

Spring Boot

as Enterprise Java Framework

Dr. Andreas Schönberger, Johannes Manner



Agenda

Part I: A bit of history of Java Enterprise

- Java EE and application containers
- Implicit middleware

Part II: Building a full stack app with Spring

- Spring Framework and Spring Boot
- Spring Initializr
- Lombok, Spring Web & Thymeleaf (creating backend & frontend)
- Spring Data JPA (extending backend to use a relational database)

Part III: Getting ready for operations

- Spring Security
- Spring HATEOAS and REST
- Deployment options
- Testing

Java EE / Jakarta EE

The classical Java Enterprise Platform

Java EE as Middleware Technology

- Java EE = Java Enterprise Edition (formerly: J2EE)

Definition (Java Glossary):

“The edition of the Java platform that is targeted at enterprises to enable development, deployment, and management of multi-tier server-centric applications.”

Goals (Src: Java EE 8 Spec.):

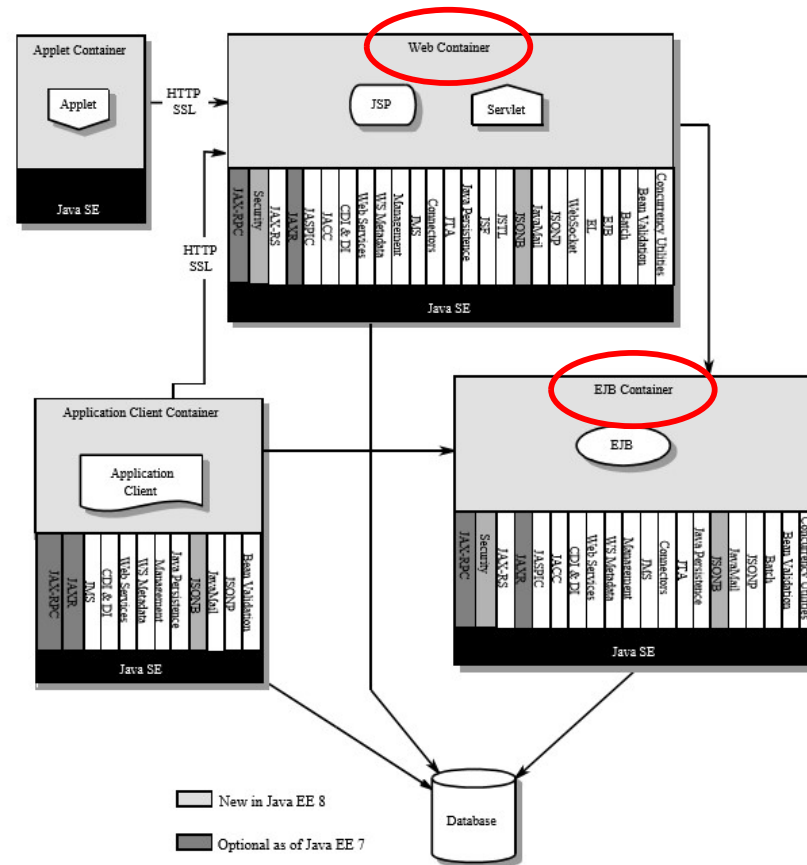
- “Enterprises today need to extend their reach, reduce their costs, and lower the response times of their services to customers, employees, and suppliers.
- Typically, applications that provide these services must combine existing enterprise information systems (EISs) with new business functions that deliver services to a broad range of users. The services need to be:
 - *Highly available*, to meet the needs of today’s global business environment.
 - *Secure*, to protect the privacy of users and the integrity of the enterprise.
 - *Reliable and scalable*, to ensure that business transactions are accurately and promptly processed.” [no change compared to JEE 7]

Java EE as Technology Toolkit

Java EE defines a broad set of technologies and APIs for client and server applications.

Support depends on the environment!

- Client container
- Web container
- EJB container

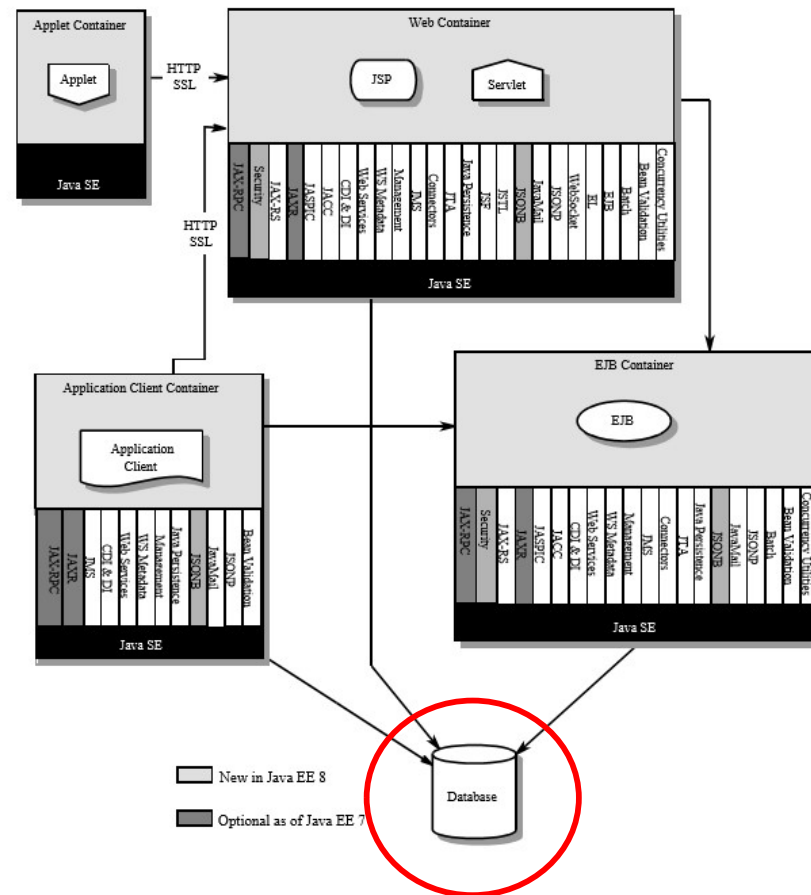


Src: Java EE 8 Spec

Java EE – the Database is Key

Java EE 8 Specification:

“The Java EE platform requires a database, accessible through the JDBC API, for the storage of business data. The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.”



Src: Java EE 8 Spec

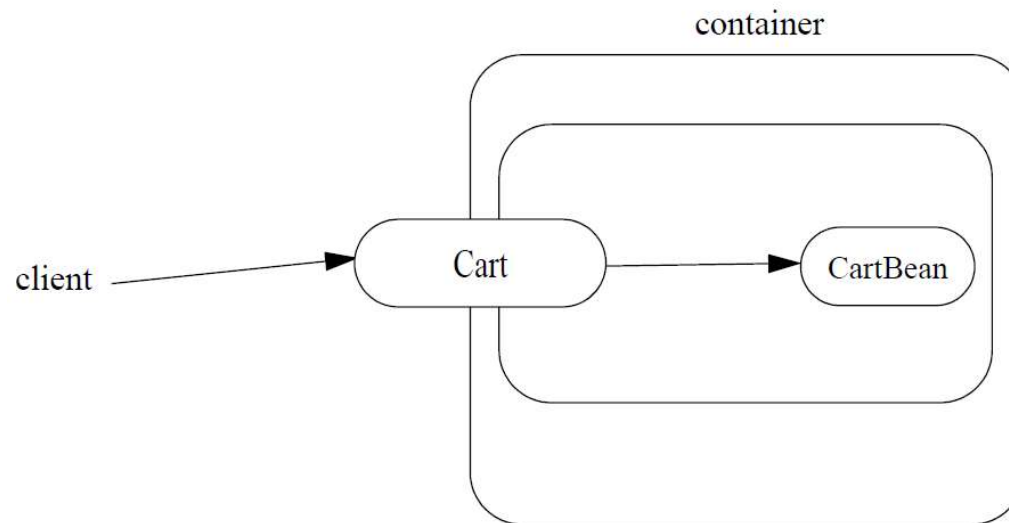
Java EE – JCP and Community

- Java EE != Implementation
- Java EE is a specification
 - Supported by “numerous” vendors
 - Vendor independence
- Java Community Process (JCP) defines(d) the specifications
Now moved to Jakarta EE
- Solution vendors provide application servers
 - GlassFish (Java EE / Jakarta EE Community version)
 - Oracle WebLogic
 - IBM Websphere Application Server (Commercial or Community)
 - (Red Hat) JBoss Enterprise Application Platform
 - ...
- Application Server = Java EE implementation (formerly)
→ today: Application Server = Any Enterprise Framework

Java EE – Core Paradigm

Core Principles:

1. Implicit Services
2. Interceptors
3. Dependency Injection
4. Annotations



Src: EJB 3.2 Spec

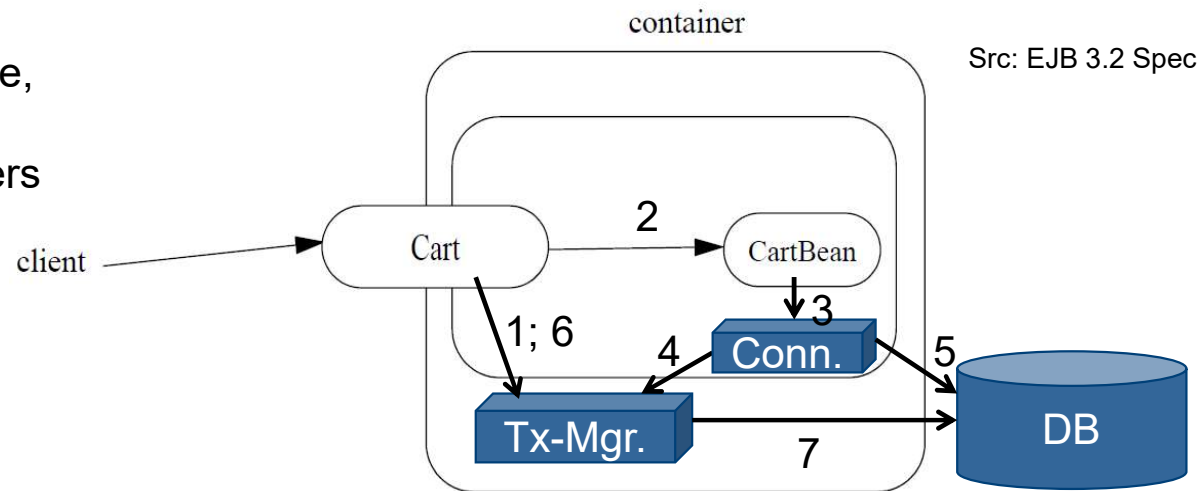
EJB 3.2 Specification:

“The Enterprise JavaBeans architecture will make it easy to write applications: application developers will not have to understand low-level transaction and state management details, multi-threading, connection pooling, or other complex low-level APIs.”

Java EE – Transactions as Implicit Service

Schematic presentation!

This is the simple case,
think of multiple dbs,
ejb modules, containers



Java EE 8 Specification:

“This specification does not require the Product Provider to implement any particular protocol for transaction interoperability across multiple Java EE products.

*Java EE compatibility requires **neither interoperability among identical Java EE products from the same Product Provider**, nor among heterogeneous Java EE products from multiple Product Providers.”*

Building a full stack app with Spring

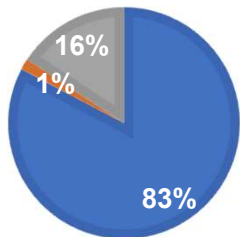
part II

Why Spring and not Java EE / Jakarta EE?

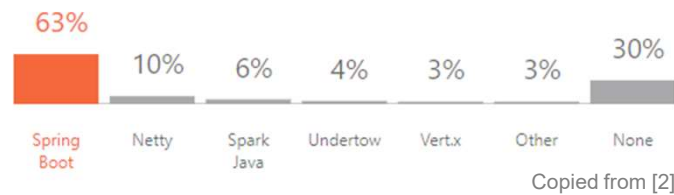
Why Spring Boot?

JAVA RUNTIME PLATFORMS [1]

■ Spring Boot ■ Micronaut ■ Others

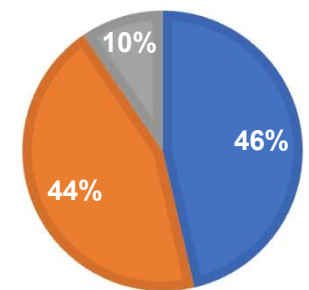


Which frameworks do you use as an alternative to an application server?

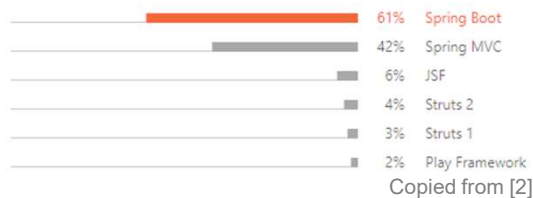


DEPLOYMENT MODELS [1]

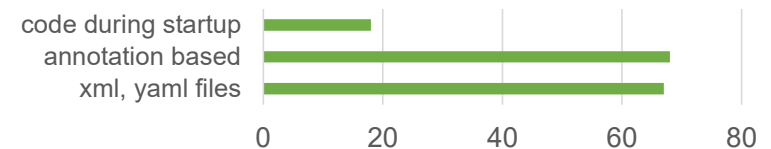
■ JAR ■ App Server ■ Others



What web frameworks do you use?



Framework Configuration in % [1]



[1] Jrebel: 2020 Java Technology Report: Diagrams are generated out of 400 responses from Java development professionals

Source: <https://www.jrebel.com/blog/2020-java-technology-report>

[2] JetBrains developer survey: <https://www.jetbrains.com/lp/devecosystem-2020/java/>

What is (early) Jakarta EE?

- First enterprise specification extending Java SE
- Formerly J2EE (1999-2006) and Java EE (2006-2019)
- Full-blown application servers (e.g. Glassfish, Wildfly) and servlet containers
- A lot of configuration effort has to be done, e.g. web.xml
- EJB (Java EE specification including e.g. concurrency, security) based programming model (needs an app server to work)

What is Spring?

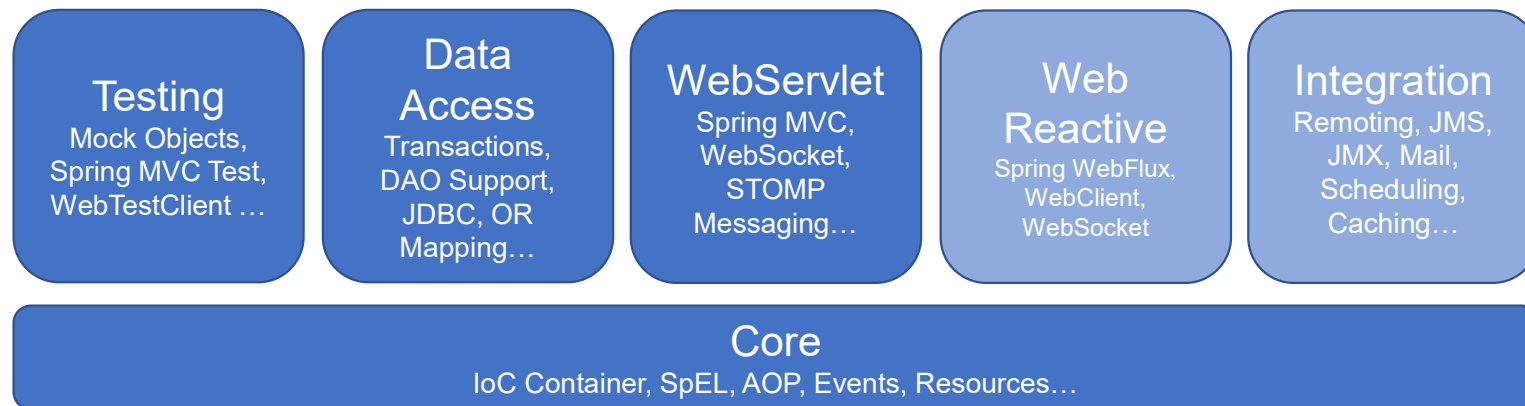
- Enterprise framework, ideas based on J2EE, Java EE (selected specifications from the EE umbrella)
- First release in 2003 under Apache 2.0 license, written by Rod Johnson
- Servlet container like Tomcat
- Annotation based configuration and via properties and profiles
- POJO based programming model (framework does the work, e.g. concurrency, security etc.)

Both ecosystems try to support developers writing enterprise code, but Spring's passion is to make enterprise coding easier and more transparent.

<https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html#overview>

Spring Framework

- Spring Framework is only the base project in the Spring Ecosystem. Extension projects include **Spring Boot**, Spring Security... All projects have a separate code base.
- Design Philosophy: Backward Compatibility
- Design Philosophy: Late decisions: E.g. switching the database provider by configuration without changing your code.



