YII2 Introduction

Sílvio Priem Mendes, PhD

Cofinanciado por:

POCH PROGRAMA OPERACIONAL CAPITAL HUMANO

PORTUGAL 2020

UNIÃO EUROPEIA
Fundo Social Europeu

- YII2 Intro
- YII2 MVC Intro
  - The Model
  - The Controller
  - The View
- YII2 Application Structure
- YII2 Request Lifecycle
- Bootstrapping
- HTTP Requests
  - HTTP Request URLs
  - HTTP Request Methods
  - HTTP Headers
- HTTP Responses
  - HTTP Status Codes
  - HTTP Headers
  - HTTP Response Body
  - HTTP Redirect
  - Sending Files
  - Flushing a Response

TOC

- Yii is a high performance, component-based PHP framework for rapidly developing modern Web applications.
- The name Yii (pronounced Yee or [ji:]) means "simple and evolutionary" in Chinese.
- It can also be thought of as an acronym for **Yes It Is!**

YII2 intro

- **How does Yii Compare with Other Frameworks?**
- Like most PHP frameworks, Yii implements the MVC (Model-View-Controller) architectural pattern and promotes code organization based on that pattern.
- Yii is a full-stack framework providing many proven and ready-to-use features: query builders and ActiveRecord for both relational and NoSQL databases; RESTful API development support; multi-tier caching support;
- Yii is extremely extensible. You can customize or replace nearly every piece of the core's code.

- Yii applications are organized according to the model-view-controller (MVC) architectural pattern.
- Models represent data, business logic and rules;
- views are output representation of models; and
- controllers take input and convert it to commands for models and views.

YII2 MVC intro

- Besides MVC, Yii applications also have the following entities:

- **entry scripts:** they are PHP scripts that are directly accessible by end users. They are responsible for starting a request handling cycle.
- **applications:** they are globally accessible objects that manage application components and coordinate them to fulfill requests.
- **application components:** they are objects registered with applications and provide various services for fulfilling requests.
- **modules:** they are self-contained packages that contain complete MVC by themselves. An application can be organized in terms of multiple modules.
- **filters:** they represent code that need to be invoked before and after the actual handling of each request by controllers.
- **widgets:** they are objects that can be embedded in views. They may contain controller logic and can be reused in different views.

```php
<?php

namespace app\models;

use Yii;
use yii\base\Model;

class EntryForm extends Model
{
    public $name;
    public $email;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

Info: yii\base\Model is used as a parent for model classes *not* associated with database tables. yii\db\ActiveRecord is normally the parent for model classes that do correspond to database tables.

THE MODEL

- The **EntryForm** class contains two public members, name and email, which are used to store the data entered by the user.
- It also contains a method named **rules**(), which returns a set of rules for validating the data.
- The validation rules declared above state that:
  - both the name and email values are required
  - the email data must be a syntactically valid email address

THE MODEL

```php
<?php

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ...existing code...

    public function actionEntry()
    {
        $model = new EntryForm();

        if ($model->load(Yii::$app->request->post()) && $model->validate()) {
            // valid data received in $model

            // do something meaningful here about $model ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // either the page is initially displayed or there is some validation error
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

THE CONTROLLER

- The action first creates an **EntryForm** object.
- It then tries to populate the model with the data from $_POST, provided in Yii by **yii\web\Request::post()**.
- If the model is successfully populated (i.e., if the user has submitted the HTML form), the action will call **validate**() to make sure the values entered are valid.

