Webentwicklung

Backend: Entwicklung mit Komponenten

Inhalt dieser Einheit

1. Wiederverwendung mit Composer
2. Templates mit Twig
3. Objektorientierung in PHP
4. Daten speichern mit PHP
5. OR-Mapper Doctrine

Heutige Einheit

- Statische Website (Supero) zu dynamischer Seite
  - Schritt für Schritt, am Quellcode

```
$> git clone https://github.com/fzieris/php-demo-supero.git
$> cd php-demo-supero/
```

  - Git-Repo ist zum Nachvollziehen mit Tags versehen

```
$> git checkout -q <tag>
```

    - `-q` unterdrückt (hier irrelevante) Hinweismeldung
  - Lokal betrachten mit PHP-Development-Server:

```
$> php -S 127.0.0.1:8000 -t web
```

    - Server stoppen: `STRG`+`C`
- (technische Vorraussetzung: PHP 7 und Composer)

Dieser Foliensatz

- Grau hinterlegte Folien:
  - Fassen zusammen, was in den jeweiligen Schritten des Beispiels passiert
  - zum Nachlesen zu Hause
- Inhaltsfolien und *Zwischenstände*:
  - *Status: Plain PHP*
  - Composer
  - Twig
  - *Status: Templates*
  - OOP
  - Persistenz
  - Doctrine
  - *Status: ORM*

Start: Ausgangspunkt

```
$> git checkout -q start
```

- Statische Website, manuell erstellte HTML-Dateien:
  - `helden.htm`: Heldentabelle
  - 3× `held_*.htm`: Heldensteckbrief
- Lokal aufrufen:
  - http://127.0.0.1:8000/helden.htm

1. Umwandlung in PHP-Dateien

```
$> git checkout -q step-01
```

- HTML-Dateien werden PHP-Skripte
  - Umbenennung `.htm` → `.php`
  - Sonst keine Änderung, außer Anpassung der Link-Ziele
- PHP-Interpreter wird nun aufgerufen
  - hat aber nichts zu tun
- Lokal aufrufen (neue Adresse):
  - http://127.0.0.1:8000/helden.php

2. Arrays und Schleifen

```
$> git checkout -q step-02
```

- Navigation auslagern und per `include` einbinden
  - Unterstrich im Dateinamen `_navi.php` ist nur Konvention
- Heldennamen in Array speichern
  - `foreach`-Schleife gibt für jeden Namen eine Tabellenzeile aus
    - (außer Batman, Flash und Robin: tote Links)

3. Heldensteckbrief als Skript

```
$> git checkout -q step-03
```

- Zusammenfassen der Steckbriefe in `held.php`
  - Speichern aller Helden-Infos in assoziativem Array
  - Auswahl des anzuzeigenden Helden per GET-Variable `name`
  - Bestimmung der CSS-Klasse anhand des Helden-Status
  - Kurze Syntax für Ausgaben:

```
<b><?=$var?></b>           <!-- die Ausgabe ist in beiden -->
<b><?php echo $var ?></b>  <!-- Fällen identisch          -->
```

Zwischenstand

- Bisher:
  - HTML-Code durch Schleifen und Fallunterscheidung erzeugen
  - Assoziative Arrays als flexible Datenstruktur
- **Bewertung**
  - neue Menüeinträge und Helden(-steckbriefe): recht einfach 😀
  - Design- und Layout-Änderungen: noch eher aufwändig 😕
- Als nächstes: Layout-Problem lösen
  - Vorbereitung: Wiederverwendung von Code

Composer

Wie man das Rad nicht ständig neu erfindet

Wiederverwendung von PHP-Code

- PHP-Standard-Bibliothek ist umfangreich
  - aber relativ “low-level” (siehe vorige Einheit)
- Ganz früher:
  - Suche im Web nach PHP-Funktionen oder -Bibliotheken
  - einfache PHP-Dateien herunterladen, per `include` einbinden
- ab 1999: PEAR
  - herunterladen von gezippten Datei-Sammlung mit `package.xml`
- ab 2012: Composer
  - Paketverwaltung, Komponenten haben selbst Abhängigkeiten
  - Fremde Komponenten & eigene Anwendung mit `composer.json`

