Opleiding Applicatie Ontwikkelaar

A4 - Leerlijn PHP

MVC

Auteur: *Harmen Steenbergen*

Datum: *12-2-2020*

Crebo: *25187*

Versie: *1.1*

Inhoudsopgave

Overzicht

Level: Domein A Level 4
Duur: 4 weken
Methode: Weekplanning

Voorkennis

Niet van toepassing.

Materialen

- Je laptop met;
- Editor, bijvoorbeeld Kladblok, Notepad++, Atom (☺ : aanrader!)
- Webbrowser, bijvoorbeeld Internet Explorer, Firefox, Chrome

Bronnen

- Zie bijlage Bronnen

Instructies

- Hoe te starten met HTML, wat moet je installeren, mappen structuur.
- Zoeken in de referenties van HTML.
- Docenten zullen verschillende colleges verzorgen over HTML en CSS.

Beschrijving

In deze opdracht leer je HTML5 gebruiken voor het indelen en CSS3 voor het opmaken van een webpagina. Eerst leer je de twee technieken vervolgens gebruik je deze bij het maken van de eindopdracht.

Doelen

Na deze periode zal je in staat zijn om een statische website in HTML en CSS te kunnen maken.

Beoordeling

Deze opdracht wordt beoordeeld aan de hand van de eindopdracht en een gesprek over het eindproduct.

MVC structuur verwerken in PHP

Als een project voor een website met PHP groter wordt zal je merken dat het steeds lastiger wordt de code netjes te houden. Een kleine verandering kan al veel werk opleveren of het oplossen van een fout in de code zorgt voor een nieuw fout.

Om dit te voorkomen moet je nadenken over de structuur van de code;

- welke mappen maak je en waar sla je welke bestanden op?
- Hoe deel je lange code op in kleinere delen?
- Hoe haal je zo veel mogelijk HTML, CSS en PHP uit elkaar?
- Hoe voorkom je zo veel mogelijk dubbele code?

Het werken met het MVC-structuur kan hierbij helpen. In deze module gaan we een deel van het MVC-structuur gebruiken om de code in te delen.

Wat vooraf ging?

In eerdere modules heb je geleerd:

- A1 - Hoe je moet programmeren in PHP,
  Wat is de syntax, hoe moet je programmeerstructuren als voorwaardelijke en herhaalde code gebruiken. Hoe kun je variabelen en arrays gebruiken om data (tijdelijk) op te slaan?
- A2 - Hoe kun je code beter indelen,
  Het gebruik van functies en/of methoden om delen van je code apart op te slaan en vaker te kunnen gebruiken. Meerder bronbestanden gebruiken en laten samenwerken.
- A3 - Basis OOP,
  Object Oriented Programming
  In plaats van één lang bronbestand gebruik maken van Klassen en Objecten om je code op te bouwen uit verschillende bronbestanden.

Binnen deze module heb je al deze technieken weer nodig, zeker OOP. Alle onderdelen van het MVC-structuur worden opgebouwd doormiddel van klassen en objecten.

Wat is MVC?

MVC is een structuur die je kan helpen bij het indelen van je code. MVC staat voor Model, View en Controller. Je gaat de onderdelen van je programma maken door deze drie onderdelen te maken. Elk onderdeel heeft zijn eigen functie en ze werken samen volgens een vaste structuur.

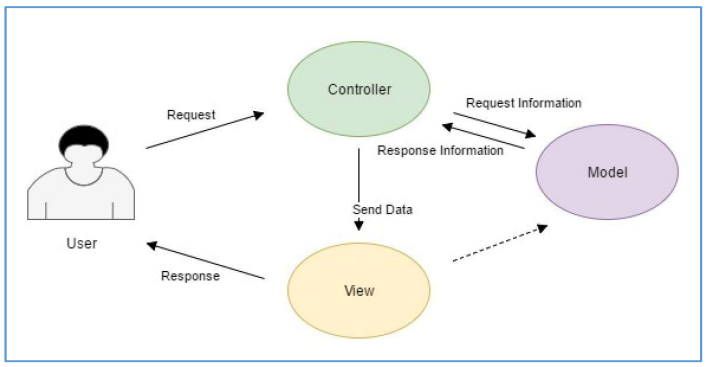
- **Model**: GEGEVENS: zorgt voor het verwerken van de gegevens. Zal ook verbinding maken met de database om gegevens op te halen of op te slaan. Ook het controleren van gegevens wordt door het Model gedaan.
- **View**: WEERGAVE: zorgt voor het weergave. Alles wat op een scherm of papier komt wordt door de View geregeld. De View krijgt gegevens van de Controller of het Model om die in de weergave te verwerken. Een View in een PHP-project zal voornamelijk zorgen voor het opbouwen van de HTML-pagina.

