Persistenz

Warum ist Persistenz wichtiges Thema?

- Moderne Anwendungen sind kompliziert
- Beinhalten eine Vielzahl von dynamischen Inhalten
  - In unterschiedlichen Fachlichkeiten
- Eigentlich immer in Form von Datenbanken
  - Optimiert für einfaches Speichern kleiner Informationseinheiten
    - Speichern die reine Information
  - Je nach Anwendungszweck spezialisiert
    - Auch oft mehrere verschiedene
- Performancekritische Schicht
  - Professionelle Lösungen machen Sinn

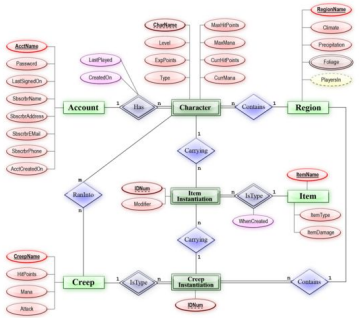
SQL

- Oft interpretiert als **S**tructured **Q**uery **L**anguage (1970er Jahre als SEQUEL)
- Datenbanksprache
  - Definition von Datenstrukturen
  - Abfrage und Manipulation von Daten
- ISO Standard
- Gedacht für relationale Datenbanken
- Basis unterschiedlicher Dialekte und Erweiterungen (z.B. MySQL, T-SQL, ...)

```
SELECT title, author FROM BlogPost WHERE title LIKE 'Web%'
```

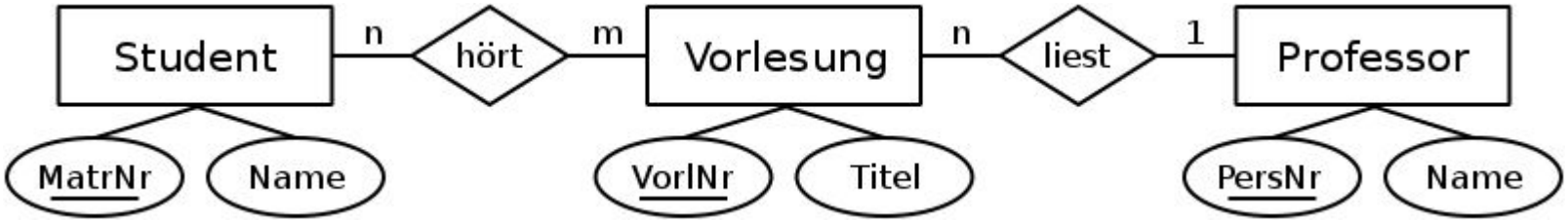
Schema

- SQL basierte Datenbanken nutzen sog. Schemata
  - Festgeschrieben Struktur für Daten
  - Und vieles mehr z.B. Stored Procedures
- Begriff oft im Zusammenhang mit relationalen Datenbanken
  - Aber auch Nicht-Relationale Schemata
- Erlauben das Definieren von Bedingungen an Daten (sog. Constraints)
- Lassen sich je nach Anforderungen vom Start weg designen
- Beschreiben Relationen zwischen Datentypen
  - Repräsentierbar durch sog. ER-Diagramme
- **Ein Schema ist ein Artefakt das es zu versionieren gilt!**



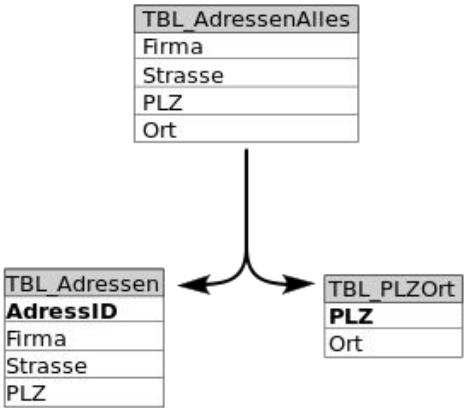
Schema - Relationen

- Relationale Datenbanken leben von Relationen (dah!)
- Drei Arten von Relation (unterschiedliche Richtungen beachten)
  - 1-zu-1, 1-zu-viele, viele-zu-viele
- Verknüpfte Datensätze referenzieren sich anhand von Identifiern
  - Sog. Foreign Keys als Referenz auf den Primary Key eines Datensatzes einer anderen Tabelle
- Foreign Keys sitzen auf der “Viele” Seite (z.B. PersNr des Professors in der Vorlesung)
  - Vergleiche 1. Normalform