<https://spring.io/projects>

<https://docs.spring.io/spring-framework/reference/>

Preliminary: Annotations

- Introduced with Java 1.5
- Are a form of metadata, kind of “declarative” programming where programmer says what to do and the compiler/tools/runtime generate the code to do it
- Most important use cases
 - Information for the compiler (e.g. `@SuppressWarnings`)
 - Compile-time processing (e.g. Lomboks `@Data` – also look at the build.gradle – Lombok only used during compile time)
 - Runtime Processing (e.g. `@Profile` – a Spring annotation which profile is active – shown at the next slides)
- Some Built-in Annotations (you may have already seen them)
 - `@Override` – specifying that you override a method from a inherited class
 - `@SuppressWarnings` – ignore some compiler warnings
 - `@Deprecated` – useful when parts of the API should not be used any more (retrofitted in Java 9 with more information)
 - `@FunctionalInterface` – supporting Lambdas introduced with Java 8

<https://docs.oracle.com/javase/tutorial/java/annotations/>
<https://www.baeldung.com/java-default-annotations>

First Spring Annotations – demystifying first concepts

- `@Configuration` is used by default to indicate components of the application (an instance of these classes will be instantiated by the middleware)
- `@Configuration` indicates that this class contains bean definitions
- `@Bean` annotation specifies the factory method, where the object is instantiated

*These beans are no
JavaEE beans*

```
@Configuration
public class Vehicles {
    @Bean
    public Vehicle getBicycle(){
        return new Bicycle();
    }
}
```

- `@Value("${PROPERTY-NAME}")` - reading properties from property files at runtime



Overwhelmed? → No problem, wait for a few minutes and keep calm

Running Example I

```
@Configuration
public class Vehicles {
    @Bean
    public Vehicle getBicycle(){
        return new Bicycle();
    }
}
```

Vehicles identified during classpath scanning
(we will discuss this in a few slides)

Bean is instantiated and managed by the IoC
container (next slide)

Self-instantiating the Vehicle, in our case a
Bicycle

```
public interface Vehicle {
    public String getWheelInfo();
}
```

```
public class Bicycle implements Vehicle {

    @Value("${vehicle.wheels}")
    private int wheels;

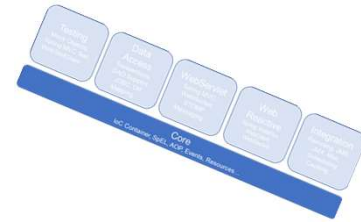
    @Override
    public String getWheelInfo() {
        return "..." + this.wheels + " wheels";
    }
}
```

application.properties

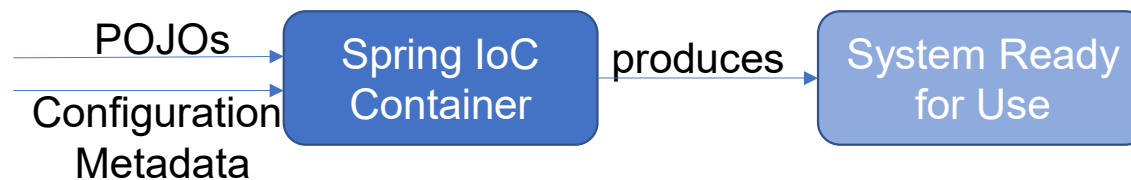
```
server.port=7777
vehicle.wheels=2
```

Source code is available at [samples/dependency-injection](#)

Spring Framework Core - Dependency Injection



- Inversion of Control (IoC) also known as Dependency Injection (DI)
- Higher level of decoupling: Objects do NOT know the location of their attributes
- Objects define their dependent attributes via constructor arguments or setters
- *IoC Container* injects the dependencies when creating the object
- *Beans* are objects managed by the Spring IoC container



- Configuration Metadata: XML-based configuration, **Annotation based configuration (@Autowired)**
- Spring generally favors constructor injection

<https://docs.spring.io/spring-framework/reference/core.html>

Running Example II

Copied from Running Example I

```
@Configuration
public class Vehicles {
    @Bean
    public Vehicle getBicycle(){
        return new Bicycle();
    }
}
```

```
@RestController
@RequestMapping(value = "vehicle")
public class VehicleController {

    private Vehicle vehicle;

    @Autowired
    public VehicleController(Vehicle vehicle){
        this.vehicle = vehicle;
    }

    @GetMapping
    public String getInfo(){
        return vehicle.getWheelInfo();
    }
}
```

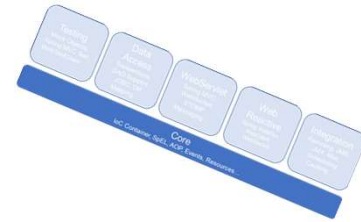
→ Necessary to invoke functionality via browser
→ <http://localhost:7777/vehicle> (see later slides)

→ IoC container managed bean is injected here
(in our case a Bicycle object)

→ Necessary to invoke functionality via browser
→ <http://localhost:7777/vehicle> (see later slides)

Source code is available at [samples/dependency-injection](#)

Profiles and Properties



- Different situations require different properties, think about running your app in dev and prod
- Profiles are the spring answer to this problem by defining suited properties, e.g. in application-dev.properties and application-prod.properties
- Via @Configuration, @Bean and @Profile: possibility to create different objects dependent on the current profile (→ next slide)
- Methods with @Profile to create single beans are also possible
- VM arguments to start the application with a comma separated list of profiles
-Dspring.profiles.active="profile1,profile2"
- When nothing is specified, "default" is used (resulting in usage of application.properties)

<https://docs.spring.io/spring-framework/reference/core/beans/environment.html>

Running Example III

Extended compared to Running Example I

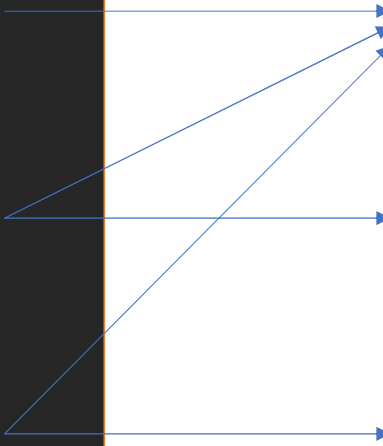
```
@Configuration
public class Vehicles {

    @Bean
    @Profile("default")
    public Vehicle getBicycle(){
        return new Bicycle();
    }

    @Bean
    @Profile("dev")
    public Vehicle getTricycle(){
        return new Tricycle();
    }

    @Bean
    @Profile("prod")
    public Vehicle getCar(){
        return new Car();
    }

}
```



application.properties

```
server.port=7777
vehicle.wheels=2
```

application-dev.properties

```
vehicle.wheels=3
```

application-prod.properties

```
vehicle.wheels=4
```

Source code is available at [REPO/samples/dependency-injection](#)

Classpath Scanning

- **Classpath** is **scanned** at startup and all *beans* and *components* are instantiated by the framework resulting in the **ApplicationContext** (sum of all configured components and beans)
- Most of the beans at runtime come from the included dependencies (that's the reason for the autoconfiguration magic you see)
- Specialized annotations for different layers of the application

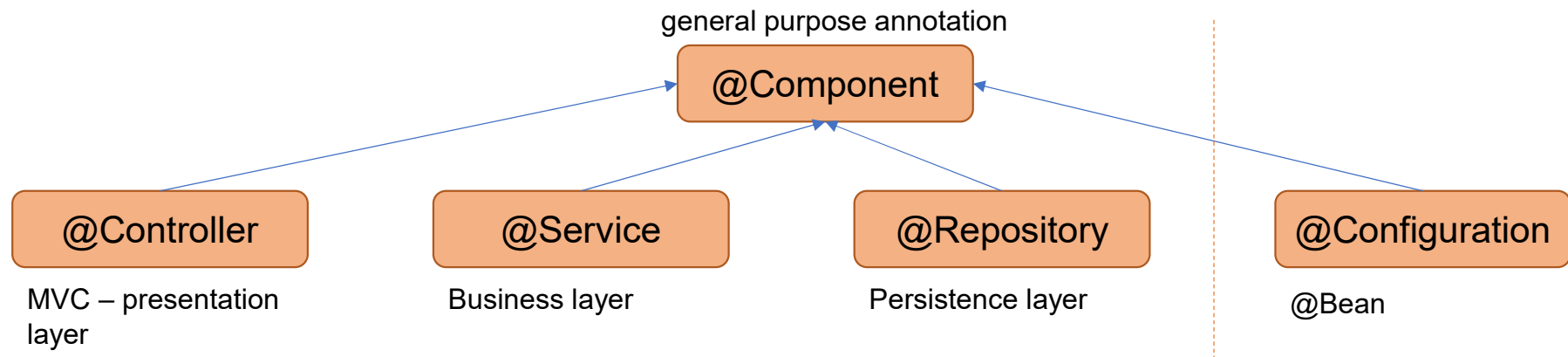
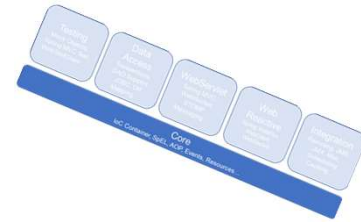


Figure inspired by <https://springbootdev.com/2017/07/31/spring-framework-component-service-repository-and-controller/>
<https://docs.spring.io/spring-framework/reference/core/beans/classpath-scanning.html>

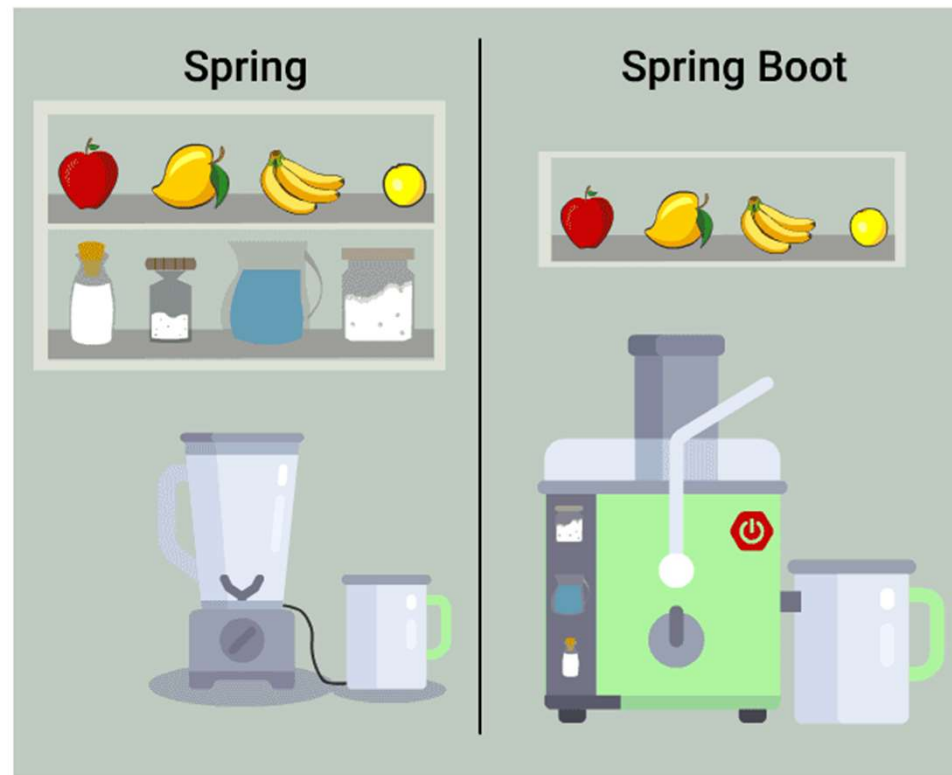


First Summary: What have we seen so far? Important Spring Annotations

- `@Service`, `@Controller`, `@Repository` are specializations of generic `@Component`
- These annotations and `@Configuration` are used by default by the classpath scanning process to indicate components of the application
- Their corresponding behavior (the annotation's semantic) is executed at runtime
- E.g. `@Configuration` indicates that this class contains bean definitions
- `@Bean` annotation specifies the factory method, where the object is instantiated. Beans also have a lifecycle (`@PostConstruct` and `@PreDestroy` methods can be specified)
- Composed annotations for ease of usage, e.g. `@RestController` is composed of `@Controller` and `@ResponseBody`
- `@Value("${PROPERTY-NAME}")` - reading properties from property files at runtime

<https://docs.spring.io/spring-framework/reference/core/beans/classpath-scanning.html>
<https://docs.spring.io/spring-framework/reference/core/beans/java/basic-concepts.html>

Spring vs. Spring Boot



Src and copyright: https://img.devrant.com/devrant/rant/r_1867059_KBtFw.gif

Spring Boot at a glance

- Extension of the spring framework (spring framework is complex to use & configure)
- Eliminating boilerplate configuration for setting up spring application - autoconfiguration
- Property based configuration
- Build dependency management via starters
- Integrated embedded server, per default Tomcat
- Resolves application context: Servlet, Filter and ServletContextInitializer
- Scanning the classpath and identifying candidate components (beans)

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

<https://www.baeldung.com/spring-vs-spring-boot>

Spring Boot – Autoconfiguration and Starters

“I’d like to show you some example code that demonstrates autoconfiguration. But I can’t. You see, autoconfiguration is much like the wind. You can see the effects of it, but there’s no code that I can show you [...] It’s this lack of code that’s essential to autoconfiguration and what makes it so wonderful.”

(Craig Walls: Spring in Action, fifth edition, Manning Publications, 2019, page 6).

- Autoconfiguration (beyond component scanning and dependency injection) uses knowledge contained in the classpath, environment variables etc. to decide which components are needed and how they interact with each other/wired together.
- A lot of configuration parameters have reasonable defaults, this is why it’s called auto (mostly you do not have to alter the config)
- Starters are a smart way of specifying build dependencies. You can include it as a single dependency in your build file. Transitively a set of dependencies with aligned versions are included in your project.
- Spring Boot’s naming convention for starters: spring-boot-starter-*
When defining your own starter, do not use spring-boot as prefix

<https://docs.spring.io/spring-boot/docs/3.1.4.RELEASE/reference/pdf/spring-boot-reference.pdf>

Spring Boot Starters

Descriptions are taken from the docs or the spring initializr website

- **spring-boot-starter** Core starter, including auto-configuration support, logging and YAML
- **spring-boot-starter-web** Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container
- **spring-boot-starter-thymeleaf** Starter for building MVC web applications using Thymeleaf views
- **spring-boot-starter-test** Starter for testing Spring Boot applications with libraries including Junit, Hamcrest and Mockito
- **lombok** Java annotation library which helps to reduce boilerplate code
- **spring-boot-devtools** Provides fast application restarts, LiveReload, and configurations for enhanced development experience.

Source Code: <https://github.com/spring-projects/spring-boot/tree/master/spring-boot-project/spring-boot-starters>

Comprehensive List of Starters: <https://docs.spring.io/spring-boot/docs/3.1.4/reference/htmlsingle/#using.build-systems.starters>

Spring Boot Starter Web Example

Dependency tree, when including spring-boot-starter-web in your build.gradle

spring-boot-starter-web (<https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-starters/spring-boot-starter-web/build.gradle>)

```

org.springframework:spring-web
org.springframework:spring-webmvc
project(":spring-boot-project:spring-boot-starters:spring-boot-starter")
    org.yaml:snakeyaml
    jakarta.annotation:jakarta.annotation-api
    org.springframework:spring-core
    project(":spring-boot-project:spring-boot") (https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot/build.gradle)
    ...
    project(":spring-boot-project:spring-boot-autoconfigure") (https://github.com/spring-projects/spring-boot/blob/master/spring-boot-project/spring-boot-autoconfigure/build.gradle)
    ...
    project(":spring-boot-project:spring-boot-starters:spring-boot-starter-logging") (...)
    ...
project(":spring-boot-project:spring-boot-starters:spring-boot-starter-json") (...)
    ...
project(":spring-boot-project:spring-boot-starters:spring-boot-starter-tomcat") (...)
    ....

```

**60 external libraries are included in your app when including this single dependency
AND all versions and dependencies are compatible to each other!!**

Enough theory for now!

**Get your hands dirty,
but how?**

**Spring Intializr, Lombok
Spring Web & Thymeleaf**

Spring Initializr – start.spring.io

The screenshot shows the Spring Initializr web application in a browser window. The interface is dark-themed and organized into several sections:

- Project:** Includes radio buttons for **Gradle - Groovy** (selected), **Gradle - Kotlin**, and **Maven**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions: 3.2.0 (SNAPSHOT), 3.2.0 (M3), 3.1.5 (SNAPSHOT), **3.1.4** (selected), 3.0.12 (SNAPSHOT), 3.0.11, 2.7.17 (SNAPSHOT), and 2.7.16.
- Project Metadata:** Includes text input fields for **Group** (com.example), **Artifact** (demo), **Name** (demo), **Description** (Demo project for Spring Boot), and **Package name** (com.example.demo).
- Packaging:** Includes radio buttons for **Jar** (selected) and **War**.
- Java:** Includes radio buttons for versions: 21, **17** (selected), 11, and 8.
- Dependencies:** A list of dependencies with a button **ADD DEPENDENCIES... CTRL + B**. The list includes:
 - Spring Boot Dev Tools** (DEVELOPER TOOLS): Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
 - Lombok** (DEVELOPER TOOLS): Java annotation library which helps to reduce boilerplate code.
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Thymeleaf** (TEMPLATE ENGINES): A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

At the bottom, there are three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**

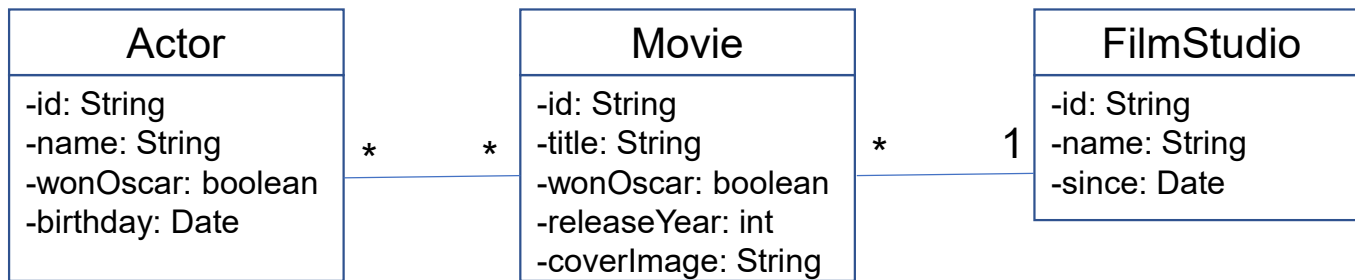
Spring Initializr

- Website to initialize your spring boot project by selecting the build tool, JVM language, Spring Boot version and metadata
- Dependency section is used for specifying the starters to include in the project
- Later, we will use further spring boot starters, but for now to build a simple web application, we use Spring Boot DevTools*, Spring Web, Lombok and Thymeleaf

* DevTools monitor the classpath and restart when changes occur. In Eclipse this happens when saving a file, in IntelliJ when building the project. Use SaveActions plugin in IntelliJ and compile files experimental option (does currently result in a lot of false positives – building more often then required).