- **Controller**: VERWERKEN: ontvangt gegevens van het Model stuurt die door naar een View of een Model. Ook handelt de Controller gebeurtenissen (Events) af. Bijvoorbeeld gegevens uit een HTML-formulier komen bij de Controller binnen en worden ingelezen en doorgestuurd naar een Model voor verwerking.

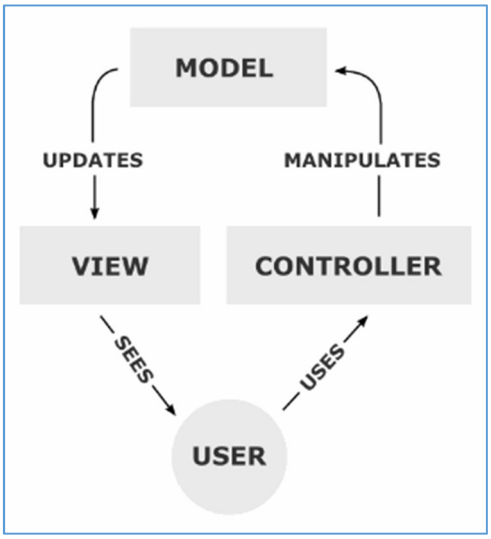
Wat is MVC niet?

MVC is geen syntax, protocol of afgesproken conventie. Dit betekent er niet één manier om MVC te gebruiken en in de loop van de tijd is MVC op verschillende manier uitgewerkt. Dit zorgt ook voor dat als je op internet gaat lezen over MVC je verschillende verhalen tegenkomt. Dat wil niet zeggen dat een of de ander goed is, het zijn gewoon verschillende manier om MVC te gebruiken.

Hieronder zie je twee schema's die als snel zal vinden als je op internet zoekt op MVC en PHP:



Figuur 1 https://medium.com/@noufel.gouirhate/create-your-own-mvc-framework-in-php-af7bd1f0ca19



Figuur 2 https://www.sitepoint.com/the-mvc-pattern-and-php-1

In beide figuren zie je de Model, Controller en View, maar de pijlen en de richting van de pijlen verschillen. Zoek maar eens op Google naar afbeelding met de zoekterm MVC. Je zult veel verschillende modellen zien, het idee is overal hetzelfde, de uitwerking anders.

Voor deze model heb ik veel gebruikgemaakt van https://www.sitepoint.com/the-mvc-pattern-and-php-1 en wordt MVC opgebouwd volgens de rechter afbeelding.

Gebruik van MVC

Je kunt MVC gebruiken voor je eigen project door het principe van MVC in je code te gebruiken. Bij het indelen van de mappen en bestanden van je project houdt er dan rekening mee dat je voor de verschillende onderdelen van je project Models, Controllers en Views gaat gebruiken. Hoe je dat doet kun je zelf bedenken of een voorbeeld van een ander overnemen. Zoals al gezegd, MVC is geen exacte manier van werking maar een principe dat je kunt gebruiken.

A4: MVC IN PHP

Daarnaast zijn er veel PHP-frameworks die ook gebruikmaken van MVC, zoals:

- Laravel
- Code Igniter
- Symphony
- Zend
- En veel meer

Deze frameworks zijn in PHP gemaakte oplossingen voor het maken van een website. Veel onderdelen van de website zijn al voorbereid en de MVC-structuur is al uitgewerkt. Dit betekent dat je in zo'n framework een website opbouwt door eigen Views, Controllers en Models te maken volgens de manier dei in de handleiding van het framework staat. Ook daarvoor geldt dat de manier waarop MVC is uitgewerkt verschillend kan zijn voor de verschillende frameworks.

MVC is dus een manier van denken over code en niet een set van coderegels. Binnen deze module gaan we een eigen MVC opbouwen om te leren begrijpen wat MVC is.