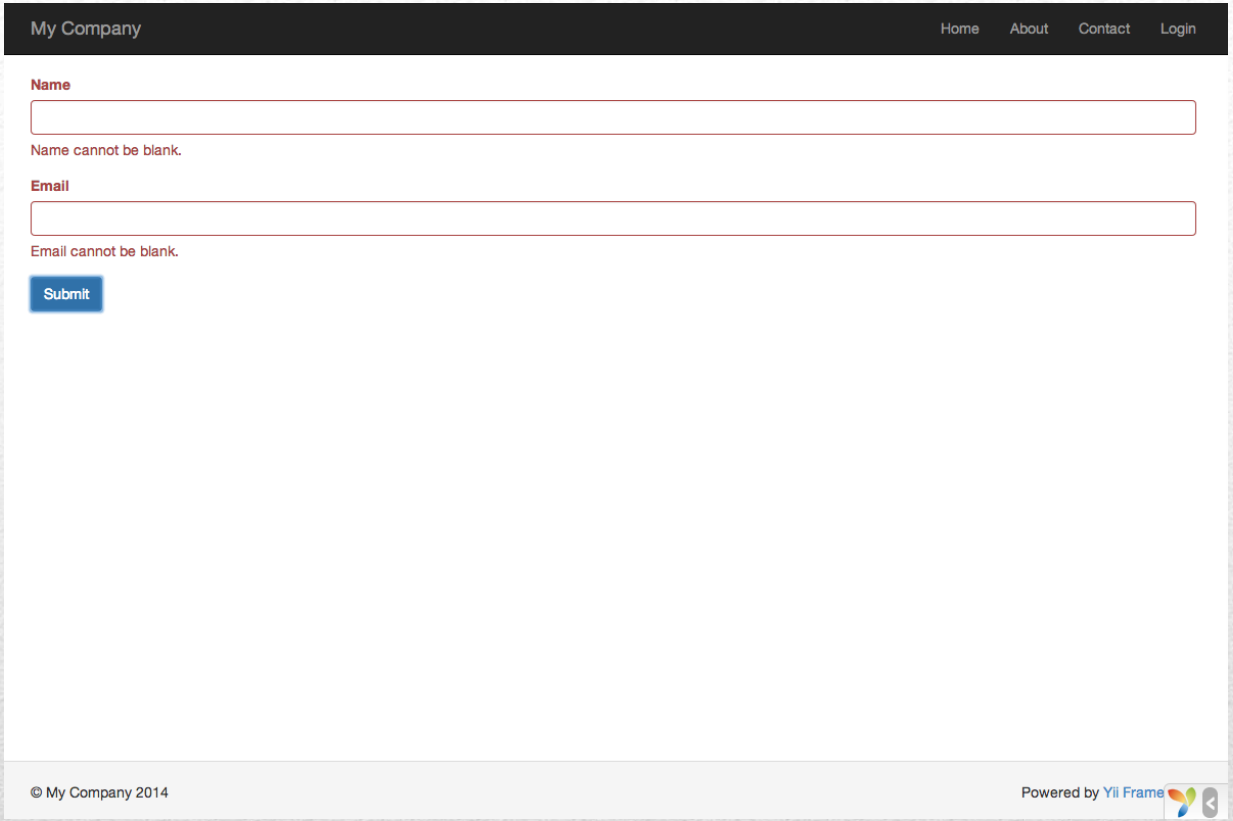
THE CONTROLLER

```php
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>

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

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

    <div class="form-group">
        <?= Html::submitButton('Submit', ['class' => 'btn btn-primary']) ?>
    </div>

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

THE VIEW (FORM)

- The view uses a powerful widget called **ActiveForm** to build the HTML form.
- The **begin**() and **end**() methods of the widget render the opening and closing form tags, respectively.
- Between the two method calls, input fields are created by the **field**() method.
- The first input field is for the "name" data, and the second for the "email" data.
- After the input fields, the **yii\helpers\Html::submitButton()** method is called to generate a submit button.

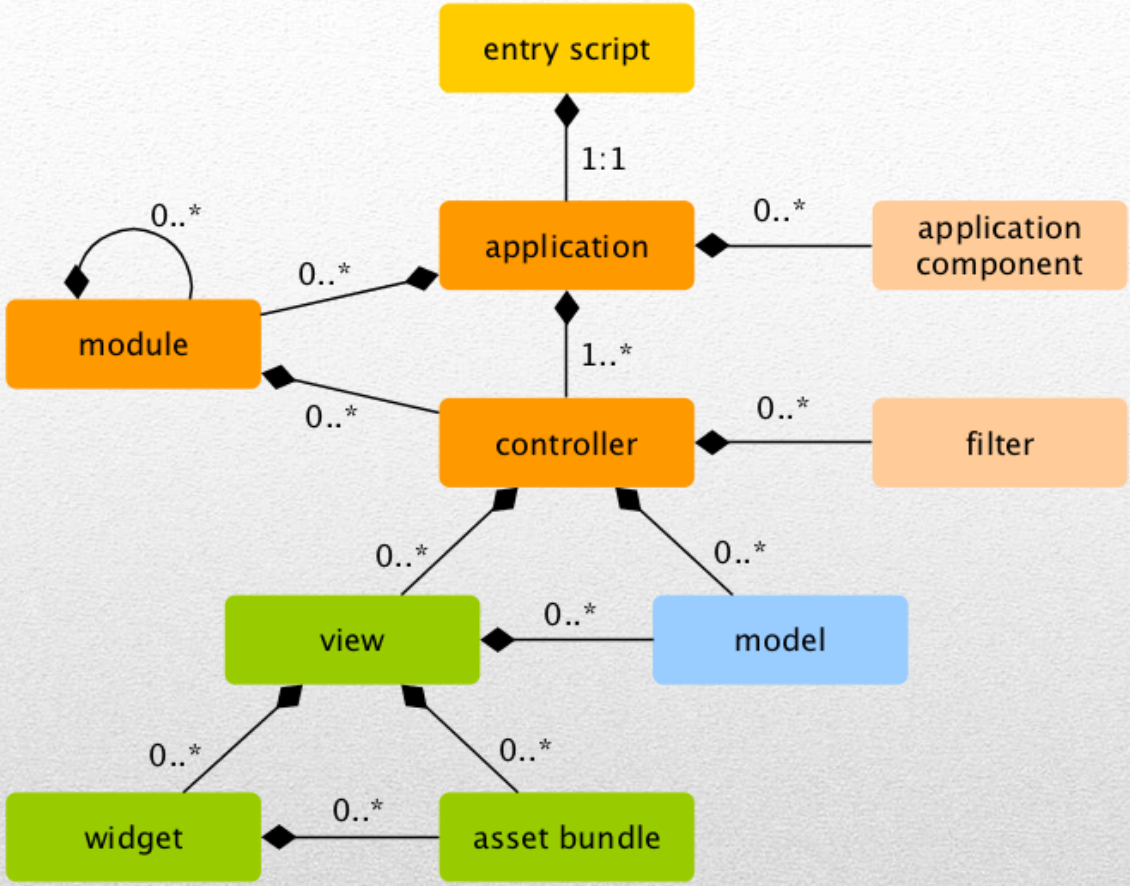
THE VIEW (FORM)



THE VIEW OUTPUT

- basic/ application base path
- composer.json used by Composer, describes package information
- config/ contains application and other configurations
- console.php the console application configuration
- web.php the Web application configuration
- commands/ contains console command classes
- controllers/ contains controller classes
- models/ contains model classes
- runtime/ contains files generated by Yii during runtime, such as logs and cache files
- vendor/ contains the installed Composer packages, including the Yii framework itself
- views/ contains view files
- web/ application Web root, contains Web accessible files
- assets/ contains published asset files (javascript and css) by Yii
- index.php the entry (or bootstrap) script for the application
- yii the Yii console command execution script

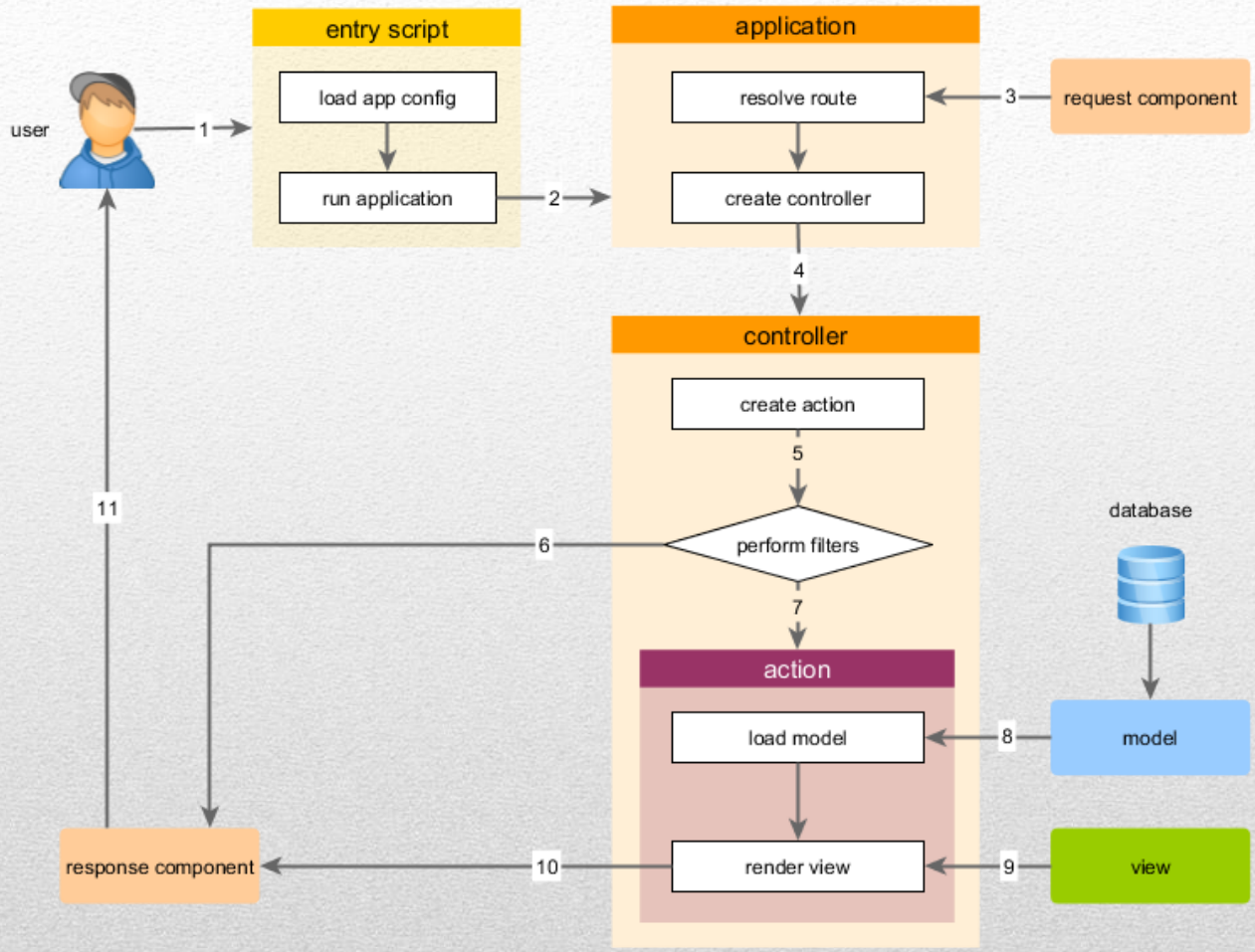
APPLICATION STRUCTURE



Static structure of an application.