Composer-Dateien

- `composer.json`: Bsp. mit einer Abhängigkeit

```
{
    "require": {
        "monolog/monolog": "1.0.*"
    }
}
```

- `composer.lock`: ausführliche, explizite Angaben

```
{
    "packages": [
        { "name": "monolog/monolog", "version": "1.0.2", ... }
    ], ...
}
```

- `vendor/`: Ziel-Ordner für heruntergeladene Komponenten
  - (*Nicht* unter Versionskontrolle; `composer.json`/`.lock` reichen)

Quelle: https://getcomposer.org/doc/04-schema.md

Composer-Befehle

- `composer install`
  - Wertet `composer.lock` aus, verwendet exakte Versionen
  - Aktualisiert Inhalte in `vendor/` (lädt, löscht, aktualisiert)
    - *(das macht Heroku automatisch beim Deployment!)*
- `composer update`
  - Aktualisiert `composer.lock`
    - Schaut, ob z.B. Version 1.0.5 erschienen ist
- `composer require`
  - Fügt neue Abhängigkeit in `composer.json` hinzu
  - Syntax: *Anbieter* `/` *Paketname* [`:` *Versionsbereich*]
  - `composer require monolog/monolog`
  - `composer require monolog/monolog:1.0.*`

Quelle: https://getcomposer.org/doc/03-cli.md

Versionen auswählen

Notation	1.2.0	1.2.1	1.2.2	1.2.3	1.3	2.0	2.1
`1.2.3`				X			
`>=1.2.2`			X	X	X	X	X
`>=1.2.2 <2.0`			X	X	X		
`1.2.*`	X	X	X	X			
`~1.2`	X	X	X	X	X		
`~1.2.1`		X	X	X			
`^1.2`	X	X	X	X	X		
`^1.2.1`		X	X	X	X		

Quelle: https://getcomposer.org/doc/articles/versions.md

4. Website als Composer-Projekt

```
$> git checkout -q step-04
$> composer install
```

- Neue Dateien:
  - `composer.json`: Interaktiv erstellt durch `composer init`
  - `composer.lock`: Generiert durch `composer install`
- aktuell prüft `composer install` nur die PHP-Version:

```
Your requirements could not be resolved to an installable
set of packages.

Problem 1
  - This package requires php >=7.0.0 but your PHP version
    (5.6.32) does not satisfy that requirement.
```

Twig

Templates zur Ausgabeerzeugung

Templates in PHP

- **Idee**: Trennung von Geschäftslogik und Darstellung
  1. PHP-Skript führt Logik aus (Eingabe-Verarbeitung, Kommunikation mit Datenbank, …)
  2. PHP-Skript ruft **Template-Engine** auf und übergibt Parameter
  3. Template selbst ist passiv, und zeigt Parameter an
     - etwas Anzeigelogik: Schleifen, Bedingungen, Filter
- **Template-Engines**:
  - Früher Vertreter: Smarty (ab 2002)
    - Syntax noch nah an PHP
  - Moderner: Twig (ab 2009)
    - Klarere Syntax, Integration in Symfony (nächste Einheit)

Twig-Beispiel

```php
// PHP-Skript (gekürzt)
$mails = [ 'Guten Morgen', 'Re: Handyvertrag' ];
echo $twig->render('index.twig',
    ['name' => 'Mia', 'mails' => $mails]
);
```

```
{# Twig-Template 'index.twig' #}
<h1>Hallo {{ name }}</h1>
<p>Mails:</p>
<ol>
    {% for subject in mails %}<li>{{ subject }}</li>{% endfor %}
</ol>
```

```
<h1>Hallo Mia</h1>
<p>Mails:</p>
<ol>
<li>Guten Morgen</li><li>Re: Handyvertrag</li>
</ol>
```