MVP en MVVM	Naast het MVC-model zijn er nog anders modellen die er veel op lijken zoals MCP en MVVM. MVP staat voor Model, View, Presenter en MVVM staat voor Model, View, View Model. Op internet vind je uitleg en veel discussies over de verschillen tussen de modellen. Uit de discussies blijkt ook al dat de verschillen niet altijd duidelijk zijn en soms een kwestie van (persoonlijke) interpretatie.

De basis structuur

De uitwerking die we gaan bouwen wordt een verzameling van verschillende klassen voor de onderdelen. We gaan dus OOP gebruiken. Daarnaast wil je bestanden op de webserver (html/css/javascript en afbeeldingen) scheiden van de code.

Voor deze module heb ik een structuur opgezet om mee te beginnen waarbij zo veel mogelijk verschillende bestanden worden gescheiden in mappen.

Opdracht 1. Voorbereiden

Je kunt de structuur ophalen van GitHub.

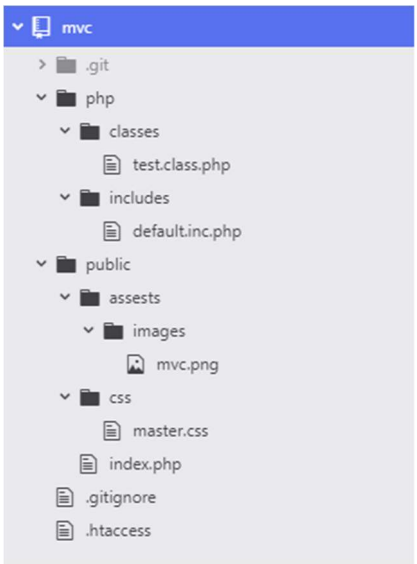
- Maak onder de DOCUMENT_ROOT van je webserver een nieuwe map voor dit project. Kies zelf een duidelijke naam.
- Ga op de commandprompt naar deze map en typ:

```
git clone https://github.com/DrentheCollege/mvc.git
```

Je kunt natuurlijk je favoriete git-toepassingen gebruiken om de kloon te maken, bijvoorbeeld GitHub Desktop, TortoiseGit of GitKraken.

De structuur uitgelegd

Je hebt nu in de gemaakte map de volgende mappen en bestanden gekregen:



Op het hoogte niveau zie je twee mappen en een bestand.

- Mappen:
  - php alle bronbestanden voor de php-code. Buiten deze map zal je weinig php tegenkomen.

- public alle bestanden die via de webserver benaderbaar moeten zijn, zoals html, css en javascript bestanden.

- Bestanden
  - .htaccess in dit bestand kun je (configuratie-)regels voor de webserver opnemen die allen gelden voor de map en onderliggende mappen waarin het bestand staat gelden. In dit bestand staan regels voor de Rewrite-engine die een aanvraag voor een pagina op de server veranderd.

```
1  #rewriting request to the public folder
2
3  RewriteEngine On
4
5  RewriteCond %{THE_REQUEST} /public/([^\s?]*) [NC]
6  RewriteRule ^ %1 [L,NE,R=302]
7
8  RewriteRule ^((?!public/).*)$ public/$1 [QSA,L,NC]
```

Het schrijven van regels voor de Rewrite-engine van Apache is een lastig en vraagt vaak veel gepuzzel om goed te krijgen.
Deze regels zorgen er voor dat een aanvraag op de server wordt omgeleid naar de map *public*. Later ga je dit bestand aanpassen om aanvragen om te leiden naar de *index.php*.