- Each application has an entry script web/index.php which is the only Web accessible PHP script in the application.
- The entry script takes an incoming request and creates an application instance to handle it.
- The application resolves the request with the help of its components, and dispatches the request to the MVC elements.
- Widgets are used in the views to help build complex and dynamic user interface elements.

Static structure of an application.



REQUEST LIFECYCLE

1. A user makes a request to the entry script **web/index.php**.
2. The entry script loads the **application configuration** and creates an **application instance** to handle the request.
3. The application **resolves the requested route** with the help of the request application component.
4. The application **creates a controller instance** to handle the request.
5. The controller **creates an action instance** and performs the filters for the action.
6. If any filter fails, the action is cancelled.
7. If all filters pass, the action is executed.
8. The **action loads a data model**, possibly from a database.
9. The **action renders a view**, providing it with the data model.
10. The rendered result is returned to the response application component.
11. The response component sends the rendered result to the user's browser.

REQUEST LIFECYCLE

BOOTSTRAPPING

Bootstrapping refers to the process of preparing the environment before an application starts to resolve and process an incoming request.

Bootstrapping is done in two places: the entry script and the application.

In the entry script, class autoloaders for different libraries are registered.

This includes the Composer autoloader through its autoload.php file and the Yii autoloader through its Yii class file.

The entry script then loads the application configuration and creates an application instance.

In the constructor of the application, the following bootstrapping work is done:

- **preInit**() is called, which configures some high priority application properties, such as basePath.
- Register the error handler.
- Initialize application properties using the given application configuration.
- **init**() is called which in turn calls bootstrap() to run bootstrapping components.
- Include the extension manifest file vendor/yiisoft/extensions.php.
- Create and run bootstrap components declared by extensions.
- Create and run application components and/or modules that are declared in the application's bootstrap property.

Because the bootstrapping work has to be done before handling every request, it is very important to keep this process light and optimize it as much as possible.

BOOTSTRAPPING

- Try not to register too many bootstrapping components.
- A bootstrapping component is needed only if it wants to participate the whole life cycle of requesting handling.

- In production mode, enable a bytecode cache, such as PHP OPcache or APC, to minimize the time needed for including and parsing PHP files.

- Some large applications have very complex application configurations which are divided into many smaller configuration files.
  - If this is the case, consider caching the whole configuration array and loading it directly from cache before creating the application instance in the entry script.

BOOTSTRAPPING

HTTP REQUESTS

- Requests made to an application are represented in terms of yii\web\Request objects which provide information such as request parameters, HTTP headers, cookies, etc.

- For a given request, you can get access to the corresponding request object via the request application component which is an instance of **yii\web\Request**, by default.

HTTP REQUESTS

- To get request parameters, you can call get() and post() methods of the request component.
- They return the values of $_GET and $_POST, respectively.

