PHP · Level 4

UserRepository

Een klasse bouwen die praat met de database

Week	W13 – W14
Leerjaar	Jaar 1 · Periode 2
Doel	De Repository-laag begrijpen en zelf bouwen als onderdeel van het CSR-patroon
Werkwijze	Individueel · gebruik je eigen IDE en XAMPP
Inleveren	PHP-map gezipt via de gebruikelijke manier

Wat ga je bouwen?

Deze opdracht bestaat uit twee delen. In **Deel 1** bouw je een `UserRepository` die gebruikersdata beheert. In **Deel 2** pas je hetzelfde patroon toe voor een `ProductRepository` met volledige CRUD: aanmaken, opvragen, aanpassen en verwijderen.

Aan het einde kun je:

- een klasse instantiëren met een databaseverbinding via de constructor
- private properties gebruiken om implementatiedetails te verbergen
- prepared statements uitvoeren vanuit een klasse
- een ruwe database-rij omzetten naar een object (hydrate)
- een repository bouwen met alle CRUD-methodes

DEEL 1

UserRepository

Benodigdheden

Voer het volgende SQL-script uit in phpMyAdmin of je terminal:

```
CREATE DATABASE IF NOT EXISTS user_repo_opdracht
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci;

USE user_repo_opdracht;

CREATE TABLE IF NOT EXISTS users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL
);
```

Composer installeren

Composer is een tool voor PHP die autoloading voor je regelt. Zonder autoloading moet je bovenaan elk PHP-bestand alle klassen handmatig inladen met `require_once`. Met Composer stel je dat één keer in, en daarna vinden PHP en jouw klassen elkaar automatisch — geen losse require-statements meer.

Downloaden

Download de installer via getcomposer.org en voer hem uit. Na de installatie is `composer` beschikbaar als commando in je terminal. Controleer dat met:

```
composer --version
```

PSR-4 autoloading instellen

Maak een `composer.json` aan in de hoofdmap van je project:

```
{
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Dit vertelt Composer: alles in de `src/` map hoort bij de namespace `App\`. Schrijf je `new App\Domain\User` in je code? Dan laadt PHP automatisch `src/Domain/User.php`.

Autoloader genereren

Voer dit commando uit in de map waar je `composer.json` staat:

```
composer dump-autoload
```

Composer maakt een `vendor/` map aan met daarin `autoload.php`. Voeg dat bestand bovenaan je `index.php` toe:

```
require_once __DIR__ . '/../vendor/autoload.php';
```

> Voer `composer dump-autoload` opnieuw uit als je een nieuwe klasse aanmaakt, anders kan PHP die niet vinden.

Projectstructuur

Maak de volgende mappenstructuur aan:

```
user_repo_opdracht/
├── composer.json
├── public/
│   └── index.php
└── src/
    ├── Domain/
    │   └── User.php
    └── Repositories/
        └── UserRepository.php
```

De twee mappen hebben elk een eigen rol:

- `src/` bevat al je PHP-klassen. Deze map staat buiten de webroot en is niet rechtstreeks via de browser op te vragen.
- `public/` is de webroot. Alles wat via een URL bereikbaar moet zijn staat hier: je registratiepagina, loginpagina enzovoort.

> Stel dat iemand `jouwsite.nl/src/Domain/User.php` intypt. Doordat `src/` buiten de webroot staat, krijgt die persoon niks terug. Je backend-code is zo beter afgeschermd.

Stap 1: De User-klasse

Maak `src/Domain/User.php` aan.

De `User`-klasse is een pure datastructuur: hij bevat alleen data, geen SQL en geen logica. Dit soort klassen noem je een **domeinklasse**.

Zet bovenaan het bestand:

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

namespace App\Domain;
```

De klasse heeft drie readonly properties: `$id` (int), `$username` (string) en `$email` (string).