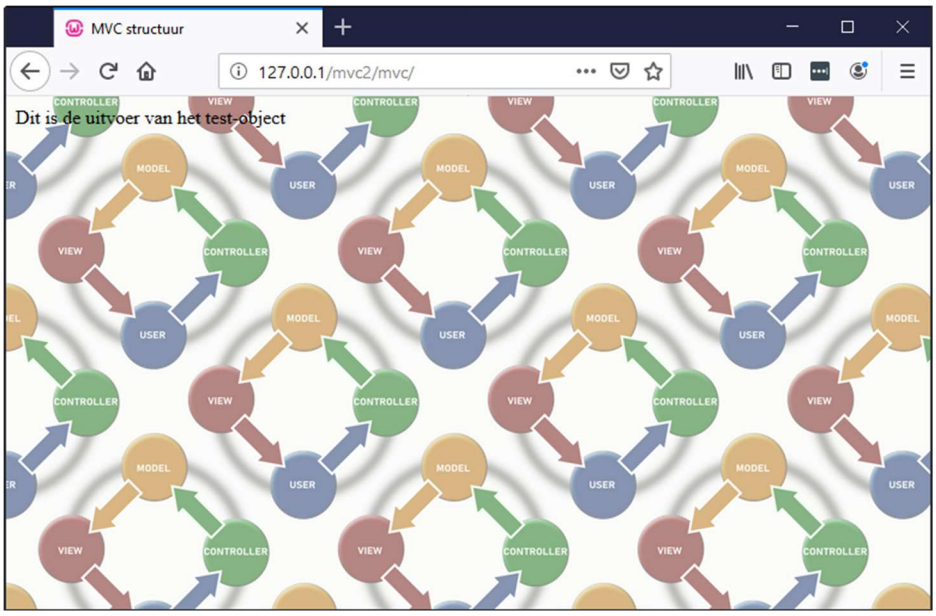
A4: MVC IN PHP

Opdracht 2. Proberen

- Start je webserver (XAMP , WAMP, etc) als dit nog niet gestart is.
- Bekijk de pagina in de browser:

```
http://127.0.0.1/<jouw map>/mvc/
```

Als het goed is gegaan zie je de volgende pagina.



Als je deze pagina niet ziet ga dan je stappen na. Lukt het niet om de pagina in beeld te krijgen vraag dan een docent om je te helpen voor je verder gaat.

Opdracht 3. Zoeken

- Zoek door de mappen en bestanden en probeer de code te volgen/lezen.

Aanvragen afvangen

Bij het gebruik van MVC wordt vaak een router gebruikt. Deze router zorgt er voor dat een aanvraag op de webserver wordt doorgestuurd naar een bepaalde controller, view en/of model. Hij bepaald dus de route na een aanvraag.

Voordat we dit gaan maken, gaan we eerst de .htaccess aanpassen zodat alle aanvragen op de webserver worden doorgestuurd naar de index.php en extra gegevens van de aanvraag worden doorgegeven aan de index.php.

.htaccess	Het bestand .htaccess kan je in een map van je apache webserver zetten om instellingen van de apache-server voor die map, en de mappen daaronder, aan te passen. Onder andere kun je hiermee regels maken die de aanvraag voor een pagina op de webserver aanpassen. Bijvoorbeeld kun je aanvraag doorsturen naar een andere pagina. Dit zijn de rewite-rules. Daar gaan we hier gebruik van maken.

Opdracht 4. .htacces aanpassen

- Voeg de volgende twee regels toe aan het bestand *.htaccess*:

```
10  RewriteCond ${REQUEST_URI} !\.(?:css|js|jpe?g|gif|png)$ [NC]
11  RewriteRule ^(public\/)([a-zA-Z0-9\-\ \/]*)$ index.php?route=$2 [QSA]
```

- Pas het bestand *index.php* in de map *public* aan door de php-code in de body-tag vervangen door:

```
10      <?php
11       if(isset($_GET['route'])){
12         echo $_GET['route'];
13       }
14       ?>
```

Nu wordt de aanvraag op de webserver door middel van het bestand *.htaccess* doorgestuurd naar *index.php* in de map *public*. De php-code zorgt er voor dat meegegeven waarde p in beeld komt op de index-pagina.

- Probeer verschillende pagina's op de vragen op je webserver in de browser:

http://127.0.0.1/<jouw map>/mvc/home
http://127.0.0.1/<jouw map>/mvc/about
http://127.0.0.1/<jouw map>/mvc/person/id/4
http://127.0.0.1/<jouw map>/mvc/address/add

Als alles goed is gegaan krijg je de volgende teksten na elkaar in beeld:

- home
- about
- person/id/4

- address/add

Nu kunnen we het MVC-model opbouwen door de verschillende onderdelen een voor een te gaan maken, testen en uitbreiden.

Het bouwen van de MVC

Voor dat we aan een view, controller of model beginnen, gaan we eerst een klasse maken die alles aan elkaar verbind. Dan kunnen we uiteindelijke in de *index.php* deze klasse gebruiken om de applicatie te starten en de rest van de MVC-objecten te maken en af te handelen.

Opdracht 5. App maken

- Om alles bij elkaar te houden maken in de map *php/classes* een nieuw map met de naam *mvc*

namespaces	In deze map gaan we alle klassen voor het mvc-model opslaan. Dankzij het gebruik van *namespaces* worden de klassen automatische gevonden. Als je *namespaces* gebruikt moet je wel een autoload-functie voor de klassen maken die *namespaces* ondersteund. De functie in php/inlcudes/default.inc.php houdt rekening met de namespaces.