Our domain model for a Movie store

- Relations between the classes result in further attributes
- We will implement them later, when storing data in the database 😊



Excursion: Lombok for the lazy ones

- Compile time annotations to reduce boiler plate code
- Setters, getters, equals(), hashCode(), toString() etc. are generated at compile time
- Only thing you need is to annotate your model classes with `@Data`
- Setup Lombok plugin for your IDE
- Enable annotation processing in IntelliJ:
Settings > Build, Execution, Deployment > Compiler > Annotation Processors

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Movie {
    private String id;
    private String title;
    private boolean wonOscar;
    private int releaseYear;
    private String coverImage;
    private List<Actor> actors;
}
```

Only necessary annotation here to generate all necessary methods.

Other useful annotations are:

- `@AllArgsConstructor`
- `@RequiredArgsConstructor`
- `@NoArgsConstructor`
- `@Slf4j` (getting a logger)

Spring MVC – How does this work – GET request?

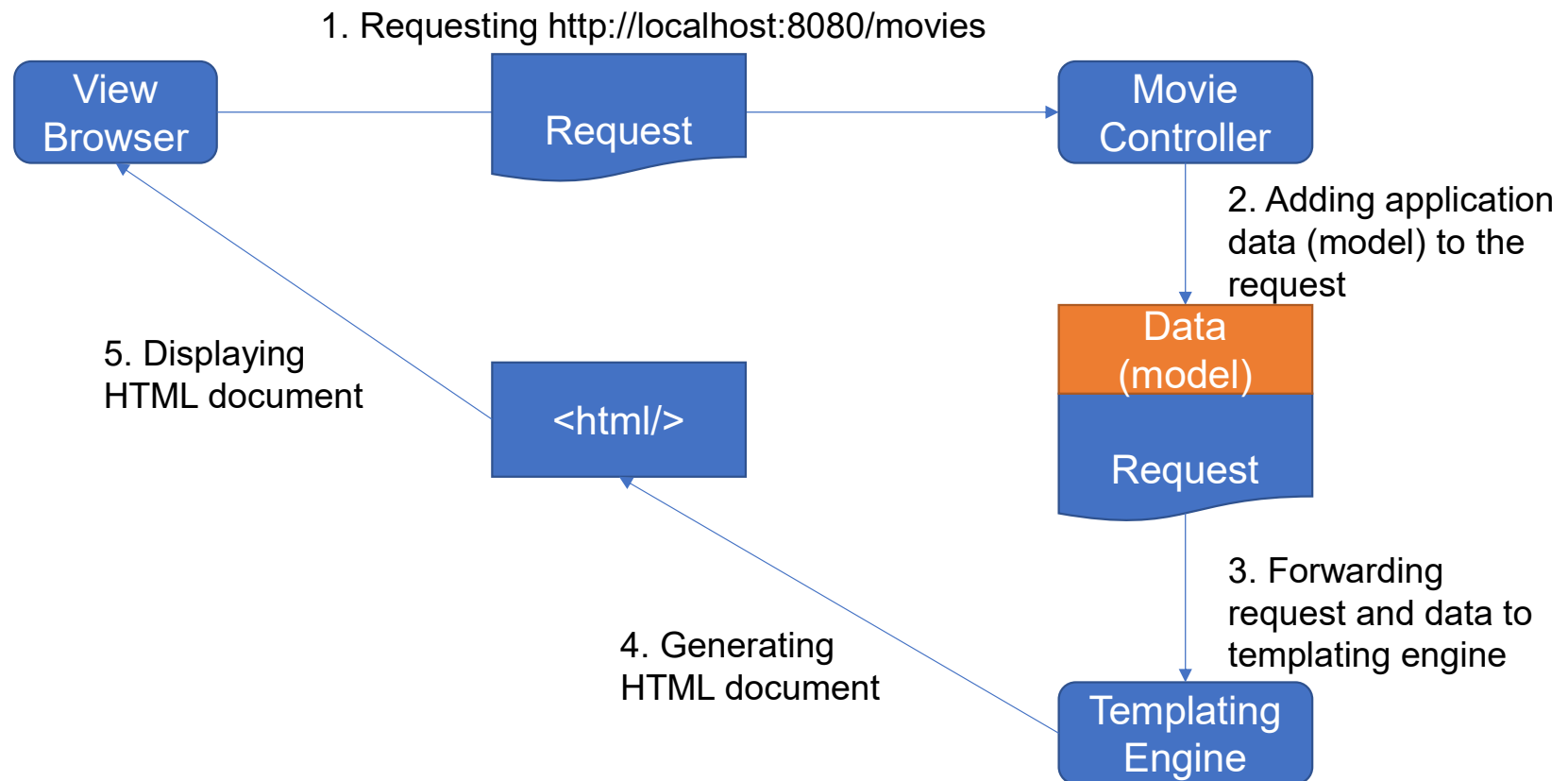


Figure adapted from Craig Walls: Spring in Action, fifth edition, 2019, page 30 (figure 2.1)

Spring MVC Request Mapping Annotations

<code>@RequestMapping</code>	General-purpose request handling
<code>@GetMapping</code>	Handles HTTP GET requests
<code>@PostMapping</code>	Handles HTTP POST requests
<code>@PutMapping</code>	Handles HTTP PUT requests
<code>@DeleteMapping</code>	Handles HTTP DELETE requests
<code>@PatchMapping</code>	Handles HTTP PATCH requests

- `@GetMapping` **easier readable than**
`@RequestMapping(method=RequestMethod.GET)`
- **You can specify the path attribute and media types via**
`@...Mapping(value = "path")`

Table copied from Craig Walls: Spring in Action, fifth edition, 2019, page 34

Spring MVC Controller (GET)

- Handle HTTP request and **(a) render HTML files** or (b) return data to user (RESTful)
- `@Controller`: mark class for component scanning, generating a bean
- Return value specifies the HTML template file name

```
@Slf4j
@Controller
@RequestMapping(value = "/movies")
public class MovieController {

    @GetMapping
    public String getMovies(Model model) {

        log.info("Client requested all movies");

        List<Movie> movies = Arrays.asList(
            new Movie("1", "Inception", ...)
        );

        model.addAttribute("movies", movies);
        model.addAttribute("movie", new Movie());

        return "movies";
    }
}
```

→ Logger provided by Lombok

→ Component scanning, bean instantiation

→ Endpoint accessible at host:port/movies

→ HTTP GET

→ Model is a springframework ui class, which serves as a basket to put data in and also get data from (we will see this when implementing a POST)

→ Data given to the templating engine

→ Placeholder for POST data

→ Name of the thymeleaf view



Thymeleaf – View template (GET)

- Thymeleaf is a HTML templating engine, decoupled from any web framework
- Spring copies `model` attribute to servlet attributes
- Put your HTML files under `src/main/resources/templates`
- If you have any static content, which is needed for the templates, put it in `src/main/resources/static` (Thymeleaf's `@{ }` operator resolves the content there)

```
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org">
...
<table>
  <thead>
    ...
  </thead>
  <tbody>
    <tr th:each="movie : ${movies}">
      <td></td>
      <td th:text="${movie.title}"></td>
      <td th:text="${movie.wonOscar}"></td>
      <td th:text="${movie.releaseYear}"></td>
    </tr>
  </tbody>
</table>
```

<https://www.thymeleaf.org/documentation.html>

First view

Cover Image	Title	Won an Oscar	Year
	Inception	false	2010
	Cloud Atlas	false	2012

Spring MVC – How does this work – POST request?

Numbering
continued from
slide 34

Overview

HTTP POST → Redirect → HTTP GET

In Detail

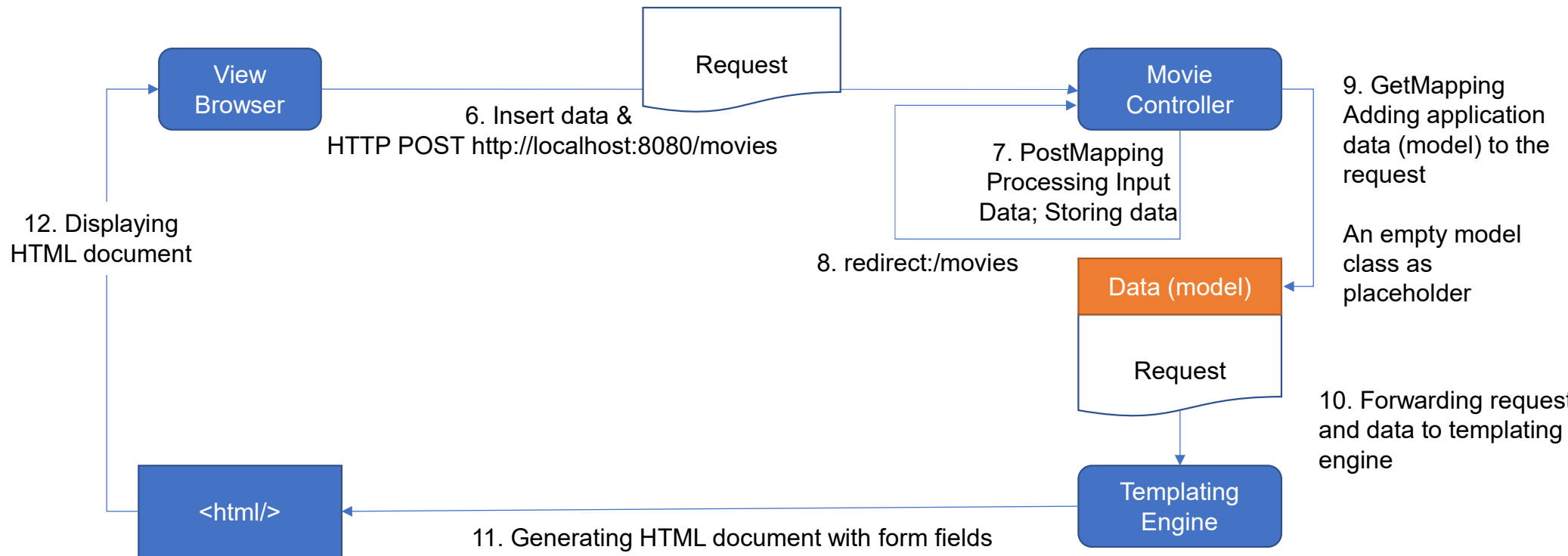


Figure adapted from Craig Walls: Spring in Action, fifth edition, 2019, page 30 (figure 2.1)

Spring MVC Controller (POST)

- Placeholder model is used by the framework to “inject” data
- Return value specifies the HTML template file name, prefixed with “redirect:”

```
@Slf4j
@Controller
@RequestMapping(value = "/movies")
public class MovieController {

    private final List<Movie> movies;

    @PostMapping
    public String processMovie(Movie movie) {
        // No validation
        log.info("... new movie: " + movie);

        movie.setId("" + (this.movies.size()+1));
        this.movies.add(movie);

        return "redirect:/movies";
    }
}
```

Annotations and their meanings:

- `@Slf4j`: Logger provided by Lombok
- `@Controller`: Component scanning, bean instantiation
- `@RequestMapping(value = "/movies")`: Endpoint accessible at host:port/movies
- `private final List<Movie> movies;`: In memory store – we will use a database later
- `@PostMapping`: HTTP POST
- `public String processMovie(Movie movie) {`: Framework injects the form data in movie (remember the placeholder in the GET request)
- `log.info("... new movie: " + movie);`: “Store” the movie
- `return "redirect:/movies";`: Redirect: Results in a HTTP GET to /movies

Thymeleaf – View template (POST)

```
<!DOCTYPE html>
<html lang="en"
      xmlns:th="http://www.thymeleaf.org">
...
<table class="table">
  <thead>
    ...
  </thead>
  <tbody>
    <form method="POST" th:action="@{/movies}" th:object="${movie}">
      <tr>
        <td>URL: <input name="coverImage" th:field="*{coverImage}" type="text"></td>
        <td><input name="title" th:field="*{title}" type="text"></td>
        <td><input name="wonOscar" th:field="*{wonOscar}" type="checkbox"></td>
        <td><input name="year" th:field="*{releaseYear}" type="text"></td>
        <td>
          <button>Create</button>
        </td>
      </tr>
    </form>
  </tbody>
</table>

</body>
</html>
```

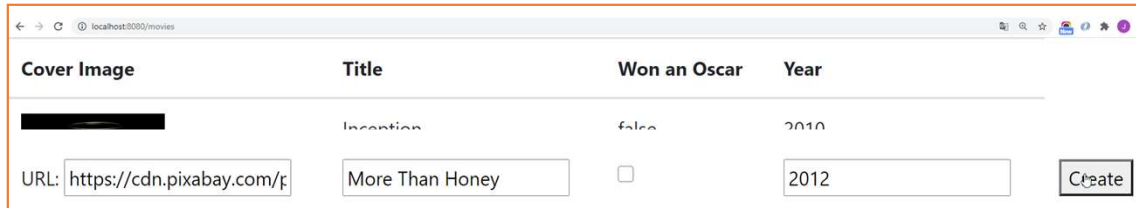
HTTP POST endpoint


Placeholder object

Mapping input to object (movie)'s attribute

<https://www.thymeleaf.org/documentation.html>

POSTing data to the server. . .

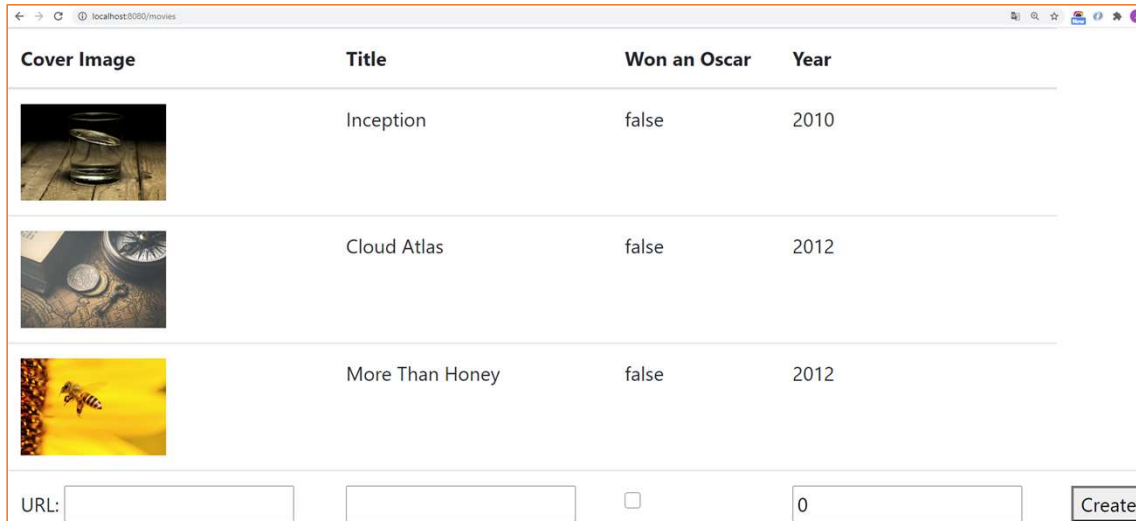





Cover Image	Title	Won an Oscar	Year
	Inception	false	2010
URL: <input type="text" value="https://cdn.pixabay.com/"/>	<input type="text" value="More Than Honey"/>	<input type="checkbox"/>	<input type="text" value="2012"/>

Create

HTTP POST

redirect:/movies
(executing GET request again)



Cover Image	Title	Won an Oscar	Year
	Inception	false	2010
	Cloud Atlas	false	2012
	More Than Honey	false	2012

URL: ☐ Create

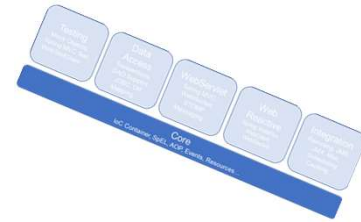
HTTP GET

New Movie 🐝

How to validate input data?

- One option is to do the validation in our `processMovie` method (having a lot of if-else blocks)
- Other option is using Java Bean Validation API (JSR-303, JSR-380)
Add `'org.springframework.boot:spring-boot-starter-validation'` to your `build.gradle`
- Declare validation at your model classes (`Movie`)
- Specify validation in your controller and change method signature of `processMovie`
- Adapt your view to display erros (`movies.html`)

<https://jcp.org/en/jsr/detail?id=303>
<https://jcp.org/en/jsr/detail?id=380>
<https://www.baeldung.com/javax-validation>



Validation

- Validator interface, where an Errors object is filled with the validation errors.
- Format your attributes via Format Annotation API, e.g. @DateTimeFormat.
- Spring supports Jakarta Bean Validation API:

```
public class PersonForm {  
    @NotNull  
    @Size(max=64)  
    private String name;  
  
    @Min(0)  
    private int age;  
}
```

- SpEL (Spring Expression Language): E.g. When reading system properties via @Value("\${data.text}")

<https://docs.spring.io/spring-framework/reference/core/validation/format.html#format-CustomFormatAnnotations>

<https://docs.spring.io/spring-framework/reference/core/validation/beanvalidation.html>

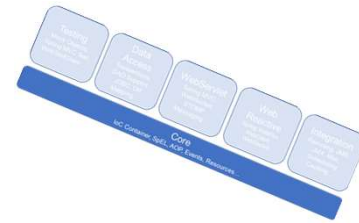
Specification: <https://beanvalidation.org/>

Implementation: <http://hibernate.org/validator/>

<https://docs.spring.io/spring-framework/reference/core/expressions/beandef.html#expressions-beandef-annotation-based>

Null Safety

- Annotation based solution to declare nullability
 - `@Nullable`: Indicates that a parameter, return value or field can be null
 - `@NonNull`: Indicates that a parameter, return value or field can NOT be null
 - `@NonNullApi`: Specified at package level, declares non-null for parameters and return values
 - `@NonNullFields`: Specified at package level, declares non-null for fields.



<https://docs.spring.io/spring-framework/reference/core/null-safety.html>

How to validate input data? - Movie

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Movie {
    private String id;
    @NotNull(message = "Title must be set")
    @NotEmpty(message = "Title not there")
    private String title;
    private boolean wonOscar;
    @Min(value = 1920, message = "Movies before 1920 are not considered!")
    @Max(value = 2022, message = "Movies after 2022 are not planned now!")
    private int releaseYear;
    @NotNull
    @Pattern(regexp = "(https:\\\\|\\\\/).*\\.?(?:jpg|gif|png)"
            , message = "Must be a valid URL to a picture.")
    private String coverImage;
    private List<Actor> actors;
}
```

jakarta.validation annotations;
message can be displayed via
Spring's Errors bean

Java regex pattern matching

How to validate input data? – MovieController

```
@Slf4j
@Controller
@RequestMapping(value = "/movies")
public class MovieController {

    private final List<Movie> movies;

    @PostMapping
    public String processMovie(@Valid Movie movie,
                               Errors errors,
                               Model model) {
        log.info("Client POSTed a new movie: " + movie);

        if(errors.hasErrors()){

            model.addAttribute("movies", this.movies);

            log.info(" . . . but there are errors included: " + movie);
            return "movies";
        }

        movie.setId("" + (this.movies.size()+1));
        this.movies.add(movie);

        return "redirect:/movies";
    }
}
```

→ Activate jakarta.validation via @Valid

→ Validation errors are included in Errors bean

→ Springframework UI Model bean

→ Check if validation errors are present

→ Add movies again, otherwise only form elements are presented

→ Return view name and display errors

How to validate input data? – View

```
<!DOCTYPE html>
...
<table class="table">
  ...
  <form method="POST" th:action="@{/movies}" th:object="${movie}">
    <tr>
      <td>
        URL: <input name="coverImage" th:field="*{coverImage}" type="text"/>
        <span th:if="${#fields.hasErrors('coverImage')}}" th:errors="*{coverImage}"/>
      </td>
      <td>
        <input name="title" th:field="*{title}" type="text">
        <span th:if="${#fields.hasErrors('title')}}" th:errors="*{title}"/>
      </td>
      <td><input name="wonOscar" th:field="*{wonOscar}" type="checkbox"></td>
      <td>
        <input name="year" th:field="*{releaseYear}" type="text">
        <span th:if="${#fields.hasErrors('releaseYear')}}" th:errors="*{releaseYear}"/>
      </td>
      <td>
        <button>Create</button>
      </td>
    </tr>
  </form>
  ...
</html>
```

Displaying the validation error for field `coverImage`

Only displaying the error, when a validation error is present for this field

(you can somehow program with `thymeleaf` as well)

Situation: Currently no way to write our domain data in a database

Spring Data

Spring Data – JDBC and JPA – Prerequisites

- JDBC (Java Database Connectivity) via JdbcTemplate (we won't use this)
- JPA (Java Persistence API) (we will use this, includes JDBC transitively ☺)
- Adding development database to your build
- Adding Spring Data JPA to your project (there are also separate starters for MongoDB, Redis, Neo4J, Cassandra to name a few)
- Hibernate is used as the default ORM (Object <--> Relational mapping) implementation

Persisting first movie entity

- Annotate your movie class with `@Entity`
- Annotate your id attribute with `@Id`
- Create repository interfaces extending `CrudRepository<Model, ID-Attribute>`
- Uncomment the `List<Actor>` attribute (we explain needed annotations later)
- Inject your repository into your controller



Working with NoSQL databases: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-nosql>

CrudRepository (Create Read Update Delete)

- Repository classes enable access to the database via Hibernate
- Spring Data JPA offers some interfaces where functionality is generated from the middleware (think about Lombok and the generation of getters, setters etc.)
- To test all this, we also need a database (H2 is a perfect development database with a comfortable web interface)
- Add following dependencies to your build.gradle

```
implementation 'com.h2database:h2'  
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
```

- We will see on the next slides, how to adapt your model classes via annotations to enable an ORM (Object Relational Mapping) mapper to find, store and delete your objects
- You can also use JpaRepository (extends CrudRepository and contains some methods for sorting etc.)

<https://www.baeldung.com/spring-data-repositories>

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

CrudRepository – method overview

- CRUD (Create, Read, Update, Delete) repository offers you the following methods by default:

```
public interface CrudRepository<T, ID> extends
Repository<T, ID> {

    <S extends T> S save(S entity);
    <S extends T> Iterable<S> saveAll(Iterable<S> entities);

    Optional<T> findById(ID id);
    boolean existsById(ID id);
    Iterable<T> findAll();
    Iterable<T> findAllById(Iterable<ID> ids);
    long count();

    void deleteById(ID id);
    void delete(T entity);
    void deleteAll(Iterable<? extends T> entities);
    void deleteAll();
}
```

Persisting first movie entity – Annotation mess - Magic

```
public interface MovieRepository  
    extends CrudRepository<Movie, Long> {  
}
```

Automatically generates a set of CRUD methods for the `Movie` class with the `Id` attribute type `Long`

You can inject a `MovieRepository` in your `MovieController` and save the movie.

```
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Entity  
public class Movie {  
    @Id  
    private Long id;  
    @NotNull(message = "Title must be set")  
    @NotEmpty(message = "Title not there")  
    private String title;  
    private boolean wonOscar;  
    @Min(value = 1920, message = "...")  
    @Max(value = 2022, message = "...!")  
    private int releaseYear;  
    @NotNull  
    @Pattern(regexp = "(https:\\\\\\\\/).*\\.(?:jpg|gif|png)",  
        message = "Must be a valid URL to a picture.")  
    private String coverImage;
```

Lombok

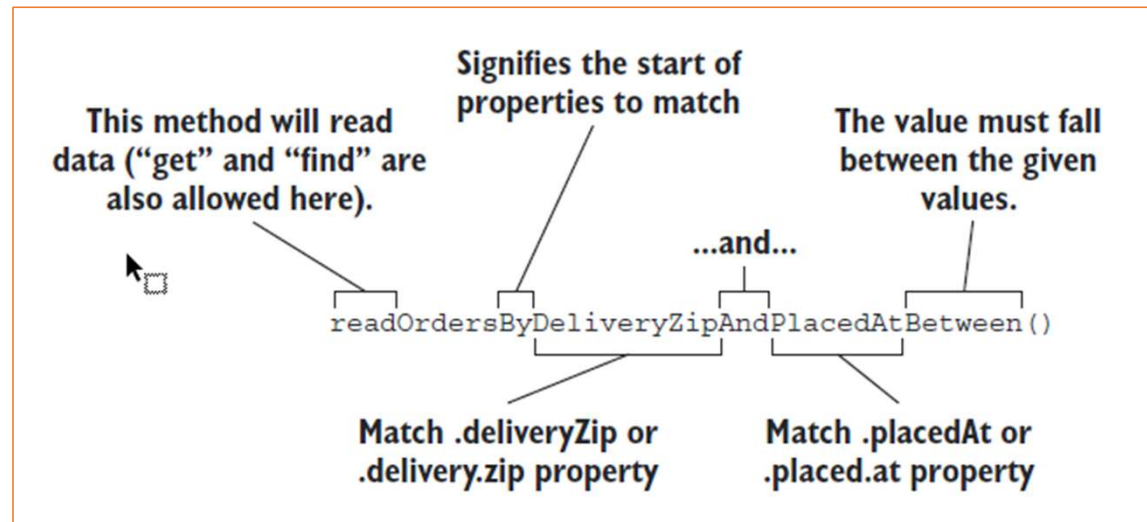
JPA Annotation marking the class as a JPA Entity

JPA Id Annotation (id must be unique, not null etc.)

Bean validation, used within presentation layer (already seen) and database layer

CrudRepository – Add additional methods

- Spring offers you an easy way to specify additional queries
- It is like prose, you will understand it ☺



- `DeliveryZip` here is appended by `Equals` operation implicitly, but you can also as in the `Between` operation choose from a set of operations: `IsAfter`, `After`, `IsGreaterThan`....

Figure taken from Craig Walls: Spring in Action, fifth edition, 2019, Figure 3.2, page 82.

Query keywords: <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

Persisting first movie entity – Controller

```
@Slf4j
@Controller
@RequestMapping(value = "/movies")
public class MovieController {

    private final MovieRepository movieRepository;

    @Autowired
    public MovieController( MovieRepository movieRepository) {
        this.movieRepository = movieRepository;
    }

    @GetMapping
    public String getMovies(Model model) { . . .

        model.addAttribute("movies", this.movieRepository.findAll());
        . . .
    }

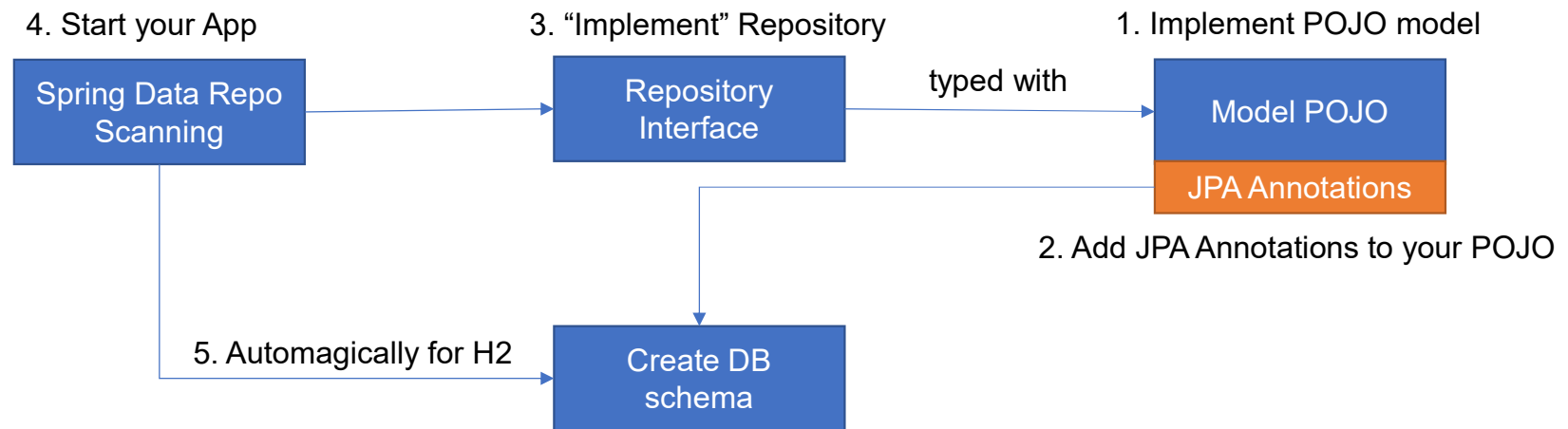
    @PostMapping
    public String processMovie(@Valid Movie movie . . .) {

        this.movieRepository.save(movie);
        . . .
    }
}
```

Dependency Injection of our
repository interface

Method implementation
automatically generated

What's happening in the background



- Runtime generates beans for the repositories
- Schema generation and update of the schema can be configured via properties

What is JPA?

- Persistency specification to enable mapping of objects into databases
- JPA (formerly Java Persistence API – now Jakarta Persistence API)
- JPA is a Jakarta EE application programming interface specification
- Current version is 2.2 (JSR 338)
New features: Stream queries, support for Java 8 Date Time API
- Spring Boot supports JPA via a starter, does a lot of configuration stuff automatically
- POJOs (entities in JPA) can be used in the persistence, service & presentation layer

Might not always
be a good idea...

https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf

JPA - Entity

- Lightweight persistence domain object
 - Annotate class with `@Entity`
 - !! No-args constructor (very important)
 - Do not declare class or fields as final
 - Must implement the `Serializable` interface (in Spring Boot not necessary)
 - Getters and Setters for fields (default naming convention, field name needs `setName()` and `getName()` methods)
 - Fields annotated with `@Transient` are not stored in the database
 - Use Bean Validation also at the database level

```
@Data
@NoArgsConstructor
@Entity
public class Movie {
}
```

→ Lombok: Generates Setters, Getters, ...

→ Lombok: No Args Constructor generation

→ JPA Annotation marking the class as a JPA Entity

JPA - Expressing relations

- `@OneToOne`: Referencing another object
- `@OneToMany`: “One entity instance can be related to many instances of the other entities”
- `@ManyToOne`: “Many instances of an entity can be related to a single entity of the other entity”.
- `@ManyToMany`: N-M Relation

JPA – Direction in Relationships

- Unidirectional: Only one entity of a relation has one of the annotations and therefore can access the associated entity
- Bidirectional: Each entity has a relationship field and refers to the mapped entity
 - “The inverse side of a bidirectional relationship must refer to its owning side by using the `mappedBy` element of the `@OneToOne`, `@OneToMany`, or `@ManyToMany` annotation. The `mappedBy` element designates the property or field in the entity that is the owner of the relationship.”
 - “The many side of many-to-one bidirectional relationships must not define the `mappedBy` element. The many side is always the owning side of the relationship.”
 - “For one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.”
 - “For many-to-many bidirectional relationships, either side may be the owning side.”

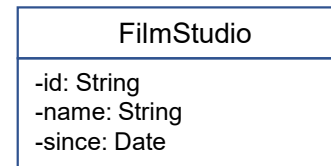
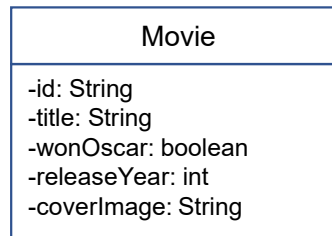
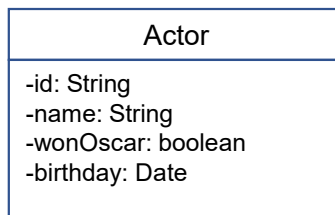
JPA - Database Evolution

```
@Entity
public class Actor {
    @Id
    private Long id;
    private String name;
    private boolean wonOscar;
    private LocalDate birthday;
}
```

```
@Entity
public class FilmStudio {
    @Id
    private Long id;
    private String name;
    private Date since;
}
```

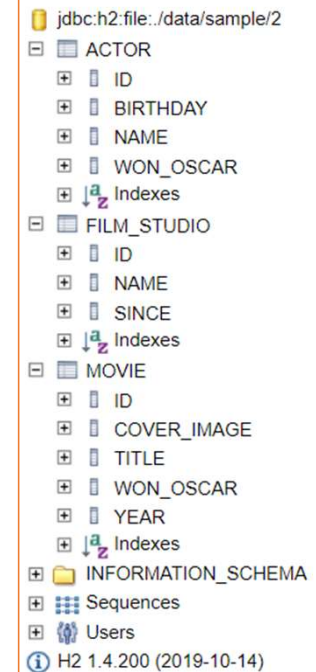
```
@Entity
public class Movie {
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Long id;
    private String title;
    private boolean wonOscar;
    private int releaseYear;
    private String coverImage;
}
```

UML



Removed all Lombok (@Data, @NoArgsConstructor) annotation. Removed all jakarta.validation annotations.