5. Twig installieren

```
$> git checkout -q step-05
$> composer install
```

- Geänderte Dateien:
  - `composer.json`: Ergänzt durch `composer require twig/twig:^2.0`
- Twig ist nun verfügbar, wird aber noch nicht genutzt

6. Twig verwenden

```
$> git checkout -q step-06
```

- Neue Dateien:
  - `bootstrap.php`:
    - Teilt Twig mit, dass die Templates in Dateien vorliegen (`/templates`)
    - Macht Twig als `$twig` verfügbar, für alle, die `bootstrap.php` nutzen
  - `templates/*.twig`: Template-Dateien
    - Dateiendung egal, Inhaltlich: Twig-Syntax
- Geänderte Dateien:
  - `helden.php` & `held.php`: erzeugen nicht mehr selbst HTML-Code
    - sondern nutzen `$twig` dafür

7. Layout auslagern

```
$> git checkout -q step-07
```

- Neue Datei: `templates/base.twig.htm`:
  - Definiert Struktur mit Platzhaltern (`block`)
- Geänderte Dateien:
  - `helden.htm.twig` & `held.htm.twig`:
    - nutzen Basis-Layout (`extends`)
    - füllen Platzhalter (`block`)

Zwischenstand

- Template-Engine per Composer in Projekt installiert
  - Einbindung erfolgt in gemeinsamer `bootstrap.php`
- PHP-Skripte beinhalten keinen HTML-Code mehr
- Twig-Templates …
  - zeigen Variablen an: `{{ name }}`
    - auch gefiltert: `{{ name|lower }}`
  - können Kontrollstrukturen nutzen: `{% for hero in heros %}`
  - unterstützen Vererbung: `{% block title %}`
- … und können um weitere Features ergänzt werden
- **Bewertung**
  - Logik und Darstellung getrennt, Layout zentral änderbar 😁
  - Heldeninfos in verschiedenen PHP-Dateien 😕

Objekt-Orientierung in PHP

Objekt-Orientierung in PHP

- Bisher: PHP nur prozedural
  - d.h. *Funktionen* definieren und aufrufen
  - Datenhaltung meist in (assoziativen) Arrays
- Strukturierter: Objekt-Orientierung
  - d.h. Bündlung von Daten und zugehöriger Logik
  - in *Klassen* mit *Feldern* und *Methoden*

Klassen definieren in PHP

```php
// Auto.php
namespace HTW\Webdev\Demo; // wie 'package' in Java

class Auto {
  const RAEDER = 4;
  private $color;
  public $speed = 0;
  public static $totalCars = 0;

  public function __construct($color) {
    $this->color = $color;
    self::$totalCars++;
  }

  public function speedUp() {
    $this->speed += 5;
  }
}
```

Klassen nutzen in PHP

```php
// auto-test.php
include 'Auto.php';

use HTW\Webdev\Demo\Auto; // wie 'import' in Java

$autos = [ new Auto('rot'), new Auto('grün') ];
echo Auto::$totalCars; // 2

$autos[0]->speedUp();
echo $autos[0]->speed; // 5
```

Übersicht: PHPs OOP-Syntax

- Klassen: [`abstract`|`final`] `class`
- Interfaces: `interface`
- Methoden:
  - [`final`|`abstract`] **`public`**|`protected`|`private` [`static`] `function`
- Felder: `public`|`protected`|`private` [`static`]
- Zugriffe
  - `$this` & `->` für Instanzen/Objekte

```php
$this->speedUp();    $auto->speedUp();
```

  - `self` & `::` für statische Zugriffe/Klassen

```php
self::$totalCars++;  Auto::$totalCars++;
```

  - Elternklasse: `parent`

```php
parent::__construct(); // Java: super();
```

Quelle: http://php.net/manual/de/language.oop5.basic.php

Magische Methoden

- Aufruf intern unter bestimmten Bedingungen
  - `__construct()`: Konstruktur (bei `new`-Aufruf)
  - `__get($name)`: Zugriff auf fehlendes Feld

```php
echo $auto->verbrauch; // $auto->__get('verbrauch')
```

