can be done manually or automatically in popular deployment platform. In autoscaled environment, the platform can detect large amount of traffics and handle it by temporarily adding additional servers.

To set up a scalable application, the application needs to be made stateless, generally nothing should be written directly to the application hosting server, so no local session or cache storage. The session, cache, and database needs to be hosted on dedicated server.

Setting up a Yii2 application for auto scaling is fairly straight forward:

12.4.1 Prerequisites

- Any well performing PaaS (Platform as a Service) solution that supports autoscaling, load balancing, and SQL databases. Examples of such PaaS are Google Cloud through its Instance Group and Load Balancer or Amazon Web Services using its AutoScaling Group and Elastic Load Balancer.
- Dedicated Redis or Memcached server. Easily launched on popular PaaS platforms with Bitnami Cloud[9]. Redis generally performs better over Memcached, so this page will be focusing on working with Redis.
- Dedicated database server (Most PaaS platforms let you easily launch one i.e. Google SQL or AWS Relational Database Service).

12.4.2 Making Your Application Stateless

Use a Yii2 supported Session Storage/Caching/Logging service such as Redis. Refer to the following resources for further instructions:

- Yii2 Class yii\redis\Session[10]
- Yii2 Class yii\redis\Cache[11]
- Yii2 Redis Logging Component[12]

[9] https://bitnami.com/cloud
[10] http://www.yiiframework.com/doc-2.0/yii-redis-session.html
[11] http://www.yiiframework.com/doc-2.0/yii-redis-cache.html
[12] https://github.com/JackyChan/yii2-redis-log

When you run on a PaaS platform, make sure to use the Redis server's internal IP and not the external IP. This is essential for your application's speed.

A redis server doesn't require much disk space. It runs on RAM. This guide recommends any new application to start with at least 1GB RAM, and vertically scaling up the instance (i.e. upgrade to a more RAM) depending on usage. You can measure your RAM usage by SSH'ing into the redis server and running `top`.

Also configure your application to use your hosted database server (hosted by i.e. Google SQL).

12.4.3 Configuring the Stack

The instructions below may be subject to change and improvement.

Set up a temporary single instance server to configure your application with.

The application must be deployed to the server by Git, so that multiple servers will stay up to date with the application. This guide recommends the following process:

- `git clone` the application into the configured `www` directory. The publicly accessed directory path can be varied depending on the platform.
- Set up a cron job to `git pull` the directory every minute.
- Set up a cron job to `composer install` the directory every minute.

When the application is up and running on the temporary server, create a snapshot of the server and use it to create your scalable server group.

Most PaaS platforms such as Google Cloud Managed Instance Groups and Amazon Elastic Beanstalk let you configure 'start up' commands. The start up command should also install/update the application (using `git clone` or `git pull` depending on if the service image already contains the application's git or not), and a `composer install` command to install all composer packages.

When the server group is set up using a disk based on the snapshot from the temporary server instance, you can remove the temporary server instance.

Your server group is now configured. Set up a load balancer on your PaaS platform (i.e. Load Balancer on Google) for the server group, andset your domain's A or CNAME records to your load balancer's static IP.

The mechanism described above is really simple yet sufficient enough to have your application up and running. There can be another process incorporated in the mechanism such as deployment failure prevention, version rollback, zero downtime, etc. These will be described in another topic.

There are couple of deployment mechanism that is provided by different platform such as Google Cloud or Amazon Web Services. These also will be described in another topic.

12.4.4 Assets Management

By default Yii generate assets on the fly and store in `web/assets` directory with names that depends on the file created time. If you deploy in multiple servers, the deployment time in each server can be different even by several seconds. This can be caused by the difference of latency to the code storage or other factors or especially in autoscaling environment when a new instance can be spun hours after last deployment.

This can cause inconsistencies for URLs generated by different servers for a single asset. Setting the server affinity in the load balancer can avoid this, meaning requests by same user will be directed to the very same server hit by the first request. But this solution is not recommended since you may cache the result of your page in a persistent centralized storage. Furthermore, in autoscaling environment underutilized server can be shut down anytime leaving the next requests served by different server that generated different URL.

A more robust solution is by configuring `hashCallback` in AssetManager[13] so it will not depend on time, rather an idempotent function.