Database schema:



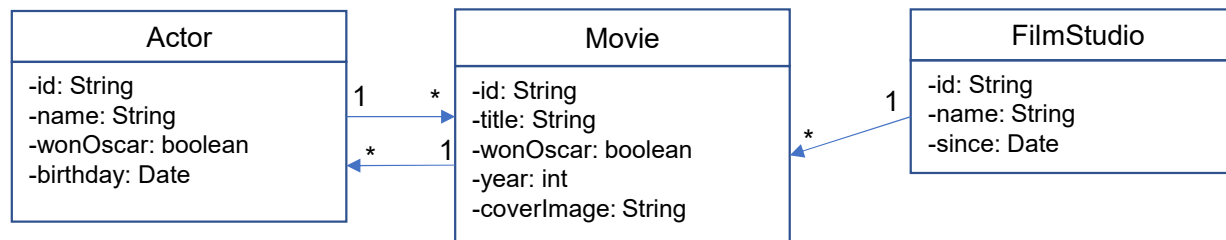
JPA - Database Evolution

```
@Data
@Entity
public class Actor {
    @Id
    private Long id;
    ...
    @ManyToMany
    private List<Movie> movies;
}
```

```
@Data
@Entity
public class FilmStudio {
    @Id
    private Long id;
    ...
    @OneToMany
    private List<Movie> movies;
}
```

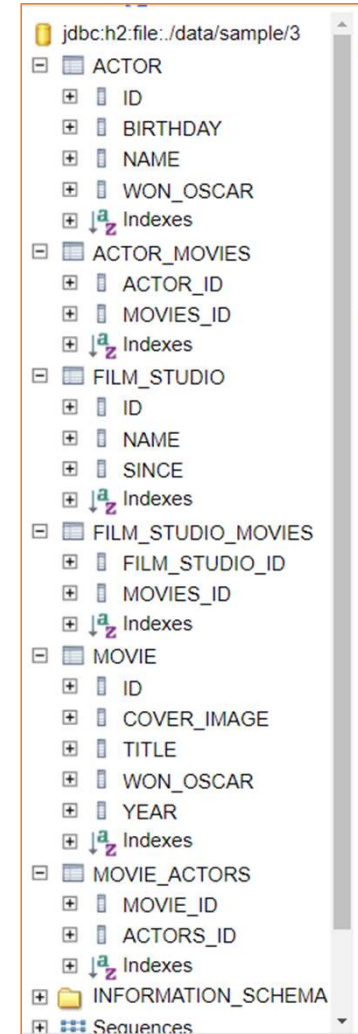
```
@Entity
public class Movie {
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Long id;
    ...
    @ManyToMany
    private List<Actor> actors;
}
```

UML



Removed all Lombok (@Data, @NoArgsConstructor) annotation. Removed all jakarta.validation annotations.

Database schema:



Join table

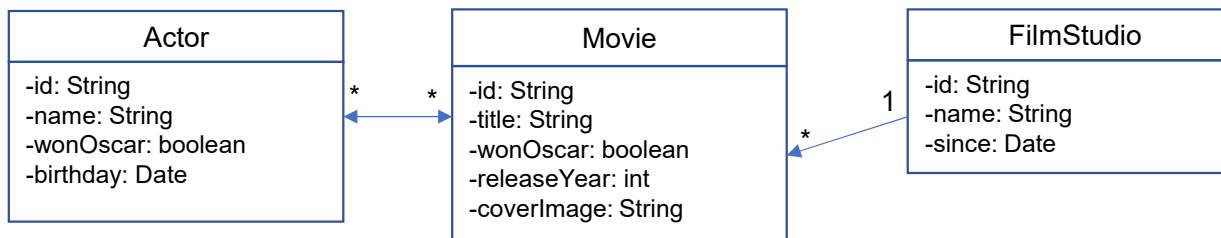
JPA - Database Evolution

```
@Data
@Entity
public class Actor {
    @Id
    private Long id;
    ...
    @ManyToMany(mappedBy = "actors")
    private List<Movie> movies;
}
```

```
@Data
@Entity
public class FilmStudio {
    @Id
    private Long id;
    ...
    @OneToMany
    private List<Movie> movies;
}
```

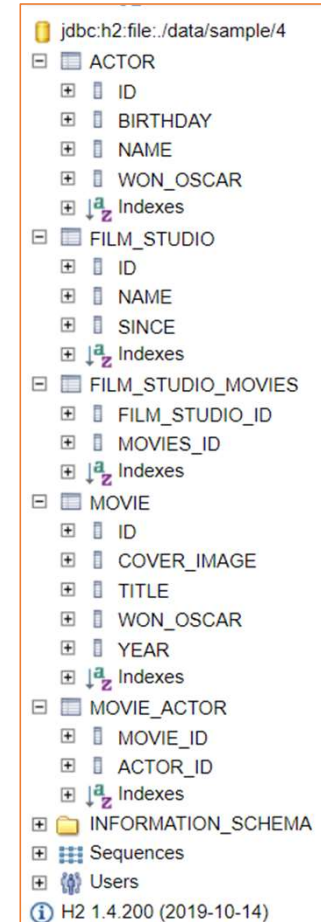
```
@Entity
public class Movie {
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    private Long id;
    ...
    @ManyToMany
    @JoinTable(name="movie_actor",
        joinColumns=@JoinColumn(name="movie_id"),
        inverseJoinColumns = @JoinColumn(name="actor_id"))
    private List<Actor> actors;
}
```

UML



Removed all Lombok (@Data, @NoArgsConstructor) annotation. Removed all jakarta.validation annotations.
Hint: For N:M mappings, you normally specify a mapping table and resolve its via two @OneToMany relations in Movie and Actor

Database schema:



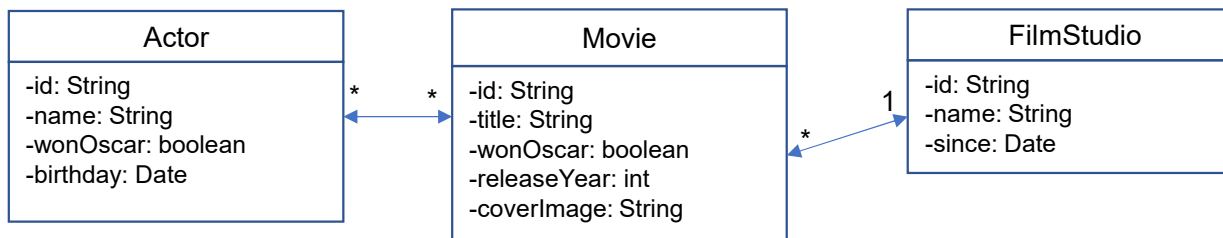
JPA - Database Evolution

```
@Data
@Entity
public class FilmStudio {
    @Id
    private Long id;
    ...
    @OneToMany(mappedBy = "filmStudio")

    private List<Movie> movies;
}
```

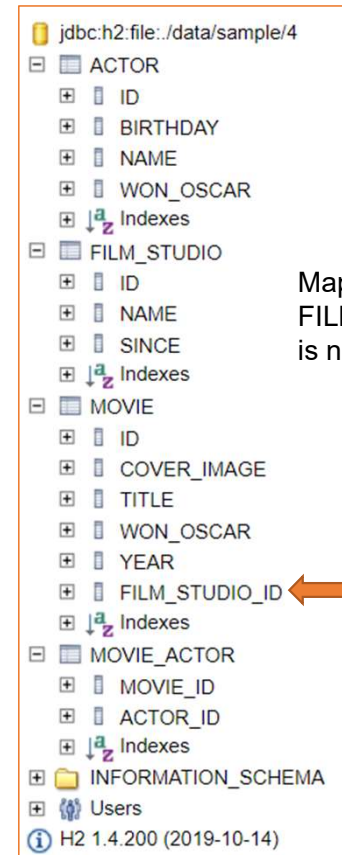
```
@Entity
public class Movie {
    ...
    @ManyToMany
    @JoinTable(name="movie_actor",
        joinColumns=@JoinColumn(name="movie_id"),
        inverseJoinColumns = @JoinColumn(name="actor_id"))
    private List<Actor> actors;
    @ManyToOne
    private FilmStudio filmStudio;
}
```

UML



Removed all Lombok (@Data, @NoArgsConstructor) annotation. Removed all jakarta.validation annotations.
Hint: For N:M mappings, you normally specify a mapping table and resolve its via two @OneToMany relations in Movie and Actor

Database schema:



Mapping table
FILM_STUDIO_MOVIES
is not present any more

LazyInitialization Exception

```
private void printMovie(long l) {
    Optional<Movie> m = this.movieRepo.findById(l);
    if(m.isPresent()){
        Movie movie = m.get();
        log.info("Print movie: " + movie.getTitle());
        for(Actor a : movie.getActors()){
            log.info("\t famous actor: " + a.getName() );
        }
    }
}
```

Currently we do not know how this movie repo stuff works, but we assume, that it will read data from our SQL database

LazyInitializationException, but why?



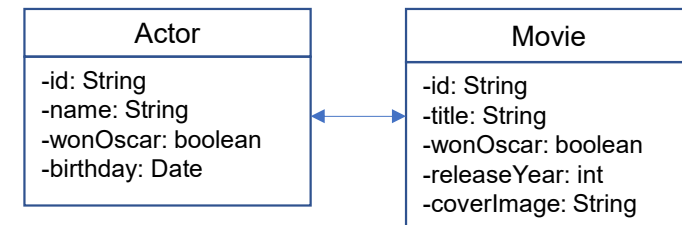
Error message: failed to lazily initialize a collection of role:
de.lion5.spring.dvd.model.Movie.actors, could not initialize proxy - no Session

- Actor and Movie are stored in different tables
- To avoid loading deep dependency trees into memory, the default behavior is to load collections lazily (only when needed)
- But when not specified and session context is not present, LazyInitializationException is thrown
- You can also specify to load the collection eagerly (loaded every time a movie is loaded from the database – only for this collection (not transitively))

https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf, Check chapter 3.7

<https://www.baeldung.com/hibernate-lazy-eager-loading>

The code present here is included in samples/dvd-jpa



Eager Loading

```
public class Movie {  
    ...  
    @ManyToMany(fetch = FetchType.EAGER)  
    @JoinTable(name="movie_actor",  
        joinColumns=@JoinColumn(name="movie_id"),  
        inverseJoinColumns = @JoinColumn(name="actor_id"))  
    private List<Actor> actors;  
}
```

Always fetches the collection, also for use cases, where you do not need it. Results in performance issues!

- Advantage: No delayed initialization at runtime (-> no performance problem here)
- Advantage: Easy to use (for the lazybones)
- Disadvantage: Long initialization times at startup
- Disadvantage: Loading a lot of unnecessary data -> performance at runtime
DON'T DO THIS

<https://www.baeldung.com/hibernate-lazy-eager-loading>

Lazy Loading

```
public class Movie {  
    ...  
    @ManyToMany(fetch = FetchType.LAZY)  
    @JoinTable(name="movie_actor",  
        joinColumns=@JoinColumn(name="movie_id"),  
        inverseJoinColumns = @JoinColumn(name="actor_id"))  
    private List<Actor> actors;  
}
```

Only fetches the data in memory,
when the data is really needed.

- Advantage: Loading of data in general much faster
- Advantage: Less memory consumption
- Disadvantage: Reload data might impact performance at crucial points in your app
- Disadvantage: Carefully think about your access paths and the additional configuration
- Disadvantage: If you are not in the same session, you get a lazy initialization exception

Consequence: You have to additionally configure LazyLoading
But How? 🤖

<https://www.baeldung.com/hibernate-lazy-eager-loading>

Lazy Loading

- EAGER (default for @ManyToOne and @OneToOne) and LAZY (default for @OneToMany, @ManyToMany) are static properties and you cannot switch between them at runtime
- Do NOT use EAGER in production scenarios (or only seldom when other approaches are not practicable)

How to configure LAZY loading with EntityGraphs (since JPA 2.1)

- Entity graphs are a set of annotations which specify (normally) lazy fetched associations to be loaded in the same query as the entity (LEFT OUTER JOIN)
- “Briefly put, the JPA provider loads all the graph in **one** select query and then avoids fetching association with more SELECT queries [often described as n+1 performance issue]. This is considered a good approach for improving application performance.”
- EntityGraphs “may be used to specify the path and boundaries for `find` operations or queries” – load collections eagerly at runtime.
- “Entity graph names must be unique within the persistence unit”
- “The attributeNodes element lists attributes of the annotated entity class that are to be included in the entity graph. “

Baeldung,
see refs

(JPA 2.2 doc,
page 413)

https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf, Check chapter 3.7

https://download.oracle.com/otn-pub/jcp/persistence-2_2-mrel-spec/JavaPersistence.pdf, Check chapter 10.3

<https://www.baeldung.com/jpa-entity-graph>

<https://www.baeldung.com/spring-data-jpa-named-entity-graphs>

Check further explanation in DemoData.java of our dvd microservice

NamedEntityGraph & EntityGraph

```
@Entity
@NamedEntityGraph(name = "Movie.movies",
    attributeNodes = @NamedAttributeNode(value="actors"))
public class Movie {
    ...
    @ManyToMany(fetch = FetchType.LAZY)
    @JoinTable(name="movie_actor",
        joinColumns=@JoinColumn(name="movie_id"),
        inverseJoinColumns = @JoinColumn(name="actor_id"))
    private List<Actor> actors;
    @ManyToOne(cascade = CascadeType.MERGE) // default fetch type: EAGER
    private FilmStudio filmStudio;
}
```

actors member is eagerly loaded, when "Movie.movies" entity graph is specified for the query

filmStudio is not included in the entity graph and is therefore loaded in a separate query (hibernate uses interceptor objects which cause problems when using Jackson – look in your debug console)

```
public interface MovieRepository extends CrudRepository<Movie, Long> {

    @Override
    @EntityGraph(value="Movie.movies") // entity graph solution
    Optional<Movie> findById(Long aLong);

    @Override
    @EntityGraph(value="Movie.movies") // entity graph solution
    Iterable<Movie> findAll();
}
```

Has to be defined for each query/find method. Otherwise defaults are used (in this case resulting in a LazyInitializationException)

Check further explanation in DemoData.java of our dvd microservice

Consequences of Eager vs. NamedEntityGraph

- The following two slides show the differences in loading an actor from the database in case of eager loading and named entity graphs.
- In the NamedEntityGraph scenario you will see the left outer joins (so only a single query is needed and only left outer entries are considered!!)
- In the EAGER scenario, you will see a lot of sql statements for the related entities which are loaded one by one, resulting in n+1 queries
- You can check the executed SQL statements when changing the sql logging level, the pretty print setting as well as the binder, which prints the ids used for the queries on the console.

```
logging.level.org.hibernate.SQL=DEBUG  
spring.jpa.properties.hibernate.format_sql=true  
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

SQL Statements when fetching Actor #3 Named Entity Graph

```
select
  actor0_.id as id1_0_0_,
  actor0_.birthday as birthday2_0_0_,
  actor0_.name as name3_0_0_,
  actor0_.won_oscar as won_osca4_0_0_,
  movies1_.actor_id as actor_id2_3_1_,
  movie2_.id as movie_id1_3_1_,
  movie2_.id as id1_2_2_,
  movie2_.cover_image as cover_im2_2_2_,
  movie2_.film_studio_id as film_stu6_2_2_,
  movie2_.title as title3_2_2_,
  movie2_.won_oscar as won_osca4_2_2_,
  movie2_.year as year5_2_2_
from
  actor actor0_
left outer join
  movie_actor movies1_
    on actor0_.id=movies1_.actor_id
left outer join
  movie movie2_
    on movies1_.movie_id=movie2_.id
where
  actor0_.id=3
```

ACTOR

<u>ID</u>	<u>BIRTHDAY</u>	<u>NAME</u>	<u>WON OSCAR</u>
1	1956-07-09	Tom Hanks	TRUE
2	1974-11-11	Leonardo Di Caprio	TRUE
3	1966-08-14	Halle Berry	TRUE

MOVIE_ACTOR

<u>MOVIE_ID</u>	<u>ACTOR_ID</u>
1	2
1	3
2	1
2	3

FilmStudio

<u>ID</u>	<u>NAME</u>	<u>SINCE</u>
1	Warner Bros. Pictures	1923-04-04

MOVIE

<u>ID</u>	<u>COVER_IMAGE</u>	<u>TITLE</u>	<u>WON OSCAR</u>	<u>YEAR</u>	<u>FILM_STUDIO_ID</u>
1		Inception	FALSE	2010	1
2		Cloud Atlas	FALSE	2012	1
3		More Than Honey	FALSE	2012	null

this.actorRepo.findById(3L);

And another query for the eagerly fetched film studio

SQL Statements when fetching Actor #3 - EAGER Loading (same table content)

this.actorRepo.findById(3L);

```
select
  actor0_.id as id1_0_0_,
  actor0_.birthday as birthday2_0_0_,
  actor0_.name as name3_0_0_,
  actor0_.won_oscar as won_osca4_0_0_,
  movies1_.actor_id as actor_id2_3_1_,
  movie2_.id as movie_id1_3_1_,
  movie2_.id as id1_2_2_,
  movie2_.cover_image as cover_im2_2_2_,
  movie2_.created_by_id as created_6_2_2_,
  movie2_.film_studio_id as film_stu7_2_2_,
  movie2_.title as title3_2_2_,
  movie2_.won_oscar as won_osca4_2_2_,
  movie2_.year as year5_2_2_,
  user3_.id as id1_4_3_,
  user3_.full_name as full_nam2_4_3_,
  user3_.password as password3_4_3_,
  user3_.phone_number as phone_nu4_4_3_,
  user3_.role as role5_4_3_,
  user3_.username as username6_4_3_,
  filmstudio4_.id as id1_1_4_,
  filmstudio4_.name as name2_1_4_,
  filmstudio4_.since as since3_1_4_
from
  actor actor0_
left outer join
  movie_actor movies1_
    on actor0_.id=movies1_.actor_id
left outer join
  movie movie2_
    on movies1_.movie_id=movie2_.id
left outer join
  user user3_
    on movie2_.created_by_id=user3_.id
left outer join
  film_studio filmstudio4_
    on movie2_.film_studio_id=filmstudio4_.id
where
  actor0_.id=3
```