```php
$request = Yii::$app->request;

$get = $request->get();
// equivalent to: $get = $_GET;

$id = $request->get('id');
// equivalent to: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// equivalent to: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// equivalent to: $post = $_POST;

$name = $request->post('name');
// equivalent to: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// equivalent to: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

HTTP REQUESTS

Info: Instead of directly accessing `$_GET` and `$_POST` to retrieve the request parameters, it is recommended that you get them via the `request` component as shown above. This will make writing tests easier because you can create a mock request component with faked request data.

- You can get the HTTP method used by the current request via the expression Yii::$app->request->method.
- A whole set of boolean properties is also provided for you to check if the current method is of certain type.

```
$request = Yii::$app->request;

if ($request->isAjax) { /* the request is an AJAX request */ }
if ($request->isGet)  { /* the request method is GET */ }
if ($request->isPost) { /* the request method is POST */ }
if ($request->isPut)  { /* the request method is PUT */ }
```

HTTP REQUEST METHODS

- Assuming the URL being requested is **http://example.com/admin/index.php/product?id=100**,

  You can get various parts of this URL as summarized in the following:

- **url**: returns **/admin/index.php/product?id=100**, which is the URL without the host info part.
- **absoluteUrl**: returns **http://example.com/admin/index.php/product?id=100**, which is the whole URL including the host info part.
- **hostInfo**: returns **http://example.com**, which is the host info part of the URL.

HTTP REQUEST URLs

- **hostInfo**: returns **http://example.com**, which is the host info part of the URL.
- **pathInfo**: returns **/product**, which is the part after the entry script and before the question mark (query string).
- **queryString**: returns **id=100**, which is the part after the question mark.
- **baseUrl**: returns **/admin**, which is the part after the host info and before the entry script name.
- **scriptUrl**: returns **/admin/index.php**, which is the URL without path info and query string.
- **serverName**: returns **example.com**, which is the host name in the URL.
- **serverPort**: returns **80**, which is the port used by the Web server.

HTTP REQUEST URLs

```php
// $headers is an object of yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// returns the Accept header value
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* there is User-Agent header */ }
```

- **The request component also provides support for quickly accessing some commonly used headers, including:**

- **userAgent: returns the value of the User-Agent header.**
- **contentType: returns the value of the Content-Type header which indicates the MIME type of the data in the request body.**

HTTP HEADERS

HTTP RESPONSES

- When an application finishes handling a request, it generates a response object and sends it to the end user.
- The response object contains information such as the HTTP status code, HTTP headers and body.

- In most cases you should mainly deal with the response application component which is an instance of **yii\web\Response**, by default.

- Yii also allows you to create your own response objects and send them to end users

HTTP STATUS CODE

- Example, to indicate the request is successfully handled, you may set the status code to be 200, like the following:
- **Yii::$app->response->statusCode = 200;**
- However, in most cases you do not need to explicitly set the status code because the default value of **yii\web\Response::$statusCode** is 200. And if you want to indicate the request is unsuccessful, you may throw an appropriate HTTP exception like the following:
- **throw new \yii\web\NotFoundHttpException;**

HTTP STATUS CODE

- When the error handler catches an exception, it will extract the status code from the exception and assign it to the response.
- For the **yii\web\NotFoundHttpException** above, it is associated with the **HTTP status 404**. The following HTTP exceptions are predefined in Yii:

- yii\web\BadRequestHttpException: status code 400.
- yii\web\ConflictHttpException: status code 409.
- yii\web\ForbiddenHttpException: status code 403.
- yii\web\GoneHttpException: status code 410.
- yii\web\MethodNotAllowedHttpException: status code 405.
- yii\web\NotAcceptableHttpException: status code 406.

- yii\web\NotAcceptableHttpException: status code 406.
- yii\web\NotFoundHttpException: status code 404.
- yii\web\ServerErrorHttpException: status code 500.
- yii\web\TooManyRequestsHttpException: status code 429.
- yii\web\UnauthorizedHttpException: status code 401.
- yii\web\UnsupportedMediaTypeHttpException: status code 415.

