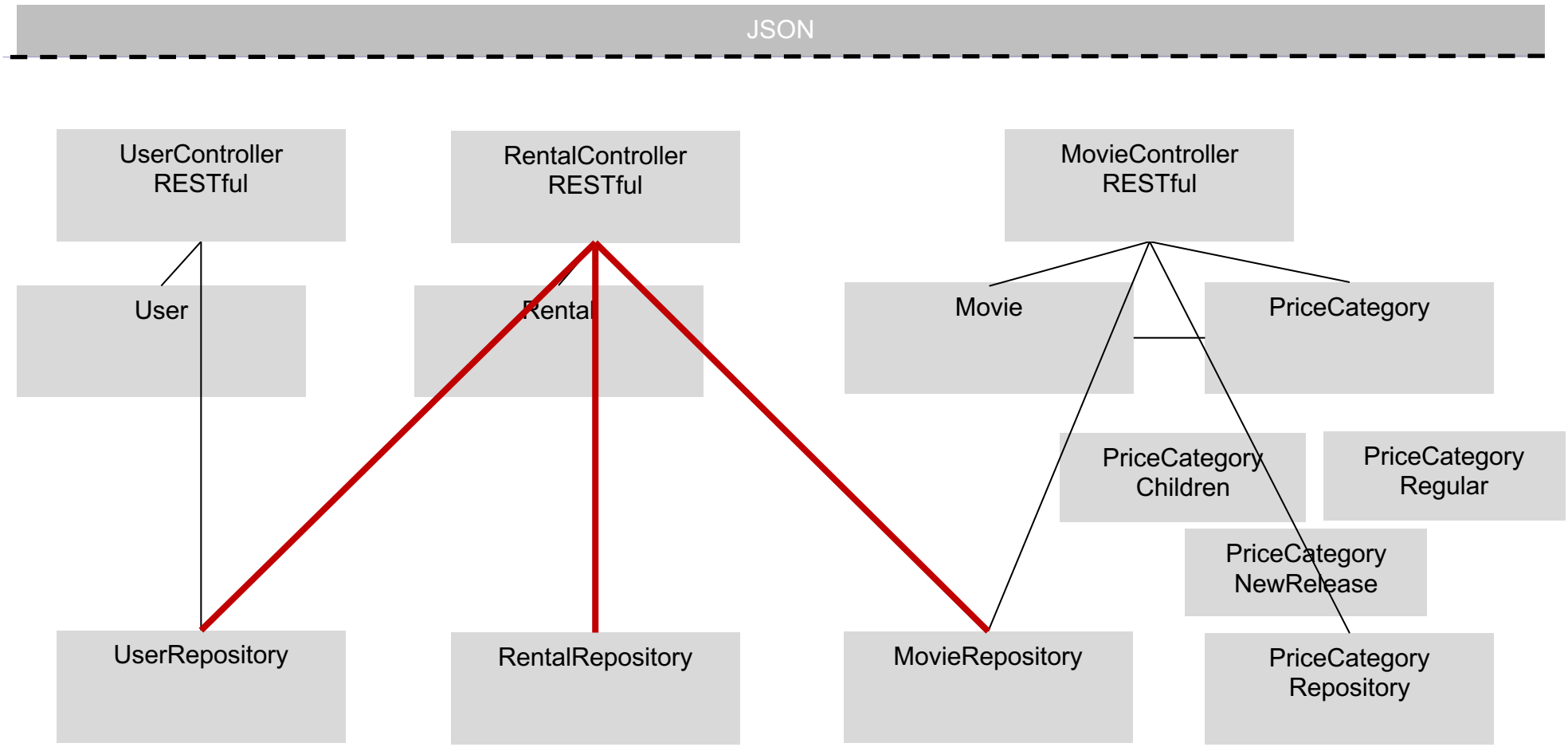
# Applikation "movierental" v2



# Übung 9 (2/2)

- Implementation des Aspekts
  - Advice implementieren
    - Advice Typ festlegen
      => @Before
    - Advice Logik implementieren
      => convert to lower case => delete white spaces
  - Pointcut definieren
    - AspectJ's Expression Language
    - Zugriff und Edit des Input-Parameters
      => MovieDto => dto

```
@Before("execution(* *..MovieController.create(..)) && args(dto)")
public void checkPriceCategoryString(MovieDto dto) throws Throwable {
    log.debug("checkPriceCategoryString() called: " + dto.getPriceCategory());
    dto.setPriceCategory(dto.getPriceCategory().toLowerCase().replaceAll("\\s", ""));
}
```

# Cache Support

## 1.2. What is Caching?

The term Caching is ubiquitous in computing. In the context of application design it is often used to describe the technique whereby application developers utilize a separate in-memory or low-latency data-structure, a Cache, to temporarily store, or cache, a copy of or reference to information that an application may reuse at some later point in time, thus alleviating the cost to re-access or re-create it.

In the context of the Java Caching API the term Caching describes the technique whereby Java developers use a Caching Provider to temporarily cache Java objects.