```
select
  actors0_.movie_id as movie_id1_3_0_,
  actors0_.actor_id as actor_id2_3_0_,
  actor1_.id as id1_0_1_,
  actor1_.birthday as birthday2_0_1_,
  actor1_.name as name3_0_1_,
  actor1_.won_oscar as won_osca4_0_1_
from
  movie_actor actors0_
inner join
  actor actor1_
    on actors0_.actor_id=actor1_.id
where
  actors0_.movie_id=2
```

```
select
  movies0_.actor_id as actor_id2_3_0_,
  movies0_.movie_id as movie_id1_3_0_,
  movie1_.id as id1_2_1_,
  movie1_.cover_image as cover_im2_2_1_,
  movie1_.created_by_id as created_6_2_1_,
  movie1_.film_studio_id as film_stu7_2_1_,
  movie1_.title as title3_2_1_,
  movie1_.won_oscar as won_osca4_2_1_,
  movie1_.year as year5_2_1_,
  user2_.id as id1_4_2_,
  user2_.full_name as full_nam2_4_2_,
  user2_.password as password3_4_2_,
  user2_.phone_number as phone_nu4_4_2_,
  user2_.role as role5_4_2_,
  user2_.username as username6_4_2_,
  filmstudio3_.id as id1_1_3_,
  filmstudio3_.name as name2_1_3_,
  filmstudio3_.since as since3_1_3_
from
  movie_actor movies0_
inner join
  movie movie1_
    on movies0_.movie_id=movie1_.id
left outer join
  user user2_
    on movie1_.created_by_id=user2_.id
left outer join
  film_studio filmstudio3_
    on movie1_.film_studio_id=filmstudio3_.id
where
  movies0_.actor_id=1
```

```
select
  actors0_.movie_id as movie_id1_3_0_,
  actors0_.actor_id as actor_id2_3_0_,
  actor1_.id as id1_0_1_,
  actor1_.birthday as birthday2_0_1_,
  actor1_.name as name3_0_1_,
  actor1_.won_oscar as won_osca4_0_1_
from
  movie_actor actors0_
inner join
  actor actor1_
    on actors0_.actor_id=actor1_.id
where
  actors0_.movie_id=1
```

```
select
  movies0_.actor_id as actor_id2_3_0_,
  movies0_.movie_id as movie_id1_3_0_,
  movie1_.id as id1_2_1_,
  movie1_.cover_image as cover_im2_2_1_,
  movie1_.created_by_id as created_6_2_1_,
  movie1_.film_studio_id as film_stu7_2_1_,
  movie1_.title as title3_2_1_,
  movie1_.won_oscar as won_osca4_2_1_,
  movie1_.year as year5_2_1_,
  user2_.id as id1_4_2_,
  user2_.full_name as full_nam2_4_2_,
  user2_.password as password3_4_2_,
  user2_.phone_number as phone_nu4_4_2_,
  user2_.role as role5_4_2_,
  user2_.username as username6_4_2_,
  filmstudio3_.id as id1_1_3_,
  filmstudio3_.name as name2_1_3_,
  filmstudio3_.since as since3_1_3_
from
  movie_actor movies0_
inner join
  movie movie1_
    on movies0_.movie_id=movie1_.id
left outer join
  user user2_
    on movie1_.created_by_id=user2_.id
left outer join
  film_studio filmstudio3_
    on movie1_.film_studio_id=filmstudio3_.id
where
  movies0_.actor_id=2
```

Resulting in 5 queries (n+1 problem): At runtime the many SQL statements, when more than only two actors are playing in a movie highly stresses the db

Transactions

Transaction – JDBC – Enabling ACID

- Before starting with Spring Boot transaction handling, get an understanding of plain old Java transaction with JDBC
- Thanks to Marco Behler for this great tutorial, where the examples are copied from

```
import java.sql.Connection;

...

Connection connection = dataSource.getConnection();

try (connection) {
    connection.setAutoCommit(false);

    // execute some SQL statements...

    connection.commit();
} catch (SQLException e) {
    connection.rollback();
}
```

Connection to database to start transaction (getting one connection of the pool (limited in connections))
Very important! Starts the transaction. Setting the commit mode to false hands over control to us. True means every single SQL statement has its own transaction!

Commit all SQL statements in our transaction

Rollback in case of an error

Good explanation of Transaction basics: <https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html>
JDBC basics in Spring/ Spring Boot: <https://www.marcoehler.com/guides/spring-transaction-management-transactional-in-depth>

Important Terminology

- Transaction manager: Coordinates participating resources, handling transactions
- Transaction context: Representation of a transaction
- Connection pool: Caching reusable database connections. (limited size)
- Transaction: A set of operations acting like a single operation (ACID)
- Savepoint: Point within a transaction, where the current state of the db is saved and the state can be rolled backed to this saved state
- Rollback: Undo all the changes in the transaction and keep the database state consistent

Transactions in Spring Boot I

- Transaction manager is autoconfigured (JPA dependency)
- Low level transaction handling is replaced via `@Transactional`

```
@Transactional
public Movie update(Movie movie) {
    return this.movieRepo.save(movie);
}
```

- Transaction starts, when the first command to db is executed
- Transaction ends, when the method returns
- Enable the following logging properties to look at transactions

```
logging.level.org.hibernate.transaction=TRACE
logging.level.org.springframework.transaction=TRACE
```

- Look for `ConnectionHolder` objects in your logs

@Transactional

- If class or a method is marked as @Transactional, Spring generates a CGLib proxy
- This proxy delegates the JDBC handling to the transaction manager
- If you call multiple @Transactional methods subsequently, only a single transaction is open due to the proxy (keep this in mind – the propagation has no effect in this case)
- Propagation options to tell Spring how it handles nested method calls to @Transactional methods:
 - REQUIRED (default) open transaction or reuse existing
 - SUPPORTS “don’t really care if a transaction is open or not” [1]
 - MANDATORY don’t open transaction but throw exception if none is present
 - REQUIRES_NEW open a new transaction, independent from previous ones
 - NOT_SUPPORTED “Execute non-transactionally, suspend the current transaction if one exists” [2]
 - NEVER “Execute non-transactionally, throw an exception if a transaction exists.” [2]
 - NESTED “Execute within a nested transaction if a current transaction exists, behave like REQUIRED otherwise.” [2]
- In case of an error, a TransactionSystemException is thrown – handle it in your REST controller or frontend class

[1] <https://www.marcobehler.com/guides/spring-transaction-management-transactional-in-depth>

[2] <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.transaction.annotation/Propagation.html>

Isolation Levels

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
TRANSACTION_NONE	Not supported	<i>Not applicable</i>	<i>Not applicable</i>	<i>Not applicable</i>
TRANSACTION_READ_COMMITTED	Supported	Prevented	Allowed	Allowed
TRANSACTION_READ_UNCOMMITTED	Supported	Allowed	Allowed	Allowed
TRANSACTION_REPEATABLE_READ	Supported	Prevented	Prevented	Allowed
TRANSACTION_SERIALIZABLE	Supported	Prevented	Prevented	Prevented

- “A dirty read [...] is reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a *dirty read* because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)”
- “A *non-repeatable read* occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.”
- “A *phantom read* occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an additional row. This row is referred to as a phantom.”

<https://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html>

@Transactional Example

```
@Transactional(propagation = Propagation.REQUIRED,  
    isolation = Isolation.READ_COMMITTED)  
public Movie saveAndSetId(Movie movie) {  
    log.info("Start save");  
  
    // First action to the db starts the transaction  
    movie.setId(this.movieRepo.count() + 1L);  
    Movie storedMovie = this.movieRepo.save(movie);  
    log.info(storedMovie.toString());  
  
    // do other processing in the transaction context  
  
    // Return will end the transaction  
    return storedMovie;  
}
```

Propagation and isolation level for the transaction to store a movie.

The first action to the db starts the transaction and keeps it open/active.

Do not call long running processes here – the number of connections are limited!!

Our examples under microservices/dvd contain two examples of how transactional annotations work.

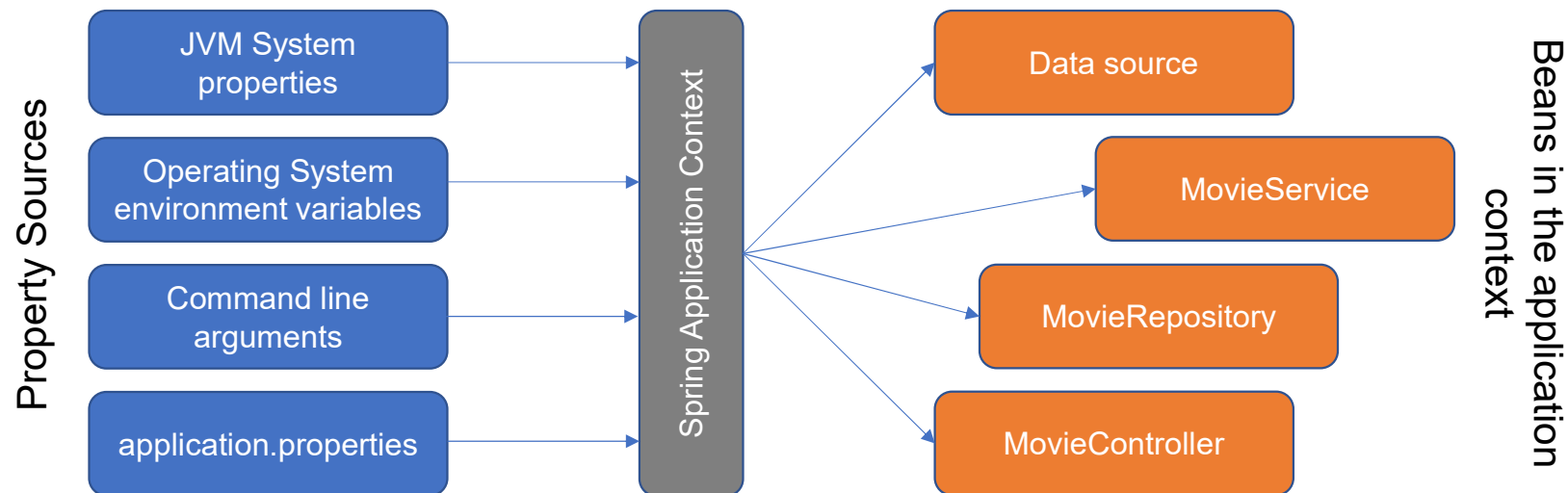
ActorService#delete (@Transactional) deletes an actor, removes this actor from all the actor's movies and updates (MovieService#update - @Transactional(propagation = Propagation.REQUIRES_NEW) each movie. For each movie update, a new connection is used (look at the connection handler hashes in your logs).

The second example is artificially created. In MovieService#saveAndSetId another call is made to MovieService#findById. Here the propagation in findById is also REQUIRES_NEW, but there is only a single connection holder used – why? As already said, Spring proxies these services by a custom transaction handler which delegates the transaction work to the transaction manager. By default the findById will not open another connection since both transactions are initiated from the same object and therefore the same connection is used.

Properties & Profiles

part III

Property Sources and the Application Context



- Properties are read from different sources, with different priorities (overriding is possible)
- Spring combines these properties and customizes their beans in the application context
- Highly flexible way for different environments

Figure adapted from Craig Walls: Spring in Action, fifth edition, 2019, Figure 5.1, page 116.

Important configuration options

- `server.port` specifies the port on which the application is started. `server.port=0` means that the server is started on a random port (good for integration testing etc.)
- Enabling SSL via configuration parameters
- Specify individual logging levels for different packages.
Property starts with `logging.level.PACKAGE=DEBUG`

Bootstrap vs. Application properties

- Properties in `bootstrap.properties` are used during the bootstrap phase (starting) of the application
- “By default, bootstrap properties [...] are added with high precedence, so they cannot be overridden by local configuration.”
- A good example is the Spring Cloud Config’s server URI

All configuration parameters are written down in the `application.properties` naming scheme – you can also use yaml notation.

For SSL config: <https://www.baeldung.com/spring-boot-https-self-signed-certificate>

SSL config is also included in the `microservices/dvd` project in the file `application-sec.properties` (another custom profile for SSL)

For logging: <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#features.logging>

Application Context – Bootstrap: https://cloud.spring.io/spring-cloud-commons/multi/multi_spring_cloud_context_application_context_services.html

Creating custom configuration properties

- Property injection via `@ConfigurationProperties(prefix="movies")`
- All properties starting with `movies.XY` can be injected if the attribute name is the same as the property name
- Often it makes sense to add another component (bean) to your app which acts as a property holder to keep other classes clean
- “Any `@Component`, `@Configuration` or `@ConfigurationProperties` can be marked with `@Profile` to limit when it is loaded”
- Look at the spring boot reference documentation for the order in which properties are applied
- Possibility to specify multiple profiles in a single `.yaml` file (not possible with property files – naming convention)

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config>

<https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-multi-profile-yaml>

In our example project under `microservices/dvd`, there is an example of configuration properties in `MovieProperties`

Spring Security

Spring Security at a glance

- Adding spring security starter to the project

```
implementation 'org.springframework.boot:spring-boot-starter-security'
```
- Getting a default `UserDetailsService` with a single user (name: user)
- Password is a randomly generated string in the logs
- That's it 😊 But what you get is only the start: (see ref in the footer)
 - Currently no login page
 - Single User with username “user”
 - No roles defined – see custom approaches
 - See `SpringSecurityConfig` to specify authentication for HTTP requests
- What's missing?
 - Login page
 - Registration page for new customers
 - Different rules for different paths

<https://docs.spring.io/spring-security/reference/servlet/getting-started.html>

More information in Craig Walls: Spring in Action, fifth edition, 2019, page 86 ff

Spring Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig implements WebMvcConfigurer {

    @Bean
    public PasswordEncoder createEncoder() { return new BCryptPasswordEncoder(); }

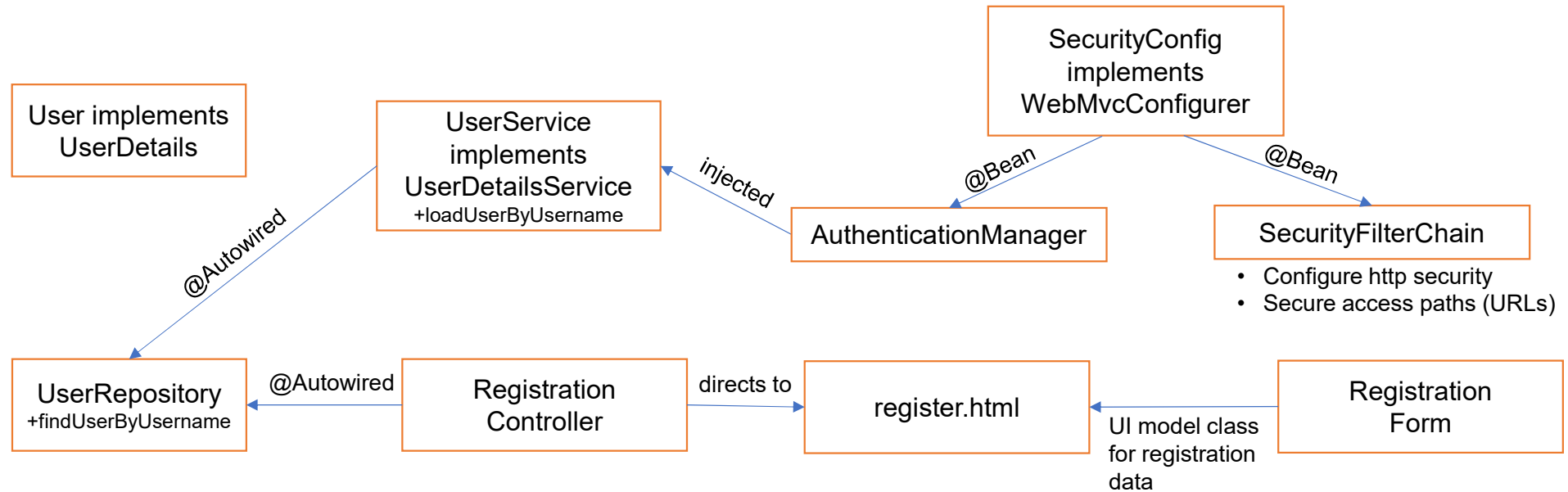
    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http,
        PasswordEncoder passwordEncoder, UserDetailsService userDetailsService) throws Exception {
        ...
        return authenticationManagerBuilder.build();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) {
        http.authorizeRequests((requests -> requests.anyRequest().authenticated()));
        return http.build();
    }
}
```

- Enable security through `@EnableWebSecurity` and configure it by adding Beans
- Options to configure user store via `AuthenticationManager`
 - In-memory user store | JDBC user store | Using external LDAP service | **Custom user details service**
- Secure specific paths using `SecurityFilterChain`

It makes sense to use incognito mode to get a new session for testing the security updates.

Custom UserDetails service - HowTo



- `@AuthenticationPrincipal` annotation is useful to inject user in post controller methods
- Add `thymeleaf-extras-security6` to your build (enables you to access user and other security related stuff in the UI)

Look at our example in the mentioned classes at `microservices/dvd`
 Good explanation can be found in Craig Walls: Spring in Action, fifth edition, 2019, chapter 4
 Thymeleaf extra security stuff: <https://www.thymeleaf.org/doc/articles/springsecurity.html>

Spring Security & Thymeleaf

- Spring adds roles (authorities) for logged in users (method in UserDetails)
- You are responsible to store these fields in your DB
- SecurityConfig secures access paths (URLs) based on roles
- Thymeleaf can – based on the rules – render or not some parts of the UI
- Most important commands (see others in the linked page)
 - `sec:authorize="isAnonymous()"`
Rendered for not logged in user
 - `sec:authorize="isAuthenticated()"`
Rendered for logged in user
 - `sec:authorize="hasRole('ROLE_ADMIN')"`
Only rendered, when user (principal) has role ADMIN
 - `sec:authentication="principal.fullName"`
Read the property fullName of the currently logged in user (you can read all attributes of you user via principal object!)