  - `__set($name, $value)`: Zugriff auf fehlendes Feld

```php
$auto->halter = 'Klara'; // $auto->__set('halter', 'Klara')
```

  - `__call($name, $arguments)`: Aufruf fehlender Methode

```php
$auto->stop(true, 20); // $auto->__call('stop', array(true, 20))
```

Quelle: http://php.net/manual/de/language.oop5.magic.php

Autoloading von Klassen

- PHP-Code wird interpretiert
  - Einstiegdatei, plus alles was per `include` geladen wird
    - d.h. Anweisungen, Funktionen und Klassen-Definitionen könnten vermischt in einer Datei stehen
- Gute Praxis (siehe PSR-1):
  - Klassen-Definitionen separat, eine Klasse pro Datei
- **Problem**: vielen Klassen → `include` unübersichtlich
- **Lösung**: *Autoloading*
  - Interpreter stößt auf unbekannte Klasse: Autoloader springt an
  - Autoloader "rät" Pfad zur Klassen-Datei
  - Konventionen erleichtern das Raten (PSR-0 oder PSR-4)
- Composer bringt Autoloader mit (→ `bootstrap.php`)

8. Objektorientierung

```
$> git checkout -q step-08
$> composer install
```

- Neue Datei: `src/Hero.php`
  - Klasse `Hero` bündelt alle Eigenschaften und Daten der Helden
- Geänderte Dateien:
  - `helden.php` und `held.php`: Rufen `Hero`-Methoden auf
    - brauchen `Hero.php` aber nicht selbst includen, denn …
  - `composer.json`: teilt Composer mit

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

    - *Klassen des Namespace ‘Supero’ sind PSR-4-konform im Ordner ‘src’*
    - *Bitte bei Bedarf automatisch laden*
- Lokal aufrufen (jetzt mit allen Steckbriefen 👍):
  - http://127.0.0.1:8000/helden.php

Persistenz

Trennung von Programm und Daten

Nächste Schritte

- Für Webanwendung: Nutzerinteraktion mit Wirkungen
  - Heldeninfos in `Hero`-Klasse sind statisch
  - Also: Programm und Daten trennen
- Möglichkeiten zur Datenauslagerung:
  1. Dateien im Dateisystem, eigenes Format
     - `file()` bzw. `file_get_contents()`, `file_put_contents()`
     - `strpos()`, `substr()`, `explode()`
  2. Dateien im Dateisystem, Standardformate
     - **Comma-separated values**: `fgetcsv()`, `fputcsv()`
     - **JSON**: `json_decode()`, `json_encode()`
     - **YAML**: `yaml_parse_file()`, `yaml_emit_file()`
  3. Datenbank

PHP und Datenbanken

- Früher: `mysql_*`-Funktionen, viele Sicherheits-Probleme
  - seit PHP 4.3 deprecated, seit PHP 7.0 entfernt
- Heute: Auswahl verschiedener Datenbanktreiber, z.B.
  - MySQL: `mysqli`-Funktionen
  - PostgreSQL: `pg`-Funktionen
  - SQLite: `sqlite`-Funktionen
- Ab PHP 5.1: Abstraktion PDO, *PHP Data Objects* (≈ JDBC)

```php
$db = new PDO("mysql:dbname=testdb;host=$host", $user, $password);
$db = new PDO("pgsql:dbname=testdb;host=$host", $user, $password);
$db = new PDO("sqlite:testdb.db");
$stmt = $db->query("SELECT * FROM products WHERE price > 10");
echo $stmt->rowCount() . " Treffer";
```