- If the exception that you want to throw is not among the above list, you may create one by extending from **yii\web\HttpException**, or directly throw it with a status code, for example,

- **throw new \yii\web\HttpException(402);**

HTTP STATUS CODE

- You can send HTTP headers by manipulating the header collection in the response component. For example,

```php
$headers = Yii::$app->response->headers;

// add a Pragma header. Existing Pragma headers will NOT be overwritten.
$headers->add('Pragma', 'no-cache');

// set a Pragma header. Any existing Pragma headers will be discarded.
$headers->set('Pragma', 'no-cache');

// remove Pragma header(s) and return the removed Pragma header values in an array
$values = $headers->remove('Pragma');
```

HTTP HEADERS

HTTP RESPONSE BODY

- Most responses should have a body which gives the content that you want to show to end users (usually with valid HTML).
- If you already have a formatted body string, you may assign it to the yii\web\Response::$content property of the response. For example,
- **Yii::$app->response->content = 'hello world!';**
- If your data needs to be formatted before sending it to end users, you should set both of the format and data properties. The format property specifies in which format the data should be formatted. For example,
- **$response = Yii::$app->response;**
- **$response->format = \yii\web\Response::FORMAT_JSON;**
- **$response->data = ['message' => 'hello world'];**

HTTP RESPONSE BODY

- Yii supports the following formats out of the box, each implemented by a formatter class. You can customize these formatters or add new ones by configuring the yii\web\Response::$formatters property.

- HTML: implemented by yii\web\HtmlResponseFormatter.
- XML: implemented by yii\web\XmlResponseFormatter.
- JSON: implemented by yii\web\JsonResponseFormatter.
- JSONP: implemented by yii\web\JsonResponseFormatter.
- RAW: use this format if you want to send the response directly without applying any formatting.

- While the response body can be set explicitly as shown above, in most cases you may set it implicitly by the return value of action methods. A common use case is like the following:

```
public function actionIndex()
{
    return $this->render('index');
}
```

- The index action above returns the rendering result of the index view. The return value will be taken by the response component, formatted and then sent to end users.

- Because by default the response format is HTML, you should only return a string in an action method.

HTTP RESPONSE BODY

- If you want to use a different response format, you should set it first before returning the data. For example,

```php
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    return [
        'message' => 'hello world',
        'code' => 100,
    ];
}
```

HTTP RESPONSE BODY

- Browser redirection relies on sending a Location HTTP header. Because this feature is commonly used, Yii provides some special support for it.

- You can redirect the user browser to a URL by calling the **yii\web\Response::redirect()** method.
- The method sets the appropriate Location header with the given URL and returns the response object itself.
- In an action method, you can call its shortcut version **yii\web\Controller::redirect()**.

```php
public function actionOld()
{
    return $this->redirect('http://example.com/new', 301);
}
```


Info: By default, the yii\web\Response::redirect() method sets the response status code to be 302 which instructs the browser that the resource being requested is *temporarily* located in a different URI. You can pass in a status code 301 to tell the browser that the resource has been *permanently* relocated.

HTTP REDIRECT

- Like browser redirection, file sending is another feature that relies on specific HTTP headers. Yii provides a set of methods to support various file sending needs. They all have built-in support for the HTTP range header.

- **yii\web\Response::sendFile**(): sends an existing file to a client.
- **yii\web\Response::sendContentAsFile**(): sends a text string as a file to a client.
- **yii\web\Response::sendStreamAsFile**(): sends an existing file stream as a file to a client.
  - These methods have the same method signature with the response object as the return value. If the file to be sent is very big, you should consider using yii\web\Response::sendStreamAsFile() because it is more memory efficient.

RESPONSE - SENDING FILES

```php
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

- If you are calling the file sending method in places other than an action method, you should also call the yii\web\Response::send() method afterwards to ensure no extra content will be appended to the response.

```php
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

RESPONSE - SENDING FILES

FLUSHING A RESPONSE

- **The content in a response is not sent to the user until the yii\web\Response::send() method is called.**
- By default, this method will be called automatically at the end of yii\base\Application::run(). You can, however, explicitly call this method to force sending out the response immediately.
- The yii\web\Response::send() method takes the following steps to send out a response:
  1. Trigger the yii\web\Response::EVENT_BEFORE_SEND event.
  2. Call yii\web\Response::prepare() to format response data into response content.
  3. Trigger the yii\web\Response::EVENT_AFTER_PREPARE event.
  4. Call yii\web\Response::sendHeaders() to send out the registered HTTP headers.
  5. Call yii\web\Response::sendContent() to send out the response body content.
  6. Trigger the yii\web\Response::EVENT_AFTER_SEND event.