<https://www.thymeleaf.org/doc/articles/springsecurity.html>

REST

**Details of REST and how to describe APIs
are discussed in detail in DSG-SOA**

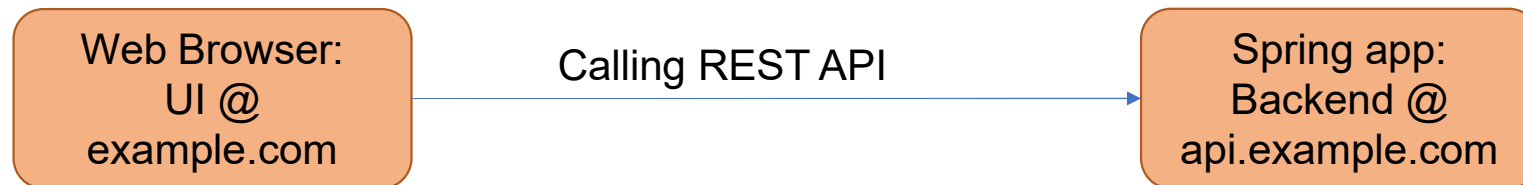
REpresentational State Transfer at a glance

- Defined by Roy Thomas Fielding in his PhD thesis
- He describes REST as an “architectural style” (It is NOT a framework or specification)
- Principles
 - Addressable resources
 - Representation-oriented manipulation of resources
 - Self-descriptive messages
 - Stateless communication
 - HATEOAS (hypermedia as the engine of application state)
- HATEOAS done correctly is like a REST API response which can be read and used like a website – navigating from one resource to others
- Most implemented REST APIs struggle by using hyperlinks in their responses to support a hypermedia driven API

Implementing first REST controller

- Use `@RestController` annotations.
It is annotated with `@Controller` and `@ResponseBody` (Component Scanning, HTTP codes)
- Specify a request mapping path and the content types, which the controller consumes and produces
- Using `ResponseEntity<T>` objects as return types
 - Builder pattern – possibility to specify different status codes
`ResponseEntity.badRequest().build()`
 - Returning custom data via constructor `new ResponseEntity<>(m.get(), OK)`
- For the first version, we use our domain classes for the REST interface

CORS – Cross Origin Resource Sharing



- REST user (web browser, JS client) is not at the same domain as the backend (spring application)
- Need CORS specified to enable communication between different domains
- Different options to allow CORS for spring applications:
 - `@CrossOrigin(origins="example.com", ...)` – Whitelisting the users, which are allowed to access the API Annotation at rest controller or method level (granular solution)
 - `cors()` method of the `HttpSecurity` object (via `WebSecurityConfig` class) – global setting

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>
<https://www.baeldung.com/spring-cors>

First REST Controller

```
@RestController
@RequestMapping(path = "/v1/movies",
    consumes = "application/json",
    produces = "application/json")

public class MovieRestController {

    // attributes & constructor

    @GetMapping
    public ResponseEntity<Iterable<Movie>> getMovies(
        @RequestParam(defaultValue = "0") int page) {
        return new ResponseEntity<>(movies, OK);
    }

    @GetMapping(value = "{id}")
    public ResponseEntity<Movie> findMovie(@PathVariable("id") Long id) {
        return new ResponseEntity<>(m.get(), OK);
    }

    @PostMapping
    public ResponseEntity<Movie> createMovie(
        @RequestBody @Valid MoviePostDTO movieDTO, Errors errors) {
        return new ResponseEntity<>(this.movieRepo.save(movie), CREATED);
    }
}
```

Controller's base path

Accept and Content Type header

Query Parameter annotation

Path Parameter annotation

Successful returning 200

Request Body annotation to map data to the method and validate it via @Valid and the validation rules

Successful returning 201

Exception handling

- Where to throw exceptions – there is no reachable call stack for the developer
- Propagating – throwing the exception up the call stack is no option
- `@ExceptionHandler({CUSTOM_EXCEPTION.class})`
 - Define this handler within your REST controller class
 - Handles custom exceptions (must be runtime exceptions, which are thrown within your class or in classes down the call stack)
 - Gives you the chance to return other payload objects as specified in the GET or POST mapping



```
@ExceptionHandler({RestControllerException.class})
public ResponseEntity<String> handleCustomException(
    RestControllerException ex,
    WebRequest request) {
    return new ResponseEntity<>(ex.getMessage(), ex.getStatus());
}
```

Update data

- `POST`: creating a resource, often used as an option to update data on the server, but semantically `PUT` or `PATCH` are cleaner (code 201)
- `PUT`: replacement operation of the resource (code 201)
- `PATCH`: partial update of the resource (code 200)

Delete data

- `DELETE`: Deleting the data (response code 204)

Try to understand the difference between `PUT` and `PATCH` and implement your API accordingly.

HATEOAS - Making your API discoverable

- Hypermedia as the engine of application state
- Spring Boot uses HAL flavor of hyperlinks
- Add HATEOAS starter to your build to enable hyperlink support
- Keep caution: HATEOAS 1.0 was released previously and some examples on websites are with prior classes (check the docs)
- `CollectionModel` and `EntityModel` are the two important resource classes for collections and single items
- `WebMvcLinkBuilder` is class with factory methods to assemble links

```
@GetMapping
public ResponseEntity<CollectionModel<EntityModel<Movie>>> getMovies() {
    Iterable<Movie> movies = this.movieRepo.findAll();

    CollectionModel<EntityModel<Movie>> collection = CollectionModel.wrap(movies);
    collection.add(WebMvcLinkBuilder.linkTo(WebMvcLinkBuilder
        .methodOn(MovieRestController.class).getMovies()).withRel("movies"));

    return new ResponseEntity<>(collection, OK);
}
```

Important method
for wrapping
domain data in
HATEOAS object

Relative link
design (nice for
refactoring and
updates)

https://github.com/mikekelly/hal_specification/blob/master/hal_specification.md
<https://docs.spring.io/spring-hateoas/docs/current/reference/html/#migrate-to-1.0.changes>

How to structure your app – a double edged sword

“Do I use my domain objects also for building my REST API or do I implement separate classes?”

Only domain classes

- more restrictions (object relation caveats)
- + smaller code base
- tighter coupling (evolvability)
- annotation based (lots of magic)

Domain and REST API classes

- + more flexible
- boiler plate code
- + loose coupling (evolvability)
- + explicit coding

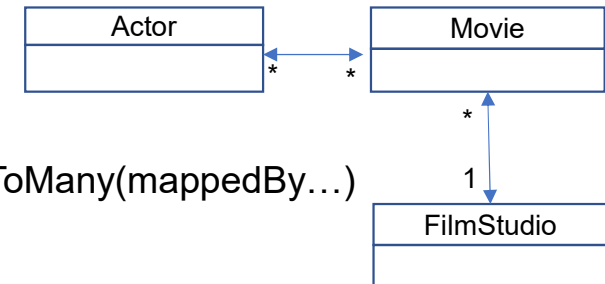
- Decision is project dependent, but in literature and on blogs is the orange way preferred
- Solves also a lot of JSON problems when implementing separate classes

<https://www.baeldung.com/entity-to-and-from-dto-for-a-java-spring-application>

Craig Walls: Spring in Action, fifth edition, 2019, NOTE on page 115.

Important JSON annotations – bidirectional relations

- Extend your domain classes with `RepresentationalModel`
- Infinite recursion when serializing data
- JPA can deal with this problem by having annotations like `@ManyToMany(mappedBy=...)`
- Jackson (JSON utility) has no automatic way to deal with it
- JSON annotations are the only way to stop the recursion properly
 - `@JsonManagedReference` (e.g., actors field in Movie, members with this annotation are serialized properly – you need a corresponding `JsonBackReference`)
 - `@JsonBackReference` (e.g., movies field in Actor, members with this annotations are not serialized – you need a corresponding `JsonManagedReference`, recursion will end here for JSON)
 - `@JsonIgnore` (as the name implies, members with this annotation are ignored during serialization and deserialization process)
 - `@JsonIgnoreProperties` (exclude only properties from JSON serialization process)
- Other useful JSON annotations
 - `@JsonView` (defining different views, where different data is serialized)
 - `@JsonIdentityInfo` (reduces the payload sent by the server)



<https://fasterxml.github.io/jackson-annotations/javadoc/2.5/com/fasterxml/jackson/annotation/JsonManagedReference.html>
<https://fasterxml.github.io/jackson-annotations/javadoc/2.5/com/fasterxml/jackson/annotation/JsonBackReference.html>
<https://fasterxml.github.io/jackson-annotations/javadoc/2.5/com/fasterxml/jackson/annotation/JsonIgnore.html>
<https://www.baeldung.com/jackson-json-view-annotation>
<https://www.baeldung.com/jackson-bidirectional-relationships-and-infinite-recursion>

How to get links to your domain objects?

```
public class FilmStudio extends RepresentationModel<FilmStudio> {  
    // already defined  
}
```

RepresentationModel is a hateoas class, marking this class as a potential Rest model

```
public class FilmStudioAssembler extends  
RepresentationModelAssemblerSupport<FilmStudio, FilmStudio> {  
  
    public FilmStudioAssembler() {  
        super(FilmStudioRestController.class, FilmStudio.class);  
    }  
  
    @Override  
    protected FilmStudio instantiateModel(FilmStudio entity) {  
        return entity;  
    }  
  
    @Override  
    public FilmStudio toModel(FilmStudio entity) {  
        return this.createModelWithId(entity.getId(), entity);  
    }  
}
```

Converter to convert the domain object into the RepresentationModel (in this case both are the same)

Base path and model class

Not necessary to override it, but highly recommended

Actual conversion
Create model with id is a nice helper here to get the correct hyperlink

This example is included in our demo project for getting all film studios. The other implemented REST endpoints are all implemented with separate REST interface objects.

HATEOAS Pitfalls - Jackson & Hibernate

- Getting weird (recursive output on screen)
Solution: Using `JsonManagedReference`, `JsonBackReference`, `JsonIgnoreProperties` or `JsonIgnore` annotations
- Getting `No serializer found for class ... exception message`
Solution: Adapt your named entity graph – the problem is that hibernate uses an interceptor class and loads entities when they are needed, but Jackson can't serialize these interceptors (do not use `@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})`)
This introduces performance issues (additional fields and additional queries – check the console's sql statements)

Pitfalls – Jackson & Hibernate – different solutions

JsonIgnoreProperties

```

Variables
+ ▶ this = (MovieRestController@10498)
  ▶ page = 0
  ▶ movies = (Collections$UnmodifiableRandomAccessList@10503) size = 2
    ▶ 0 = (Movie@10553) ""
      ▶ id = (Long@10556) 1
      ▶ title = "Inception"
      ▶ wonOscar = false
      ▶ year = 2010
      ▶ coverImage = "https://cdn.pixabay.com/photo/2017/05/15/17/43/calm-2315559_960_720.jpg"
      ▶ actors = (PersistentBag@10559) size = 2
      ▶ filmStudio = (FilmStudio$HibernateProxy$ajGcKV3@10560) ""
        ▶ $$hibernate_interceptor = (ByteBuddyInterceptor@10561)
          ▶ id = null
          ▶ name = null
          ▶ since = null
          ▶ movies = null
          ▶ createdBy = null
        ▶ 1 = (Movie@10554) ""
      ▶ movieRepo = ($Proxy74@10504) "org.springframework.data.jpa.repository.support.SimpleJpaRepository@6488209b"
      ▶ props = (MovieProperties@10505) "MovieProperties(pageSize=2)"

```

NamedEntityGraph

```

Variables
+ ▶ this = (MovieRestController@10364)
  ▶ page = 0
  ▶ movies = (Collections$UnmodifiableRandomAccessList@10365) size = 2
    ▶ 0 = (Movie@10419) ""
      ▶ id = (Long@10422) 1
      ▶ title = "Inception"
      ▶ wonOscar = false
      ▶ year = 2010
      ▶ coverImage = "https://cdn.pixabay.com/photo/2017/05/15/17/43/calm-2315559_960_720.jpg"
      ▶ actors = (PersistentBag@10425) size = 2
      ▶ filmStudio = (FilmStudio@10426) ""
        ▶ id = (Long@10422) 1
        ▶ name = "Warner Bros. Pictures"
        ▶ since = (LocalDate@10428) "1923-04-04"
        ▶ movies = (PersistentBag@10429) size = 2
        ▶ createdBy = null
        ▶ 1 = (Movie@10420) ""
      ▶ movieRepo = ($Proxy74@10366) "org.springframework.data.jpa.repository.support.SimpleJpaRepository@43d5fae1"
      ▶ props = (MovieProperties@10367) "MovieProperties(pageSize=2)"

```

- The left solution performs two queries, one for movies and another (when the film studio is needed) for film studios
- The right solution performs only a single query

Check our possible implementation under microservices/dvd – there we used the named entity graph solution (more performant in normal situations)

How to get links to your domain objects?

```
public class MovieModel extends RepresentationModel<MovieModel> {  
  
    @Getter private final String title;  
    @Getter private final boolean wonOscar;  
    @Getter private final int year;  
    @Getter private final String coverImage;  
    @Getter private final List<ShortActorModel> actors;  
    @Getter private final ShortFilmStudioModel filmStudio;  
    @Getter private final String username;  
  
    public MovieRepresentationalModel(Movie movie) {  
        // initializing all fields  
    }  
}
```

RepresentationModel is a hateoas class, marking this class as a potential Rest model

Copy the members from the movie class you want to expose to your customer (!!)

Short versions of your model classes with a subset of attributes (the hyperlink is included for navigating to the full info version). Keeps your responses small.

```
public class MovieAssembler extends  
    RepresentationModelAssemblerSupport<Movie, MovieModel> {  
  
    //same methods and implementation as in film studio example  
}
```

Converter to convert the domain object (FilmStudio) into the newly created RepresentationalModel

This example is included in our demo project `microservices/dvd`.
The "Domain and REST API class" approach is also used for all other endpoints despite the one we've already seen.

Full and Short Representations

Situation: I want to get a list of movies

Result: I get a list of movies, but also get all actor information and its related classes

Solution: Defining also short representations where needed

- Short representations keep the response small
- Include only the most important information in the context (e.g. the name of the actor and the link to his or her full info)
- Disadvantage is the additional LOC and classes

Request: <http://localhost:8080/v1/movies>

```
{
  _embedded: {
    movieRepresentationalModelList: [
      {
        title: "Inception",
        wonOscar: false,
        year: 2010,
        actors: [
          {
            name: "Leonardo Di Caprio",
            _links: {
              self: {
                href: "http://localhost:8080/v1/actors/2"
              }
            }
          },
          { further movies }
        ],
        filmStudio: {
          name: "Warner Bros. Pictures",
          _links: {
            self: {
              href: "http://localhost:8080/v1/studios/1"
            }
          },
          username: null,
          _links: {
            self: {
              href: "http://localhost:8080/v1/movies/1"
            }
          },
          { further movies }
        ]
      },
      { further movies }
    ],
    _links: {
      movies: {
        href: "http://localhost:8080/v1/movies?page=0"
      }
    }
  }
}
```

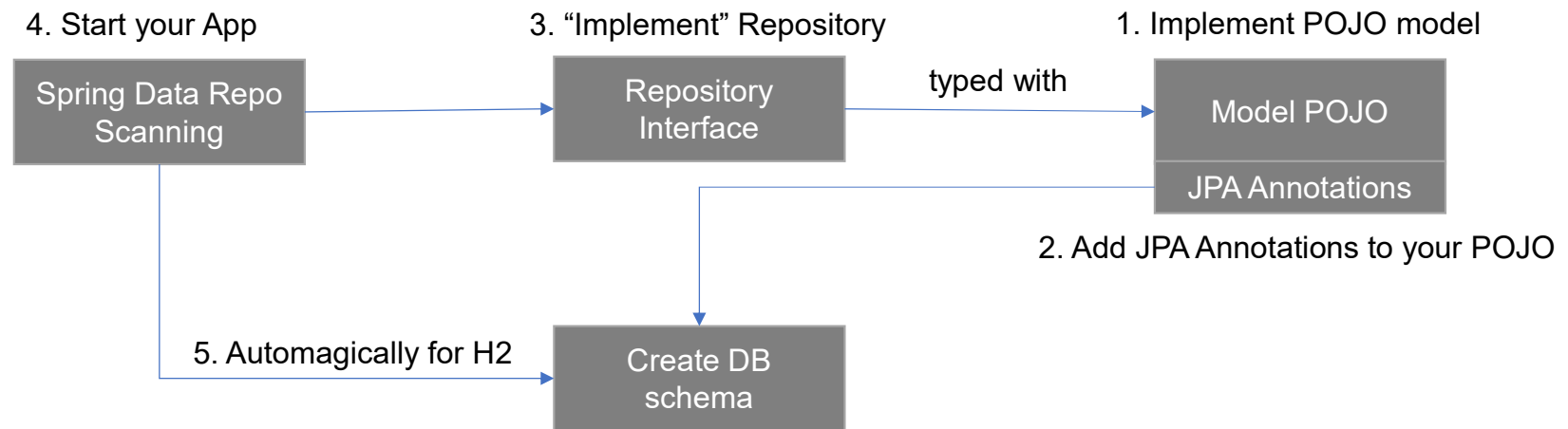
You can adjust the root element name by another annotation at your representational model class

```
@Relation(value = "movie",
collectionRelation = "movies")
```

Short versions of an actor, but you can follow the link to get more information.