For example, if you deploy your code in exact path in all servers, you can configure the `hashCallback` to something like

```
$config = [
    'components' => [
       'assetManager' => [
           'hashCallback' => function ($path) {
               return hash('md4', $path);
            }
       ]
    ]
];
```

If you use HTTP caching configuration in your Apache or NGINX server to serve assets like JS/CSS/JPG/etc, you may want to enable the `appendTimestamp` [14] so that when an asset gets updated the old asset will be invalidated in the cache.

```
$config = [
    'components' => [
       'assetManager' => [
           'appendTimestamp' => true,
           'hashCallback' => function ($path) {
               return hash('md4', $path);
            }
       ]
    ]
];
```

[13] http://www.yiiframework.com/doc-2.0/yii-web-assetmanager.html#%24hashCallback-detail

[14] http://www.yiiframework.com/doc-2.0/yii-web-assetmanager.html#%24appendTimestamp-detail

Load balancing without server affinity increases scalability but raises one more issue regarding the assets: assets availability in all servers. Consider request **1** for a page is being received by server **A**. Server **A** will generate assets and write them in local directory. Then the HTML output returned to the browser which then generates request **2** for the asset. If you configure the server affinity, this request will hit server **A** given the server is still available and the server will return the requested asset. But in this case the request may or may not hit server **A**. It can hit server **B** that still has not generated the asset in local directory thus returning `404 Not Found`. Server **B** will eventually generate the assets. The more servers you have the longer time they need to catch up with each others and that increases the number of `404 Not Found` for the assets.

The best thing you can do to avoid this is by using Asset Combination and Compression[15]. As described above, when you deploy your application, generally deployment platform can be set to execute hook such as `composer install`. Here you can also execute asset combination and compression using `yii asset assets.php config/assets-prod.php`. Just remember everytime you add new asset in your application, you need to add that asset in the `asset.php` configuration.

[15] http://www.yiiframework.com/doc-2.0/guide-structure-assets.html#combining-compressing-assets

Chapter 13

External code

13.1 Using Yii in third party apps

Legacy code. It happened to all of us. It's hard but we still need to deal with it. Would it be cool to gradually move from legacy towards Yii? Absolutely.

In this recipe you'll learn how to use Yii features in an existing PHP application.

13.1.1 How to do it

First of all, we need Yii itself. Since existing legacy application already takes care about routing, we don't need any application template. Let's start with `composer.json`. Either use existing one or create it:

```
{
    "name": "mysoft/mylegacyapp",
    "description": "Legacy app",
    "keywords": ["yii2", "legacy"],
    "homepage": "http://www.example.com/",
    "type": "project",
    "license": "Copyrighted",
    "minimum-stability": "dev",
    "require": {
        "php": ">=5.4.0",
        "yiisoft/yii2": "*"
    },
    "config": {
        "process-timeout": 1800
    },
    "extra": {
        "asset-installer-paths": {
            "npm-asset-library": "vendor/npm",
            "bower-asset-library": "vendor/bower"
        }
    }
}
```

Now run `composer install` and you should get Yii in `vendor` directory.

> **Note:** The dir should not be accessible from the web. Either keep it out of webroot or deny directory access.

Now create a file that will initialize Yii. Let's call it `yii_init.php`:

```
<?php
// set it to false when in production
defined('YII_DEBUG') or define('YII_DEBUG', true);

require(__DIR__ . '/vendor/autoload.php');
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');

$config = require(__DIR__ . '/config/yii.php');

new yii\web\Application($config);
```

Now create a config `config/yii.php`:

```
<?php

return [
    'id' => 'myapp',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
        'request' => [
            // !!! insert a secret key in the following (if it is empty) -
    this is required by cookie validation
            'cookieValidationKey' => '',
        ],
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'levels' => ['error', 'warning'],
                ],
            ],
        ],
    ],
    'params' => [],
];
```

That's it. You can require the file and mix Yii code into your legacy app:

```
// legacy code
```

```
$id = (int)$_GET['id'];


// new code
require 'path/to/yii_init.php';

$post = \app\models\Post::find()->where['id' => $id];

echo Html::encode($post->title);
```

13.1.2 How it works

In the `yii_init.php` we are including framework classes and composer autoloading. Important part there is that `->run()` method isn't called like it's done in normal Yii application. That means that we're skipping routing, running controller etc. Legacy app already doing it.

Chapter 14

Tools

14.1 IDE autocompletion for custom components

Using IDE for development is quite typical nowadays because of the comfort it provides. It detects typos and errors, suggests code improvements and, of course, provides code autocomplete. For Yii 2.0 it works quite good out of the box but not in case of custom application components i.e. `Yii::$app->mycomponent->something`.