- In deze nieuwe map maak je een nieuw bestand met de naam app.class.php
- Neem in dat bestand de volgende code over

```php
<?php
  namespace mvc;

  class App{

    public function __toString(){
        return "this is app";
    }

  }
 ?>
```

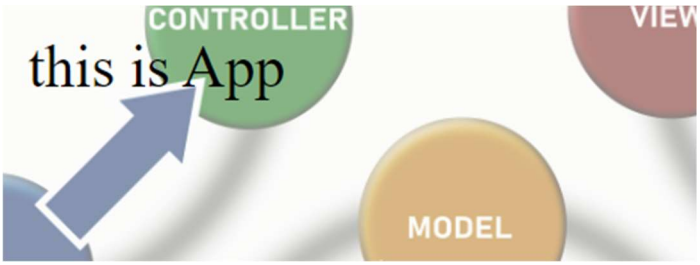
De functienaam __toString is in php een speciale naam voor een function in een klasse. Deze functie wordt automatisch gebruikt wanneer een object van deze klasse op een plek wordt gebruikt waar een tekenreeks (String) wordt verwacht, zoals we nu gaan doen in de index.php achter het echo-commando.

- Verander de php-code in index.php in

```
10      <?php
11          $app = new \mvc\App();
12          echo $app;
13      ?>
```

In regel 12 wordt het object $app gebruikt als tekenreeks en wordt automatisch de functie __toString aangeroepen.

- In je browser moet het volgende te zien krijgen:



Een basis View

De gemaakt app voert nu een simpele tekst uit voor het testen. De uitvoer van de app moet uit een view komen. De uiteindelijke applicatie zal bestaan uit meerdere views, controllers en models. Als je meerdere views maakt wil je wel dat ze ongeveer gelijk werken. Daarom gaan eerst een basis view maken en een eerste test-view daarop baseren. We krijgen dan:

- Een view-interface; beschrijft wat een view minimaal moet hebben maar bevat geen code. Je weet wel zeker dat elke view die gebaseerd is op de view-interface bepaalde methodes bevat. Dit geeft een structuur voor alle views.
- Een abstracte view; dit is een klasse waarop onze eigen views worden gebaseerd. Hierin staat gedeelde code die voor elke view op dezelfde manier werkt. Alle views kunnen de methodes uit deze klasse gebruiken.
- Een eigen view; deze wordt gebruikt om iets op het scherm te zetten. Alle code van de abstracte view werkt ook voor deze view. Alle methoden/functies die in de view-interface zijn vastgelegd moeten worden gemaakt.

Opdracht 6. View opbouwen

- Maak in de map php/classes/mvc een nieuw map met de naam interfaces.

- Maak in deze map een bestand met de naam view.class.php
- Neem in dit bestand de volgende code over:

A4: MVC IN PHP

```php
<?php
  namespace mvc\interfaces;

  interface View {
    public function getHTML();
  }
?>
```

interface

Een interface is een speciale klasse waarin je alleen de structuur van een klasse kunt bepalen. Een interface bevat geen uitvoerbare code. Een interface kun je in een andere klasse implementeren. In de nieuwe klasse ben je dan verplicht de structuur van de interface over te nemen en voor de aangegeven methodes code te schrijven. Daarmee verplicht je dat elke implementatie van de interface bepaalde methodes bevat. In het voorbeeld moet elke implementatie van de interface een methode getHTML bevatten.

- Maak in de map **php/classes/mvc** een nieuw bestand met de naam **view.class.php**
- Neem in dit bestand de volgende code over:

```php
<?php
  namespace mvc;

  abstract class View implements \mvc\interfaces\View {

    protected $controller;
    protected $model;

    public function __construct($controller, $model){
      $this->controller = $controller;
      $this->model = $model;
    }

  }
?>
```

Abstracte klasse

Een abstracte klasse is een klasse waarvan geen object gemaakt kan worden. Het is een klasse die als basis gaat dienen voor andere klasses die de code uit de abstracte klasse delen. Omdat je van een abstracte klasse nog geen object kunt maken hoef je de methodes uit de interface hier nog niet de implementeren.