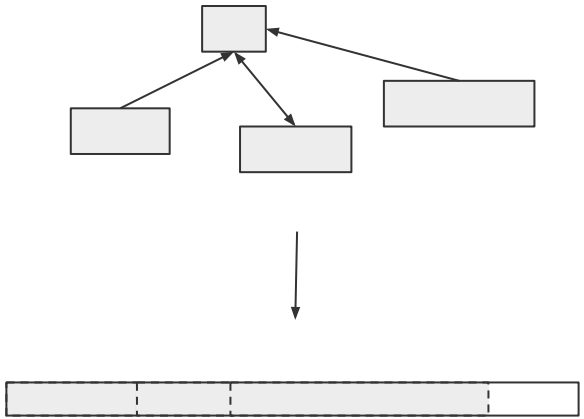
Schema - Normalisierung

- Normalisierung beschreibt die Verteilung von Daten auf mehrere Tabellen um Redundanz zu minimieren
- Keine Redundanz == Maximale Konsistenz
- Leichter programmatisch zu verarbeiten
- Grad der Normalisierung wird beschrieben durch sog. **Normalform**
  - Aufeinander aufbauend
  - Sechs Stück
- Beispiel
  - 1NF: Atomare Werte und Freiheit von Wiederholungsgruppen (Feld “Titelliste” eines Albums)
  - 5NF: Keine Mehrfachabhängigkeit mit Verstrickung (“Lieferant” liefert “Produkt” an “Projekt” in einer Tabelle)
- Nachteil: Komplexität



Schema - Denormalisierung

- Bewusste Rücknahme von Normalisierung
  - Z.B. Zusammenführen von Tabellen in eine Einzelne
- Reduziert Komplexität
- Erhöht Performance von Abfragen und Manipulation
- Wichtig: **Rücknahme**
  - Denormalisierung sollte bewusste Maßnahme mit klarem Ziel sein
- Nachteil: Redundanz, Inkonsistenz
- Mischformen möglich mit Normalisierten Tabellen



Schema - Migration

- **Veränderung eines Schemas erfordert klaren Weg für ein Update**
  - Daten sind zentraler Inhalt einer Anwendung und besonders schützenswert
- Migration zwischen Schemaversionen kann komplex sein
  - Auf welchem System besteht welche Version?
  - Müssen bestehende Daten angepasst werden?
  - Sind Backups dann noch nutzbar?
- Sollte mit entsprechendem Tooling vorgenommen werden (z.B. Phinx, Doctrine, etc.)
  - Frameworks generieren SQL Befehle zum Schemaumbau
  - Erlauben Vor- und Zurückspringen in Versionen
- Rollback sollte möglich sein

NoSQL

- **N**ot **o**nly **SQL** (2009 als Begriff, 2000er Jahre als Technik)
- Meist nicht-relationaler Aufbau
- Grundidee: Optimierung für viele und umfangreiche Lese-/Schreibzugriffe
- Große Untergliederung nach Anforderung
  - Graphdatenbanken, Dokument-Datenbanken, BigData Optimierungen, …
- Häufigste Anwendung im Web:
  - Dokument-Datenbanken zur schemalosen Speicherung beliebiger Daten (MongoDB, CouchDB)
  - Key-Values Stores zur Verteilung globaler Daten in Skalierungsmodellen (Redis, memcached)
  - Unterbau für performante Suchfunktion (Solr, Elasticsearch)
- Hauptziel: Performance
- Nachteil: Weniger Sicherheit durch mangelnde Struktur

SQL und NoSQL Zusammenspiel

- Häufiges Muster: Datenhaltung normalisiert -> Redundanz in Denormalisierung
  - Erlaubt Nutzung der Vorteile beider Systeme
  - Z.B. schnelle Suche über Elasticsearch
- Z.B. durch regelmäßige Prozesse über Cron
  - Erzeugen denormalisierten Inhalt für NoSQL Datenbestand
- Achtung: komplex in der Synchronisation!
  - Inkonsistenzen haben negative Effekte
  - *Single version/source of truth* Probleme
  - Kosten/Nutzen genau abwägen!