Doctrine

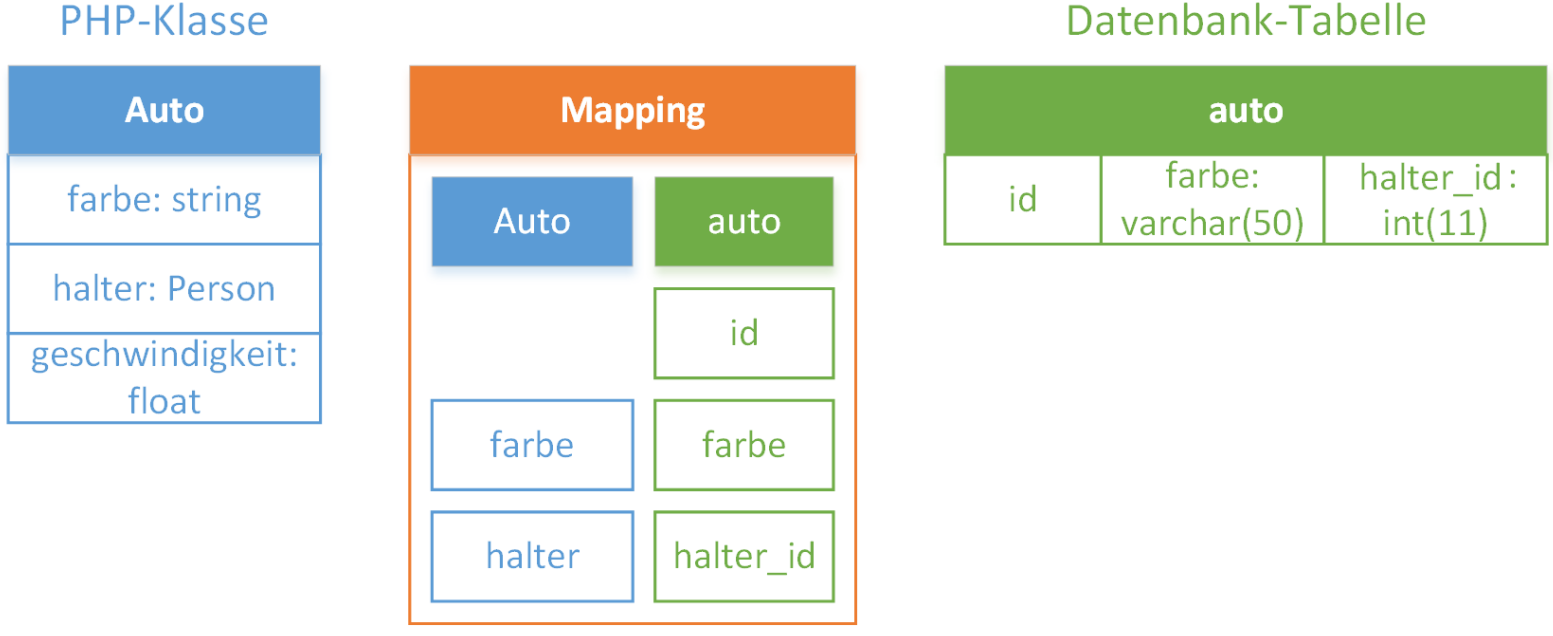
Datenbankanbindung und Objekt-Relationale Mapper

Objekt-Relationale Mapper

- In objekt-orientierten Programmentwürfen:
  - Klasse/Attribut/Objekt → Tabelle/Spalte/Tabellenzeile
  - Assoziation → Fremdschlüssel
    - (für `n:m`-Assoziationen: eigene Tabelle)
- Häufige Aufgaben
  - Neue Klasse/neues Attribut: Datenbankschema anpassen
  - Immer wieder ähnliche SQL-Queries zum Anlegen, Bearbeiten, Auflisten und Löschen von Objekten schreiben
- Objekt-relationale Mapper übernehmen das
  - und verstecken die Datenbank
- Beispiele:
  - Hibernate (Java), ActiveRecord (Ruby), in Django (Python)
  - PHP: **Doctrine ORM**

Einfacher ORM-Anwendungsfall

1. Existierende PHP-Klasse mit Feldern
2. Ergänze Mapping-Information:
   - Welche Felder sollen in die Datenbank?
     - z.B. durch Annotationen in PHP-Klasse, oder separate XML-Datei
3. Doctrine erzeugt Datenbankschema