- Maak in de map **php/classes** een nieuwe map aan met de naam **views**
- Maak in deze map een nieuw bestand aan met de naam **testview.class.php**

A4: MVC IN PHP

- Neem de volgende code over in dit bestand:

```php
<?php
  namespace views;

  class TestView extends \mvc\View {
    public function getHTML(){
      return "Dit is een test view"
    }
  }
 ?>
```

Om deze nieuwe test view te gaan testen gaan de app aanpassen.

- Verander de code in **app.class.php** in:

```php
<?php
  namespace mvc;

  class App{
    private $view;

    public function __construct(){
      $this->view = new \views\testView(null, null);
    }

    public function __toString(){
        return $this->view->getHTML();
    }

  }
?>
```

Als je het resultaat in de browser bekijkt moet je de tekst “dit is de test view” te zien krijgen.

Een basis Controller

De volgende stap is het maken van een controller. De controller handelt de invoer van een gebruiker af. Dat zou dus de invoer in een HTML-formulier of het klikken op een bepaalde link kunnen zijn. De controller gaan we op dezelfde manier maken als de view, dus een interface, een abstracte klasse en eigen (verschillende) controllers gebaseerd op de abstracte controller klasse.

Opdracht 7. Controller bouwen

- Maak in de map **/php/classes/mvc/interfaces** een nieuw bestand met de naam **controller.class.php**.

A4: MVC IN PHP

- Neem de volgende code over in het bestand:

```php
<?php
  namespace mvc\interfaces;

  interface Controller {
  }
?>
```

- Maak in de map **/php/classes/mvc** een nieuw bestand met de naam **controller.class.php**
- Neem de volgende code over in het nieuwe bestand:

```php
<?php
  namespace mvc;

  abstract class Controller implements \mvc\interfaces\Controller {
    private $model;

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

    public function getPostData($name){
      return $this->getData($name);
    }

    public function getGetData($name){
        return  $this->getData($name, "GET");
    }

    public function getData($name, $type = "POST"){
      $dataSource = ($type == "POST"? $_POST : $_GET);
      return isset($dataSource[$name]) ? $dataSource[$name] : null;
    }
  }
?>
```

- Maak in de map **php/classes** een nieuwe map met de naam **controllers**
- Maak in deze nieuwe map een bestand met de naam **testcontroller.class.php**
- Neem de volgende code over in dat bestand:

```php
<?php
  namespace controllers;

  class TestController extends \mvc\Controller {

  }
?>
```

A4: MVC IN PHP

De TestController bevat verder geen code. Alles wat de controller moet doen wordt al door de abstracte klasse Controller gedaan.

- Pas de code in **app.class.php** aan zodat de nieuwe controller wordt gebruikt:

```php
<?php
  namespace mvc;

  class App{
    private $view;
    private $controller;

    public function __construct(){
      $this->controller = new \controllers\TestController(null);
      $this->view = new \views\testView($this->controller, null);
    }

    public function __toString(){
        return $this->view->getHTML();
    }
  }
?>
```

- Controleer het resultaat in de browser.
  Als het goed is, is er niets in de uitvoer veranderd omdat de controller nog niets doet. Je krijgt dus alleen de uitvoer van de test-view.
  Om invoer te krijgen die de controller kan verwerken gaan we de test-view aanpassen.
- Pas de code van **testview.class.php** aan:

```php
<?php
  namespace views;

  class TestView extends \mvc\View {
    public function getHTML(){
      $klik = $this->controller->getGetData("klik");
      $output = "";
      $output .= "<h1>" . $klik++ . "</h1>";
      $output .= "<a href=\"test?klik=$klik\">klik<a>";
      $output .= "<br>";
      return $output;
    }
  }
?>
```

De view laat een hyperlink zien die gekoppeld is aan dezelfde pagina waarbij een parameter klik met een waarde wordt meegegeven. De meegegeven waarde van klik wordt door de controller uitgelezen en in de view gebruikt.

Meerdere views en controller maken

Tot nu toe hebben we één View en één Controller die door de app worden gebruikt. Een applicatie opgebouwd in een mvc-model zal bestaan uit meerdere views, controllers en models. Om de verschillende elementen te kunnen aanroepen maken we gebruik van een router. De router gaat een variabele uitlezen waarin de mogelijk routes staan. Elke route is gekoppeld aan een view, een controller en een model.

De router gaat bestaan uit een bestand met de instellingen en een klasse die de juiste route uitleest en gebruikt.