Datenbankzugänge

- Zugriff über PDO (**P**HP **D**ata **O**bject)
  - Alternativen möglich (z.B. MySQLi)
  - Abstraktion von Datenbankzugriffen
  - Erlaubt Nutzung unterschiedlicher (relationaler) Datenbanken über “Treiber” System
- PDO ist voll OOP
- Nutzt sog. DSN (**D**ata **S**ource **N**ame)

```php
try {
    $connection = new PDO( dsn: "mysql:host=localhost;dbname=webentwicklung", $username, $password);
    // set the PDO error mode to exception
    $connection->setAttribute( attribute: PDO::ATTR_ERRMODE,  value: PDO::ERRMODE_EXCEPTION);
    echo "Connected successfully";
} catch(PDOException $e) {
    echo "Connection failed: " . $e->getMessage();
}

$query = $connection->prepare( statement: "show tables");
$query->execute();
```

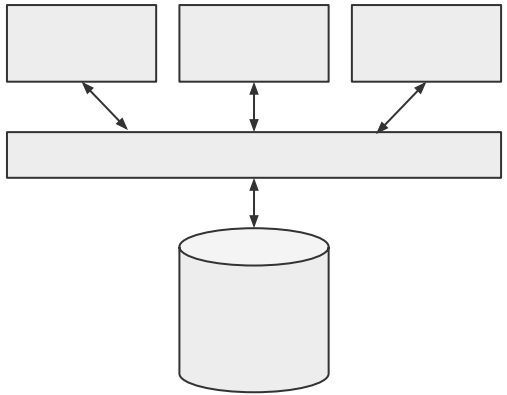
Datenbankzugänge

- Datenbanken haben eigene Rechteverwaltung
- Logindaten/etc. notwendig
  - Für verschiedene Systeme unterschiedlich
- Nutzung von Systemabhängigen Speicherorten
  - Systemvariablen des Betriebssystems
  - (Lokale) Konfigurationsdateien
  - PDO bieten bereits gute Mechanismen dazu
- Empfohlen: *.env* Dateien z.B. geladen über “vlucas/phpdotenv”
  - Lassen sich einfach verwalten und über Filesystem und Webserver schützen
  - In die *.gitignore* eintragen!
  - In PHP Zugriff über verschiedene Methoden

```
HOST="localhost"
DB_NAME="webentwicklung"
USERNAME="we_user"
PASSWORD="IG§RUBFFu8h9gefwfvzbiSLOt&§HIS"
DSN="mysql:host=${HOST};dbname=${DB_NAME}"
```

Zugriffsschicht

- Boilerplate (sich wiederholender Grund-Code) stört und ist fehleranfällig
  - Abstraktion schafft Abhilfe
- Sprachliche Trennung (SQL und PHP) auch oft sinnvoll
- Aufbau einer kapselnden Zugriffsschicht um Datenbankzugriffe und -abfragen
  - Erlaubt leichten Austausch bei Technologiewechsel
  - Intuitiver im Zugriff durch OOP Schnittstellen
- Können unterschiedlich gestaltet werden
  - Oft Nutzung von Design Patterns



Zugriffsschicht - Patterns: Repository

- Persistenz Abstraktion mit Schnittstellen ähnlich einer einfachen Liste
- Bietet einfache Methoden
  - add/save
  - remove/delete
  - get
  - ...
- Liefert als Ergebnis Liste von Objekten
- Oft spezifisch für einzelne Models implementiert, z.B. *BlogPostRepository*
  - Erlaubt feinen Zugriff z.B. durch Weglassen der *add()* und *update()* Methoden

```php
interface BlogPostRepositoryInterface
{
    public function save(BlogPost $post): void;
    public function get($id): BlogPost;
    public function getList(BuilderInterface $collectionBuilder): array;
    public function delete(BlogPost $post): void;
}
```