Doctrine-Konzepte

- In Datenbank: “Objekte” (keine Zeilen)
- **EntityManager**: Zugriff auf alle Objekte in der Datenbank
  - Zugriff: Klassenname und Eigenschaften
    - z.B. `Supero\Hero` und ID oder `name = 'Batman'`
  - Aufruf von Objekt-Methoden
    - z.B. `$hero->setStatus('gebucht');`
  - pro Objekt: Änderung vormerken
    - z.B. `$entityManager->persist($hero);`
  - Am Ende: Transaktion abschließen
    - `$entityManager->flush();`
- Doctrine optimiert die SQL-Queries:
  - z.B. Helden-Status 3× geändert: speichert nur letzten Stand

9. Doctrine installieren

```
$> git checkout -q step-09
$> composer install
```

- Geänderte Datei: `composer.json`
  - Doctrine hinzugefügt mit `composer require doctrine/orm:^2.5`
  - Doctrine wird aber noch nicht benutzt

10. `Hero` ins Datenbankschema

```
$> git checkout -q step-10
$> vendor/bin/doctrine orm:schema-tool:create
```

- Geänderte Dateien:
  - `bootstrap.php`: teilt Doctrine mit …
    - wo die Mapping-Informationen stehen
    - wo die Datenbank liegt (hier: SQLite-Datenbank)
  - `src/Hero.php`:
    - Annotationen zur Speicherung in der Datenbank
- Neue Datei `cli-config.php`:
  - Damit die Doctrine-Kommandozeile `bootstrap.php` kennt
- Datenbank beinhaltet noch keine Daten

11. `Hero`-Objekte in Datenbank

```
$> git checkout -q step-11
$> php tools/reset_database.php
```

- Neue Datei `tools/reset_database.php`
  - Helfer für Demo-Zwecke, befüllt Datenbank mit Helden-Daten
- Geänderte Dateien:
  - `helden.php` und `held.php` nutzen Doctrines `EntityManager`
  - `.twig`-Templates nutzen `Hero`-Objekte
  - `Hero`-Klasse braucht die statischen Methoden nicht mehr
- Lokal aufrufen (Daten kommen aus der Datenbank):
  - http://127.0.0.1:8000/helden.php

Ausblick: Mögliche Erweiterung

```php
// held-buchen.php
$name = filter_input(INPUT_GET, 'name');
$hero = $entityManager->getRepository('Supero\Hero')
          ->findOneBy(['name' => $name]);

if ($hero->getStatus() != 'ausgebucht') {
  $hero->setStatus('ausgebucht');
  $msg = "Erfolgreich gebucht";
} else {
  $msg = "Leider schon ausgebucht";
}

$entityManager->persist($hero);
$entityManager->flush();

echo $twig->render('held-buchen.twig.htm',
                   ['hero' => $hero, 'status' => $msg]
);
```

Zusammenfassung Endzustand

- **Bewertung**:
  - Durch Templates sind Darstellung und Geschäftslogik sauber getrennt
  - Durch ORM können die Helden-Infos leicht durch Skripte geändert und in der Datenbank dauerhaft gespeichert werden
  - Aber: Es fehlen noch viele “Kleinigkeiten”
    - Architektur, um Funktionen zu ergänzen
    - Nutzer-/Sessionverwaltung
    - Rechteverwaltung (Helden buchen vs. Helden neu eintragen)
    - Fehlerbehandlung
    - lesbare URLs
    - Sicherheit, …
- Für das alles gibt es jeweils Komponenten
  - Zusammen: *Web-Framework*, nächste Einheit

Zusammenfassung

- **Composer** zur Abhängigkeitsverwaltung
- **Twig** als Template-Engine
  - Trennung von Geschäftslogik und Darstellung
  - einfache und erweiterbare Template-Sprache
- **OOP**-Syntax von PHP
- Möglichkeiten der **Datenspeicherung** und -verwaltung
- **Doctrine** als OR-Mapper
  - Objekt-Orientierte Geschäftslogik
  - Datenbank-Tabellen und -Queries nicht selbst verwalten müssen

Danke!