*It is often assumed that information from a database is being cached. This however is not a requirement of caching. Fundamentally any information that is expensive or time consuming to produce or access can be stored in a cache. Some common use cases are:*

- *client side caching of Web service calls*
- *caching of expensive computations such as rendered images*
- *caching of data*
- *servlet response caching*
- *caching of domain object graphs*

from **Java Caching API** (JSR-107)

# Caching mit Spring

- stellt eine Abstraktion bereit für verschiedene Implementationen:
  - JDK based caches, Ehcache 2.x, JSR-107 compliant cache (Ehcache 3.x), ...
- basiert auf Spring AOP
- unterstützt deklarative Beschreibung

| | |
|---|---|
| @Cacheable: | Triggers cache population. |
| @CacheEvict: | Triggers cache eviction. |
| @CachePut: | Updates the cache without interfering with the method execution. |
| @Caching: | Regroups multiple cache operations to be applied on a method. |
| @CacheConfig: | Shares some common cache-related settings at class-level. |

- unterstützt JCache (JSR-107) Annotationen

# Default Key Generation

- Caches sind im Wesentlichen **Key-Value Stores.**
- Es braucht immer einen **Key** für den Cache-Zugriff.
- Der Key muss aus der Signatur einer entsprechenden Methode generiert werden, durch:
  - "Default Key Generation" in Spring:
    - If no params are given, return SimpleKey.EMPTY.
    - If only one param is given, return that instance.
    - If more than one param is given, return a SimpleKey that contains all parameters.
  - Einsatz von SpEL (Spring Expression Language)

```
@Cacheable(cacheNames="books", key="#isbn")
public Book findBook(ISBN isbn, boolean checkWarehouse, boolean includeUsed)
```

# Email Support

- Das **JavaMail-API** bietet ein plattformunabhängiges und protokollunabhängiges Framework zum Erstellen von Mail- und Messaging-Anwendungen => **javax.mail.***
- Das Spring Framework vereinfacht den Umgang mit dem Mailing durch eine entsprechende Library => **org.springframework.mail.***
- Spring Boot ergänzt diesen Email Support u.a. mit Auto-Configuration:
  - Starter-Modul "spring-boot-starter-mail"
  - JavaMailSender als Spring Bean (als Default)
  - **Properties spring.mail.*** für eine einfache Konfiguration über das File "application.properties"

- Setup:

| | | |
|---|---|---|
| Mail Client | => | SMTP Server |
| **JavaMailSender** | **=>** | **SMTP Server (FHNW, Gmail, ...)** |

# Background Tasks & Scheduling Support

- Bei Geschäftsanwendungen kann eine zeitgesteuerte Ausführung (Scheduling) von Aufgaben benötigt werden. Dabei unterscheidet man beim Scheduling prinzipiell zwischen zwei Arten:
  - eine Aufgabe zu einem bestimmten Zeitpunkt ausführen
    z.B.: am 24.12.2000, um 18:00 Uhr
  - Aufgaben in festgelegten zeitlichen Abständen wiederholt ausführen
    z.B.: alle 10 Minuten
- Spring unterstützt dieses Scheduling folgendermassen:
  - **Begrenzt auf Spring Beans**
  - **Method-Annotation @Scheduled** mit
    - Methode mit "void" als Return-Wert
    - Methode ohne Input-Parameter
  - @Scheduled mit genau einem Parameter aus #cron, #fixedReate oder #fixedDelay
  - **@EnableScheduling** in der Konfigurationsklasse

# Unit Testing RESTController (1/3)

- Einsatz Mock-Objekte
  - Mock-Objekte initialisieren
    **Welche Abhängigkeiten bestehen? Für welche braucht es ein Mock-Objekt?**
  - Mock-Objekte vorbereiten, "bespielen"
    **Wie soll ein Mock-Objekt antworten, wen es innerhalb des Tests aufgerufen werden?**
  - Unit-Test ausführen
    **Wie kann ein HTTP-Request ausgelöst werden?**
  - Resultat überprüfen
    **Wie kann die HTTP-Response überprüft werden?**
  - Mock-Objekte verifizieren
    **Wie kann überprüft werden, dass die Abhängigkeiten korrekt aufgerufen wurden?**

# Unit Testing RESTController (2/3)

- **Beispiel eines Unit-Tests: RentalController.findById(...)**

  public ResponseEntity<RentalDto> **findById**(@PathVariable Long id) ...

  RentalControllerTest.findById_RentalFound_ShouldReturnFound() ...

- **Neuer Unit-Test: RentalController.create(...)**

  public ResponseEntity<RentalDto> **create**(@RequestBody RentalDto dto) ...

# Unit Testing RESTController (3/3)

1. Welche Abhängigkeiten bestehen? Für welche braucht es ein Mock-Objekt?
2. Wie soll ein Mock-Objekt antworten, wen es innerhalb des Tests aufgerufen werden?
3. Wie kann ein HTTP-Request ausgelöst werden?
4. Wie kann die HTTP-Response überprüft werden?
5. Wie kann überprüft werden, dass die Abhängigkeiten korrekt aufgerufen wurden?

# Docker und mehrere Applikationen

- "movierental" v3 besteht aus:
  - **movierental** webservice
  - **mysql** database
  - **phpmyadmin** webapp

- "docker-compose", um
  - mehrere Container gleichzeitig zu starten.
  - ein gemeinsames Netzwerk zu erzeugen.

# File "docker-compose.yml"

```
version: "3.7"

volumes:
    db-data:

services:
    mysql: ...

    phpmyadmin: ...

    movierental:
        build: .
        container_name: movierental
        ports:
            - 8080:8080
        depends_on:
            - mysql
        environment: # Pass environment variables to the service
            SPRING_DATASOURCE_URL: >
                jdbc:mysql://mysql:3306/eaf?useSSL=false&serverTimezone=UTC&useLegacyDatetimeCode=false
            SPRING_DATASOURCE_USERNAME: eaf
            SPRING_DATASOURCE_PASSWORD: eaf
        networks:
            - backend:

networks:
    backend:
```

Volume für MySQL Daten

MySql Container

phpMyAdmin Container

Zugang zum Dockerfile

externer:interner Port

Abhängigkeit von mysql

Environment Variablen

internes Netzwerk

# Starten der Container mit "docker-compose"

```
$ docker-compose build

# start all apps
$ docker-compose up -d
$ docker-compose logs –f
Ctrl^C

# show network
$ docker network ls
$ docker network inspect movierental_backend

# stop all apps
$ docker-compose down
```