Gebruik **constructor property promotion**: schrijf de properties direct als parameters van de constructor, met hun access modifier erbij. Je hoeft ze dan niet apart te declareren.

```php
class User
{
    public function __construct(
        public readonly int $id,
        public readonly string $username,
        public readonly string $email,
    ) {}
}
```

> `readonly` zorgt ervoor dat de waarde na aanmaken niet meer gewijzigd kan worden. Dat is precies wat je wilt: een gebruiker die uit de database komt verandert niet.

Stap 2: De UserRepository-klasse aanmaken

Maak `src/Repositories/UserRepository.php` aan.

Zet bovenaan:

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

namespace App\Repositories;

use App\Domain\User;
use PDO;
```

Maak daarna een lege klasse aan:

```php
class UserRepository
{
}
```

Constructor

De `UserRepository` heeft een databaseverbinding nodig. Voeg een constructor toe die een `PDO`-object accepteert en opslaat in een **private** property. Gebruik constructor property promotion:

```
public function __construct(private readonly PDO $pdo) {}
```

Waarom `private`? De PDO-verbinding is een intern detail van de repository. Andere klassen hoeven niet te weten hoe de verbinding eruitziet of hoe die tot stand is gekomen.

Stap 3: findByUsername

Voeg een methode `findByUsername` toe die zoekt op gebruikersnaam.

```
public function findByUsername(string $username): ?array
```

De `?array` return type betekent: de methode geeft een `array` terug, of `null` als er niets gevonden is.

De methode doet het volgende:

1. Maak een prepared statement aan via `$this->pdo->prepare()`.
2. Voer het statement uit met `$username` als parameter.
3. Haal de rij op via `fetch()`.
4. Als er een rij is: geef een array terug met `'user'` (het User-object) en `'password'` (de wachtwoordhash).
5. Als er geen rij is: geef `null` terug.

Waarom geef je de wachtwoordhash apart mee? Het `User`-object zelf bevat nooit een wachtwoord. Maar de code die inloggen afhandelt heeft de hash nodig om te vergelijken met wat de gebruiker intypt.

Maak nu eerst het `User`-object direct aan in deze methode. In stap 5 verplaats je dat naar een aparte methode.

Stap 4: register

Voeg een methode `register` toe die een nieuwe gebruiker opslaat.

```
public function register(string $username, string $email, string $passwordHash):
User
```

De methode doet het volgende:

1. Maak een prepared statement aan voor een INSERT.
2. Voer het statement uit met username, email en wachtwoordhash.
3. Haal het nieuwe ID op via `$this->pdo->lastInsertId()`. Cast dit naar `int`.
4. Geef een `new User(...)` terug met het nieuwe ID, de username en het email.

> **Let op:** het hashen van het wachtwoord doe je **niet** in de repository. De repository slaat alleen data op. Hash het wachtwoord in `index.php` voordat je `register()` aanroept: `password_hash($password, PASSWORD_DEFAULT)`.

Stap 5: hydrate

Je maakt nu op meerdere plekken een `User`-object aan vanuit een database-rij. Als je later ook `findByEmail` of `findById` toevoegt, kopieer je steeds dezelfde code.

Maak een **private** methode `hydrate` die een database-rij omzet naar een `User`-object:

```
private function hydrate(array $row): User
```

De methode maakt een `new User(...)` aan met de waardes uit `$row`. Gebruik **named arguments** voor de leesbaarheid:

```
return new User(
    id: (int) $row['id'],
    username: $row['username'],
    email: $row['email'],
);
```

Vervang daarna in `findByUsername` het directe aanmaken van het `User`-object door `$this->hydrate($row)`.

> `hydrate` betekent letterlijk "water toevoegen". In code: je vult een leeg object op met data uit de database. Je blaast het als het ware tot leven.

Stap 6: Testen in index.php

Maak `public/index.php` aan en test je repository.