Zugriffsschicht - Patterns: Builder

- Zerteilt komplexe Vorgänge (Z.B. Aufbau eines SQL Queries) in kleine Schritte
- Erlaubt granulares Anpassen an spezielle Situation
- Oft als direkte Abstraktion um Datenbankabfragen
  - Weniger Tech-Vermischung
- Erleichtert programmatischen Aufbau von Queries
- Erhöht Lesbarkeit (bei komplexen Queries)

```
$query = new Query( select: '*', from: 'User', alias: 'u', where: 'u.id = ?1', sort: 'u.name', order: 'ASC', param: 1, value: 100);
// oder
$id = 100;
$query = $connection->prepare( statement: "select * from User u where u.id = :id order by u.name ASC");
$query->bindParam( parameter: ':id', &variable: $id);
```

↓

```
$queryBuilder = new QueryBuilder();
$queryBuilder->select( select: '*')
    ->from( from: 'User', alias: 'u')
    ->where( where: 'u.id = ?1')
    ->orderBy( sort: 'u.name', order: 'ASC')
    ->setParameter( param: 1, value: 100);
```

Prepared Statement

- Pre-Compiling von Datenbankabfragen vor der Ausführung
  - Macht Abfrage effizient mehrfach nutzbar
- Kann parametrisiert werden
- Verhindert dynamische Manipulation von Abfragen durch Dritte (“Hacker”)
  - Sog. SQL Injection
  - Durch Trennung von dynamischen Inhalten (Parametern) und Abfrage

```
$title = 'test';
$query = $connection->prepare( statement: "select * from blog_posts where title = :title");
$query->bindParam( parameter: ':title',  &variable: $title);
$query->execute();
```

Integration Model

- Problem: Daten aus Datenbank müssen PHP Objekte werden
  - Einfacher für Objekt-Datenbanken
- Ein Datensatz (eine Zeile) === eine Instanz
- Tabellen werden zu Arrays oder Collection-Objekten zusammengefasst
- Komplexe Aufgabe
  - Spezielle ORM Frameworks/Bibliotheken (Doctrine, Propel, …)
- Kleiner gedacht: Deserialisierung (Z.B. JMSSerializer)
- Schlicht auch mit Bordmitteln (PDO) erreichbar

Integration Model - ORM Framework

- **O**bject-**r**elational **M**apping
- PHP Bibliothek zur einfachen Handhabung von Persistenz
- Bieten fertige Zugriffsschicht zur Persistenz
  - Inklusive Umwandlung in PHP Objekte
- Erlauben komplexes Mapping mit Schema
  - Komfortables Auflösen von Relationen
- Können Schema aus Code generieren
  - Oft weiteres Tooling
- Bieten ausgefeiltes Caching

```php
/**
 * Class Quiz
 *
 * @ORM\Entity
 */
class Quiz
{
    /**
     * The unique domain object ID.
     *
     * @var string
     *
     * @ORM\Id
     * @ORM\Column(type="string")
     * @ORM\GeneratedValue(strategy="UUID")
     */
    protected $id;

    /**
     * @var Question[]
     *
     * @ORM\ManyToMany(
     *     targetEntity="Question",
     *     inversedBy="quizzes",
     *     cascade={"persist"})
     */
    protected $questions;
}
```