Opdracht 8. Een router maken

- Maak in de map **/php/** een nieuw bestand met de naam **routes.php**.
- Neem de volgende code over het nieuwe bestand.

```php
<?php
  define("DEFAULT_ROUTE", "home");

  $routes = array(
    "home" => array(
      "view" => "HomeView",
      "controller" => "HomeController",
    ),
    "form" => array(
      "view" => "FormView",
      "controller" => "FormController",
    ),
    "test" => array(
      "view" => "TestView",
      "controller" => "TestController",
    ),
  );
 ?>
```

- Maak in de map **php/classes/mcv/** een nieuw bestand aan met de naam **router.class.php**.
- Neem de volgende code over in het nieuwe bestand.

A4: MVC IN PHP

```php
<?php
  namespace mvc;

  class Router{

    private $route;
    private $view;
    private $controler;
    private $model;

    public function __construct(){
      require_once(LOCAL_ROOT . "php/routes.php");
      if(isset($_GET['route'])){
        $this->route = explode("/", $_GET['route']);
      }
      $route = isset($routes[$this->getRoute()]) ? $this->getRoute() : DEFAULT_ROUTE;
      $controller = "\\controllers\\" . $routes[$route]['controller'];
      $view = "\\views\\" . $routes[$route]['view'];
      $this->controller = new $controller(null);
      $this->view = new $view($this->controller, null);
    }

    private function getRoute(){
      return count($this->route) > 0 ? $this->route[0] : DEFAULT_ROUTE;
    }

    public function getView(){
      return $this->view;
    }

  }
?>
```

Nu moeten we er voor zorgen dat de app-klasse de nieuwe router gaat gebruiken.

- Pas de code in **/php/classes/mvc/app.class.php** aan.

A4: MVC IN PHP

```php
<?php
  namespace mvc;

  class App{
    private $router;

    public function __construct(){
      $this->router = new \mvc\Router();
    }

    public function __toString(){
      try {
        return $this->router->getView()->getHTML();
      } catch (Exception $e) {
        return $e.getMessage;
      }
    }

  }
?>
```

- Probeer de test route uit door in de browser adresbalk te typen:

`http://127.0.0.1/<jouw map>/mvc/test`

Als alles goed is gegaan krijg je in je browser nu de test-view te zien.

Eindopdracht

Breid de te nu toe gemaakt mvc-applicatie uit met de volgende onderdelen:

- Maak een Home-view en Home-Controller zodat de in router.php aangegeven standaard route ook werkt. Wat er op de view en in de controller komt te staan mag je zelf bepalen.
- Maak nog een View en Controller met inde view een html-formulier. Zorg er voor dat de controller de informatie uit het formulier op de volgende pagina laat zien. Deze pagina die de gegevens laat zien mag een ander route (view, controller, model) zijn als de route van het formulier.

beveiliging

Bij het bovenstaande voorbeeld toon je de invoer van een gebruiker direct op de website. Als je de invoer niet filtert geeft dit een beveiligingsrisico omdat de gebruiker naast normale tekst ook tekst kan invoeren die code bevat.
Typ bijvoorbeeld de code

```
<script>   alert("HACKED"); </script>
```

maar eens als invoer. Als het goed is krijg je dan in het volgende venster het javascript alert venster te zien.
In PHP zijn speciale functies om de invoer te filteren zodat het injecteren van javascript of bijvoorbeeld SQL niet meer mogelijk is.

- Maak zelf de objecten (interface, abstracte klassen) voor het model aan.
- Pas **router.class.php** aan zodat ook het nieuwe model gebruikt.
- Pas **router.class.php** aan zodat in het bestand routes.php niet voor elke route een view, controller en een model moeten worden gegeven. Als een element niet is opgegeven wordt dat element met **`null`** ingesteld, net als eerder met het model gebeurde.
- Maak voor de test- en de home-route de modellen aan. Deze mogen leeg zijn, omdat de view en controller geen gegevens uit het model gebruiken.
- Maak een nieuwe route (view, controller, model) waarbij de view gegevens uit de model laat zien. De gegevens hoeven niet uit een database te komen, maar dat mag wel.
- Maak een nieuwe route aan die alleen bestaat uit een controller. De controller moet de aanvraag redirecten naar één van de andere routes. Als je bijvoorbeeld de volgende twee routes hebt gemaakt

```
21      "red" => array(
22        "controller" => "RedController",
23      )
24
25      "blue" => array(
26        "controller" => "BlueController"
27      )
```

89

Dan zou de RedController er voor moeten zorgen dat de route blue wordt aangeroepen en daardoor de BlueController wordt uitgevoerd.