Self-link specifying the request

Spring Data REST – mechanics in the background



Add Spring Data REST starter to your project and Spring automatically generates a full-blown REST API out of your JPA annotated classes.

HINT: Consider the same points as on the blue marked slides with the the "Only domain classes" banner
Futher infos at <https://docs.spring.io/spring-data/rest/docs/4.1.5/reference/html/>

To the cloud . . .

**Docker and Kubernetes are discussed in great detail in DSG-SOA,
but as with REST it is important to get an idea how to build a container to deploy it 😊**

Deployment: Plan zero – source code deployment

- Some platform offer version control integration, e.g. Heroku
- Implement your app and bind the repository to the platform
- Define a script/trigger to start the build via a build tool and deploy application

<https://blog.heroku.com/six-strategies-deploy-to-heroku>

Deployment: First choice containers

- A container is an instance of an image
 - An image is a stack of layers, where only the top layer is writable (helps in organizing images efficiently and running container in performant way)
 - Dockerfiles are the skeleton of an image and makes image creation reproducible
 - To build your solution you need a JDK in Java (but JDK is huge in size – influencing the startup and runtime behavior of your app)
 - Using only a JRE is sufficient to run your app in production
-
- Docker multi-staged builds to the rescue
 - first stage building the jar (JDK)
 - second stage copying the generated jar and run it (JRE)
 - Only the last stage is included in the image
 - Reduces the size from roughly 1GB to 300MB

Docker – Building a multi-staged image

```
1  # base image - builder stage
2  FROM openjdk:11.0.7-jdk AS builder
3  # Environment Variable
4  ENV APP_HOME=/root/dev/beverage
5  # Working directory
6  WORKDIR $APP_HOME
7  # Copy all the stuff (easiest way)
8  COPY . $APP_HOME
9  # Run the build
10 RUN ./gradlew build
11
12 # base image for the final image (java runtime environment is sufficient)
13 FROM openjdk:11.0.7-jre
14 # specifying work directory
15 WORKDIR /root/
16 # only copy the fat jar, which includes all dependencies (only a java runtime environment is needed to run it)
17 COPY --from=builder /root/dev/beverage/build/libs/beverage-all.jar .
18 # Run it
19 CMD ["java", "-jar", "beverage-all.jar"]
```

- Builder stage (not included in the image – only the last stage is included – beginning at last FROM statement)
- “Image stage” – All commands here result in a single layer
- Access to the builder stage and copying of the relevant file

Second choice: Jar Deployment

- Build a fat Jar (use the bootJar gradle command)
- Select your platform of choice, e.g. Heroku, CloudFoundry
- Install the CLIs and read the docs
- Deploy the jar or the repository with build file
- Enjoy your app in the cloud
- Example: PWS Pivotal Web Services
 - Install CLI
 - Log In
 - Reduce source compatibility to 8, then: `$ gradlew bootJar`
 - Execute: `$ cf push DVD-SERVICE -p PATH-TO-JAR`
 - Use Route to access it 😊

The screenshot shows the Pivotal Web Services (PWS) dashboard. The left sidebar contains navigation links: Space, App (1), Services, Member (1), Settings, Networking, and Routes (2). The main content area shows the 'hannes-space' with a summary of running, stopped, and down instances. Below this, a table lists the applications.

Status	Name	Instances	Memory	Last Update	Route
● RUNNING	DVD-SERVICE	1	1 GB	1 min	https://dvd-service.cfa...

Testing

Unit Testing

- Test a single class in isolation or a method of this class
- Not influenced by Spring/Spring Boot
- Via starter (spring-boot-starter-test), a lot of useful libraries are included, JUnit 5, Spring Test & Spring Boot Test, AssertJ, Mockito, Hamcrest, JSONassert, JsonPath.

Integration testing

- Spring helps with already known magic and a couple of new features to test the interaction of various components
- Spring wires the components to be tested together within an application context
- Loading application context and caching them between tests (“all tests run in the same JVM”)
- Transaction management: To not influence the persistent store, the testing “framework creates and rolls back a transaction for each test”/ each transactional method.
- Support by abstract classes of Spring’s TestContext framework

<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-testing>

Stub, Mock and Spy

- Stubs

- Additional test classes implementing business interfaces to simulate interaction between classes logically
- “Test mirror” of your business objects
- (+) You can add complex logic to the test classes and test your business objects accordingly
- (-) When logic changes, also the “test mirror” classes must be changed
- (-) dependency trees, “test mirror” class might also include other dependencies to business objects...

- Mocks

- Mocking your business objects, behavioral interaction between your classes is tested (which methods are called, how often, which input parameters are used etc.) (behavioral testing)
- (+) Easy to use, no additional classes are needed
- (-) Only interaction is tested, result within a dependent class is not assessed
- (-) By invoking methods, nothing is executed logically

- Spies

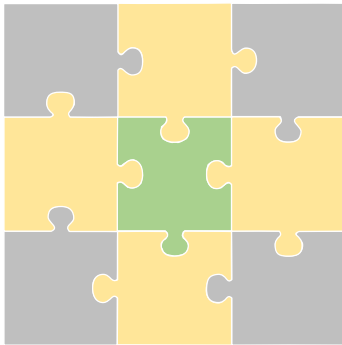
- Create a partial mock/facade of the real business object
- (+) methods which are not stubbed, will be executed from the real object
- (+) state testing of the real object is to some extent possible
- (-) real object involvement, think carefully about its dependencies

Mock vs. Spy: <https://www.baeldung.com/mockito-spy>

Comparison of 3 concepts: <https://www.javatpoint.com/mock-vs-stub-vs-spy>

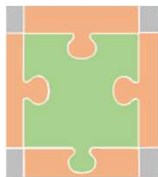
From SUT to Integration Testing over Unit Testing

System under
Test (SUT)



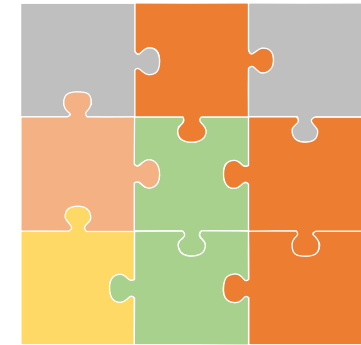
Green: class in focus
Yellow: dependencies
Grey: other unrelated classes

Unit test



Green: class in focus
Orange: mocks for
dependent classes

Integration testing



Green: classes in focus
(their integration)
Orange: mocks for
dependent classes
Red: Stubs to interact
logically
Yellow: Spy – using the
'real' dependencies, when
no stub implementation is
present

Figures inspired by: <https://www.jrebel.com/blog/mock-unit-testing-with-mockito>

Spring Test & Mockito

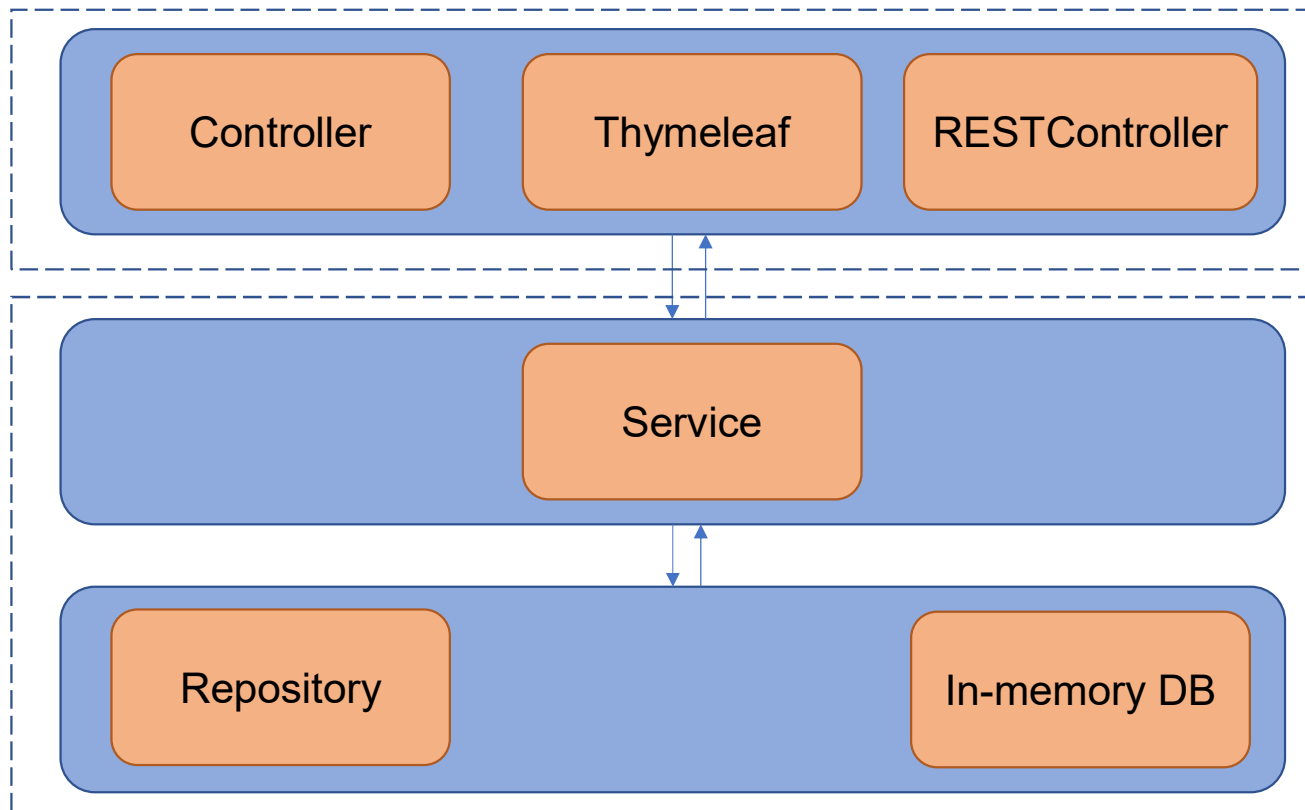
- Mockito is a testing framework for Java applications (<https://site.mockito.org/>)
- Provides functionality for mocking, stubbing and spying
- Spring automatically includes Mockito and provides additional annotations like `@MockBean` and `@SpyBean`
- Mock beans are automatically reset after each test method – otherwise behavioral test would fail
- Every test method annotated `@Transactional` is automatically rolled back (“caution should be taken if Spring-managed or application-managed transactions are configured with any propagation type other than `REQUIRED` or `SUPPORTS`.”)
- `spring-security-test` starter adds additional functionality to the test environment for access management or security related stuff

We included an example of another propagation type different to `REQUIRED` and `SUPPORTS` in our demo project under `test/.../MovieServiceTest`
JavaDoc source: <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/context/TestExecutionListener.html>

Mockito at a glance

- Getting a mock object via `mock(YourObject.class)`
- Mocking with Mockito – behavioral verification – selection of methods
 - Verification for number of invocations (`times(n)`, `never()`, `only()`, `atLeastOnce(n)`, `atLeast(n)`, `atMost(n)`)
e.g. `verify(mockObject, times(1)).myServiceMethod();`
 - Verification of order of invocation: `inOrder(mockA, mockB)`
 - Verification that no non-verified actions occurred: `verifyNoMoreInteractions(mock)`
 - Verification that no interaction occurred: `verifyNoInteractions(mock)`
- Stubbing with Mockito – simulate logic of stubbed dependency
 - Stubbing, when a specific method is called
 - Option to return a specific value, throw an exception or provide a callback implementation
 - Parameters can be set statically or via `any(XY.class)`
`when(pingPongPlayerMock.method(any(Ball.class))).thenReturn("Victory")`
 - Other options `...thenThrow(throwable)`
 - Other options `...thenAnswer(answer/callback)`
 - Method chaining is possible with `thenXXX` methods
 - Exception: methods which return void
`doXY().when(mock).voidMethod();`
- Spying with Mockito – wrapping business objects to spies
 - `spy(object)`
 - Stub and mock methods can be used on the returned or annotated object

A possible test strategy



Test UI/User Interaction classes independent to the other parts of the application. Mock service and database interactions.

Make integration test since the repositories are normally generated by Spring. Use an in-memory DB to speed up tests (this helps you also to test transactions).

MVC SpringBootTest (1/2)

```

@SpringBootTest
@AutoConfigureMockMvc
public class MovieControllerTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private MovieService movieService;

    @BeforeEach
    public void initCommonUsedData() {
        ...
    }

    @Test
    public void getRequestMovies_anonymousUser_redirectToLogin()
        throws Exception {

        when(this.movieService.findAll(any(PageRequest.class)))
            .thenReturn(new PageImpl<>(this.movies));
        this.mvc.perform(get("/movies").with(anonymous()))
            .andExpect(status().is3xxRedirection())
            .andExpect(redirectedUrlPattern("**/login"));
    }
}

```

SpringBootTest creates an application context. It does not start a server by default, only when port is configured via webEnvironment property (check the documentation)

Need a mocked web environment for testing endpoints, i.e. thymeleaf

Add mock object to the application context

JUnit 5 annotation. Method is executed before each test method (mocks are reset by default)

Testing GET request to movies endpoint as anonymous user. Result should be a redirect to login page.
(Stubbing the findAll method of the movie service here)

<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-testing-spring-boot-integration-tests>
Integration tests with spring security: <https://www.baeldung.com/spring-security-integration-tests>

MVC SpringBootTest (2/2)

```

@Test
public void p...() throws Exception {

    when(this.movieService.findAll(any(PageRequest.class)))
        .thenReturn(new PageImpl<>(this.movies));
    Movie m = new Movie(...);

    List<Movie> spyList = Mockito.spy(this.movies);
    when(this.movieService.saveAndSetId(any(Movie.class)))
        .thenAnswer(invocation -> {
            System.out.println(invocation.getArguments().length);
            Movie requested = invocation.getArgument(0, Movie.class);
            spyList.add(requested);
            return requested;
        });

    this.mvc.perform(this.createPostRequestBuilder(m)
        .with(csrf()).with(user(this.adminUser))
        .andExpect(status().is3xxRedirection()));

    verify(this.movieService).saveAndSetId(m);
    verify(this.movieService, times(1))
        .saveAndSetId(any(Movie.class));
    verify(spyList, times(1)).add(any(Movie.class));

    assertEquals(2, spyList.size());
}

```

Stubbing

Spying (delegating all methods to ArrayList implementation – only for demonstration purposes here)

Testing Spring Security (also the anonymous() call was a test security method)

Mocking

Plain unit testing

<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-testing>
 Integration tests with spring security: <https://www.baeldung.com/spring-security-integration-tests>

Data/Transaction Test Management

- Special test annotation (`@DataJpaTest`), but you can also work with `@SpringBootTest`
- Testing data access with an in-memory database is reasonable (h2 is on the classpath)
- Each test annotated with `@Transactional` is automatically rolled back, if only a single transaction context is present (check the propagation strategies)
- Since repositories are generated via the framework, we test the service layer and the database together

```
@SpringBootTest
public class MovieServiceTest {

    @Autowired
    private MovieService movieService;

    @Test
    @Transactional
    public void testCorrectIdHandlingForNewInsertedMovies() {
        int elementsInDb = // count elements in Movie table
        Movie m = movieService.saveAndSetId(new Movie("My Movie", false, 2003,
            "https://.png", null, null, null));
        assertEquals(elementsInDb + 1, m.getId());
    }
}
```

Some information on data jpa tests: <https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-testing-spring-boot-applications-testing-autoconfigured-jpa-test>

Testing REST Controller

- Due to hypermedia support of Spring HATEOAS, middleware adds additional data
- Therefore no easy way to parse the response to custom java objects
- Solution: Checking the response structure to be API compliant
 - (+) You know, when you introduce a breaking change in your API before your users
 - (-) A lot of code but worth the effort in bigger projects
- JsonPath and other libraries are helpful for checking the responses from the server

```
@Test
public void getRequestMovies_minimalFullAPICompliantResponse() throws Exception {
    // stubbing . . .
    Movie test = this.movies.get(0);

    this.mvc.perform(MockMvcRequestBuilders.get("/v1/movies")
                                                .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(MockMvcResultMatchers.jsonPath("$. _embedded.movies[0].title",
            Matchers.is(test.getTitle())))
        .andExpect(MockMvcResultMatchers.jsonPath("$. _embedded.movies[0].wonOscar",
            Matchers.is(test.isWonOscar())))
        . . .
}
```

<https://jsonpathfinder.com/>
<https://jsonpath.com/>

Final Remarks

Automagically – Pros & Cons

- For 90% of the cases, default auto-configuration is sufficient (and easy to use)
- Fewer LOC compared to other frameworks
- Lot of online help
- Annotation based configuration is good to read
- Great documentation
- For other cases it is hard to identify the error
- Loss of control and therefore flexibility
- Lot of online help (often misleading when problem only circumvented but not solved)
- Mechanisms are often influence each other – questions of priorities

<https://www.marcobehler.com/guides/spring-boot>

What we have seen . . .