Integration Model - PDO FetchMode

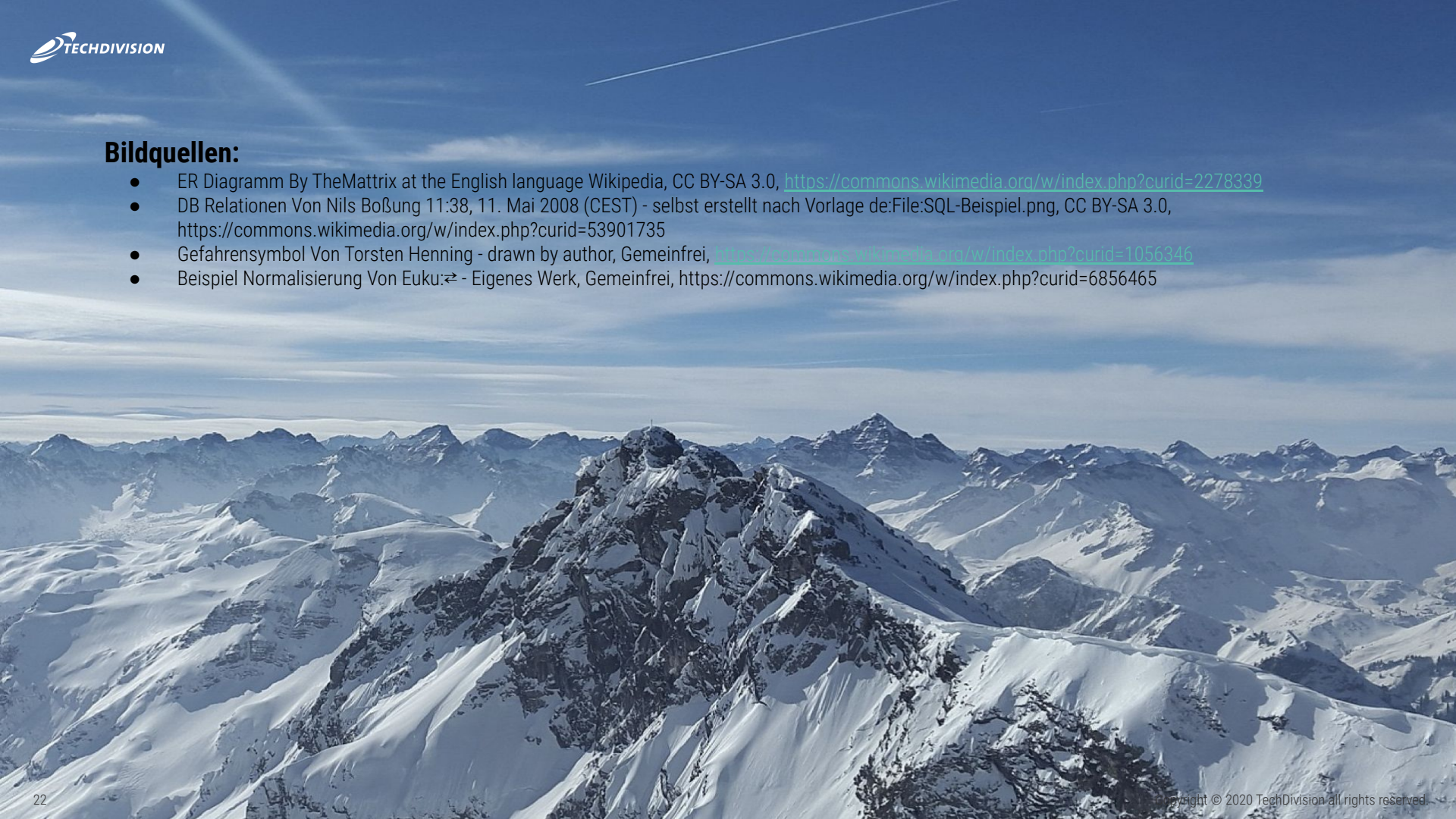
- PDO bietet rudimentäre ORM Fähigkeiten
  - Über Property Namen === Spaltennamen
  - Erlaubt zusätzliche Konstruktorparameter
- Erlaubt Ausgabe als assoziatives Array zum manuellen Mappen
- Rudimentär aber ohne Framework nutzbar

```php
class BlogPost
{
    public string $id;
    public string $title;
    public string $text;
    public string $author;
}
```

```php
$query = $connection->prepare( statement: "select * from blog_posts");
$query->execute();
$query->setFetchMode( mode: PDO::FETCH_CLASS, classNameObject: BlogPost::class);
$result = $query->fetchAll();
```

Quellen:

- https://de.wikipedia.org/wiki/SQL
- https://www.php.net/manual/en/book.pdo.php
- https://en.wikipedia.org/wiki/Object-relational_mapping
- https://de.wikipedia.org/wiki/Normalisierung_(Datenbank)
- https://de.wikipedia.org/wiki/MySQL
- https://www.doctrine-project.org/



Bildquellen:

- ER Diagramm By TheMattrix at the English language Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2278339
- DB Relationen Von Nils Boßung 11:38, 11. Mai 2008 (CEST) - selbst erstellt nach Vorlage de:File:SQL-Beispiel.png, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=53901735
- Gefahrensymbol Von Torsten Henning - drawn by author, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=1056346
- Beispiel Normalisierung Von Euku:⇄ - Eigenes Werk, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=6856465