Redirect

Stuur in php een aanvraag door naar een andere pagina met de header functie.
De opdracht `header("location:index.php")` stuur de aanvraag door naar de **index.php**.

Bronnen

- ✓ **Video serie HTML op Youtube (UK)**
  https://www.youtube.com/watch?v=-USAeFpVf_A&list=PLr6-GrHUlVf_ZNmuQSXdS197Oyr1L9sPB&index=2

- ✓ **Video serie HTML op Youtube (NL)**

https://www.youtube.com/watch?annotation_id=annotation_4177865221&feature=iv&src_vid=3ejolTCFrYg&v=yQChsQQwJvc

- ✓ **Code Avengers**
  https://www.codeavengers.com (betaald vanaf $20, - / maand)
- ✓ **Thimble van Mozilla (NL), online editor met ondersteuning**
  https://thimble.mozilla.org/nl/
- ✓ **W3C – Referentie materiaal**
  https://www.w3.org/standards/webdesign/htmlcss

- ✓ **HTML Dog**
  http://www.htmldog.com/guides/html/beginner/

Bijlagen

Broncode

Bestand **.htaccess**

```
#rewriting request to the public folder

RewriteEngine On

RewriteCond %{THE_REQUEST} /public/([^\s?]*) [NC]
RewriteRule ^ %1 [L,NE,R=302]

RewriteRule ^((?!public/).*)$ public/$1 [QSA,L,NC]

RewriteCond ${REQUEST_URI} !\.(?:css|js|jpe?g|gif|png)$ [NC]
RewriteRule ^(public\/)([a-zA-Z0-9\-\ \/]*)$ index.php?route=$2 [QSA]
```

Bestand **/php/classes/mvc/app.class.php**

```
<?php
  namespace mvc;

  class App{
    private $router;

    public function __construct(){
      $this->router = new \mvc\Router();
    }

    public function __toString(){
      try {
        return $this->router->getView()->getHTML();
      } catch (Exception $e) {
        return $e.getMessage;
      }
    }

  }
 ?>
```

Bestand **/php/classes/mvc/interfaces/view.class.php**

```
<?php
  namespace mvc\interfaces;

  interface View {
    public function getHTML();
  }
 ?>
```

Bestand **/php/classes/mvc/view.class.php**

```
<?php
  namespace mvc;

  abstract class View implements \mvc\interfaces\View {

    protected $controller;
    protected $model;

    public function __construct($controller, $model){
      $this->controller = $controller;
      $this->model = $model;
    }

  }
 ?>
```

Bestand **/php/classes/mvc/interfaces/controller.class.php**

```
<?php
  namespace mvc\interfaces;

  interface Controller {
  }
 ?>
```

Bestand **/php/classes/mvc/controller.class.php**

```
<?php
  namespace mvc;

  abstract class Controller implements \mvc\interfaces\Controller {
    private $model;

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

    public function getPostData($name){
      return $this->getData($name);
    }

    public function getGetData($name){
        return  $this->getData($name, "GET");
    }

    public function getData($name, $type = "POST"){
      $dataSource = ($type == "POST"? $_POST : $_GET);
      return isset($dataSource[$name]) ? $dataSource[$name] : null;
    }

  }
 ?>
```

Bestand **/php/classes/mvc/router.class.php**

```
<?php
  namespace mvc;

  class Router{

    private $route;
    private $view;
    private $controler;
    private $model;

    public function __construct(){
      require_once(LOCAL_ROOT . "php/routes.php");
      if(isset($_GET['route'])){
        $this->route = explode("/", $_GET['route']);
      }
      $route = isset($routes[$this->getRoute()]) ? $this->getRoute() :
DEFAULT_ROUTE;
      $controller = "\\controllers\\" . $routes[$route]['controller'];
      $view = "\\views\\" . $routes[$route]['view'];
      $this->controller = new $controller(null);
      $this->view = new $view($this->controller, null);
    }

    private function getRoute(){
      return count($this->route) > 0 ? $this->route[0] : DEFAULT_ROUTE;
    }

    public function getView(){
      return $this->view;
    }

  }
 ?>
```