14.1.1 Using custom Yii class

The best way to give IDE some hints is to use your own `Yii` file which isn't actually used when running code. This file could be named `Yii.php` and the content could be the following:

```
<?php
/**
 * Yii bootstrap file.
 * Used for enhanced IDE code autocompletion.
 */
class Yii extends \yii\BaseYii
{
    /**
     * @var BaseApplication|WebApplication|ConsoleApplication the
    application instance
     */
    public static $app;
}

/**
 * Class BaseApplication
 * Used for properties that are identical for both WebApplication and
    ConsoleApplication
 *
 * @property \app\components\RbacManager $authManager The auth manager for
    this application. Null is returned if auth manager is not configured.
    This property is read-only. Extended component.
```

```
 * @property \app\components\Mailer $mailer The mailer component. This
    property is read-only. Extended component.
 */
abstract class BaseApplication extends yii\base\Application
{
}

/**
 * Class WebApplication
 * Include only Web application related components here
 *
 * @property \app\components\User $user The user component. This property is
     read-only. Extended component.
 * @property \app\components\MyResponse $response The response component.
    This property is read-only. Extended component.
 * @property \app\components\ErrorHandler $errorHandler The error handler
    application component. This property is read-only. Extended component.
 */
class WebApplication extends yii\web\Application
{
}

/**
 * Class ConsoleApplication
 * Include only Console application related components here
 *
 * @property \app\components\ConsoleUser $user The user component. This
    property is read-only. Extended component.
 */
class ConsoleApplication extends yii\console\Application
{
}
```

In the above PHPDoc of `BaseApplication`, `WebApplication`, `ConsoleApplication` will be used by IDE to autocomplete your custom components described via `@property`.

> **Note**: To avoid "Multiple Implementations" PHPStorm warning and make autocomplete faster exclude or "Mark as Plain Text" `vendor/yiisoft/yii2/Yii.php` file.

That's it. Now `Yii::$app->user` will be our `\app\components\User` component instead of default one. The same applies for all other `@property`-declared components.

14.1.2 Custom Yii class autogeneration

You can generate custom `Yii` class automatically, using the components definitions from the application config. Check out bazilio91/yii2-stubs-generator[1] extention.

[1] https://github.com/bazilio91/yii2-stubs-generator

Customizing User component

In order to get autocompletion for User's Identity i.e. `Yii::$app->user->identity`, `app\components\User` class should look like the following:

```
<?php

namespace app\components;

use Yii;

/**
 * @inheritdoc
 *
 * @property \app\models\User|\yii\web\IdentityInterface|null $identity The
    identity object associated with the currently logged-in user. null is
    returned if the user is not logged in (not authenticated).
 */
class User extends \yii\web\User
{
}
```

As a result, Yii config file may look this way:

```
return [
    ...
    'components' => [
        /**
         * User
         */
        'user' => [
            'class' => 'app\components\User',
            'identityClass' => 'app\models\User',
        ],
        /**
         * Custom Component
         */
        'response' => [
            'class' => 'app\components\MyResponse',
        ],
    ],
];
```

14.2 Using custom migration template

For many cases it's useful to customize code that's created when you run `./yii migrate/create`. A good example is migrations for MySQL InnoDB where you need to specify engine.

14.2.1 How to do it

First of all, copy standard template i.e. `framework/views/migration.php` to your app. For example, to `views/migration.php`.

Then customize template:

```
<?php
/**
 * This view is used by console/controllers/MigrateController.php
 * The following variables are available in this view:
 */
/* @var $className string the new migration class name */

echo "<?php\n";
?>

use yii\db\Migration;

class <?= $className ?> extends Migration
{
    public function up()
    {
        $tableOptions = null;
        if ($this->db->driverName === 'mysql') {
            // http://stackoverflow.com/questions/766809/whats-the-
    difference-between-utf8-general-ci-and-utf8-unicode-ci
            $tableOptions = 'CHARACTER SET utf8 COLLATE utf8_unicode_ci
    ENGINE=InnoDB';
        }


    }

    public function down()
    {
        echo "<?= $className ?> cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Now in your console application config, such as `config/console.php` specify new template to use:

```
return [
...

    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'templateFile' => '@app/views/migration.php',
        ],
    ],

...
];
```

That's it. Now `./yii migrate/create` will use your template instead of standard one.