```
<?php
require_once __DIR__ . '/../vendor/autoload.php';

use App\Repositories\UserRepository;

$pdo = new PDO('mysql:host=localhost;dbname=user_repo_opdracht', 'root', '');
$pdo->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

$userRepository = new UserRepository($pdo);
```

Test de volgende dingen:

1. Roep `register()` aan met een username, email en een gehashed wachtwoord: `password_hash('test1234', PASSWORD_DEFAULT)`.

2 Dump het teruggegeven `User`-object met `var_dump()`.

3 Roep daarna `findByUsername()` aan met de username die je zojuist hebt opgeslagen.

4 Dump ook dit resultaat en controleer of alle velden kloppen.

Uitbreidingsopdrachten

1 Voeg een methode `findByEmail(string $email): ?array` toe. Gebruik daarin ook `hydrate()`.

2 Voeg een methode `findById(int $id): ?User` toe. Deze geeft direct een `?User` terug, want je hebt de wachtwoordhash hier niet nodig.

3 Gebruik `findById` in een klein scriptje dat een gebruiker ophaalt op basis van een ID dat je zelf invult.

DEEL 2

ProductRepository

De `UserRepository` staat. Nu ga je hetzelfde patroon toepassen voor een `ProductRepository` met volledige CRUD. De stappen zijn korter opgezet dan in Deel 1 — je kent de opbouw nu.

Database tabel aanmaken

Voeg de volgende tabel toe aan dezelfde database:

```
CREATE TABLE IF NOT EXISTS products (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(8,2) NOT NULL,
    stock INT UNSIGNED NOT NULL DEFAULT 0
);
```

Stap 1: De Product-klasse

Maak `src/Domain/Product.php` aan. De klasse heeft vier readonly properties:

- `$id` — int
- `$name` — string
- `$price` — float
- `$stock` — int

Gebruik dezelfde opbouw als de `User`-klasse: strict types, de juiste namespace en constructor property promotion.

Stap 2: De ProductRepository aanmaken

Maak `src/Repositories/ProductRepository.php` aan. Zelfde namespace-structuur en constructor als `UserRepository`. Begin met een lege klasse en voeg de constructor met `private readonly PDO $pdo` toe.

Stap 3: findById

```
public function findById(int $id): ?Product
```

SELECT op basis van ID. Gebruik `hydrate()` om de rij om te zetten. Geef `null` terug als er geen rij gevonden is.

Stap 4: findAll

```
public function findAll(): array
```

SELECT alle rijen uit de tabel. Gebruik `fetchAll()` in plaats van `fetch()` om alle rijen tegelijk op te halen. Loop daarna over de rijen en roep op elke rij `hydrate()` aan.

```
$rows = $stmt->fetchAll();
$products = [];
foreach ($rows as $row) {
    $products[] = $this->hydrate($row);
}
return $products;
```

Stap 5: insert

```
public function insert(Product $product): Product
```

INSERT statement. Gebruik `$product->name`, `$product->price` en `$product->stock` als waarden. Haal het nieuwe ID op via `lastInsertId()` en geef een nieuw `Product`-object terug met dat ID.

Stap 6: update

```
public function update(Product $product): void
```

UPDATE statement. Pas `name`, `price` en `stock` aan. Gebruik `$product->id` in de WHERE-clausule om het juiste product te vinden. Het return type is `void` — je hoeft niets terug te geven.

Stap 7: delete

```
public function delete(int $id): void
```

DELETE statement. Gebruik het meegegeven `$id` in de WHERE-clausule. Return type is `void`.

Stap 8: hydrate (private)

Voeg een private `hydrate(array $row): Product` methode toe, net als in `UserRepository`. Let op de juiste casts voor de numerieke velden:

- `$row['id']` en `$row['stock']` casten naar `int`
- `$row['price']` casten naar `float`

Stap 9: Alles testen

Test alle CRUD-methodes in `index.php`:

1. Voeg een paar producten toe via `insert()`.
2. Haal alle producten op met `findAll()` en dump de array.
3. Zoek één product op via `findById()`.
4. Pas een product aan via `update()` — verander de prijs of de voorraad.
5. Verwijder een product via `delete()`.
6. Roep `findAll()` nogmaals aan en controleer of de wijzigingen kloppen.