Hogeschool PXL
PXL-Digital

Java Advanced 2

Author
Nele Custers

April 18, 2023

Contents

Chapter 1

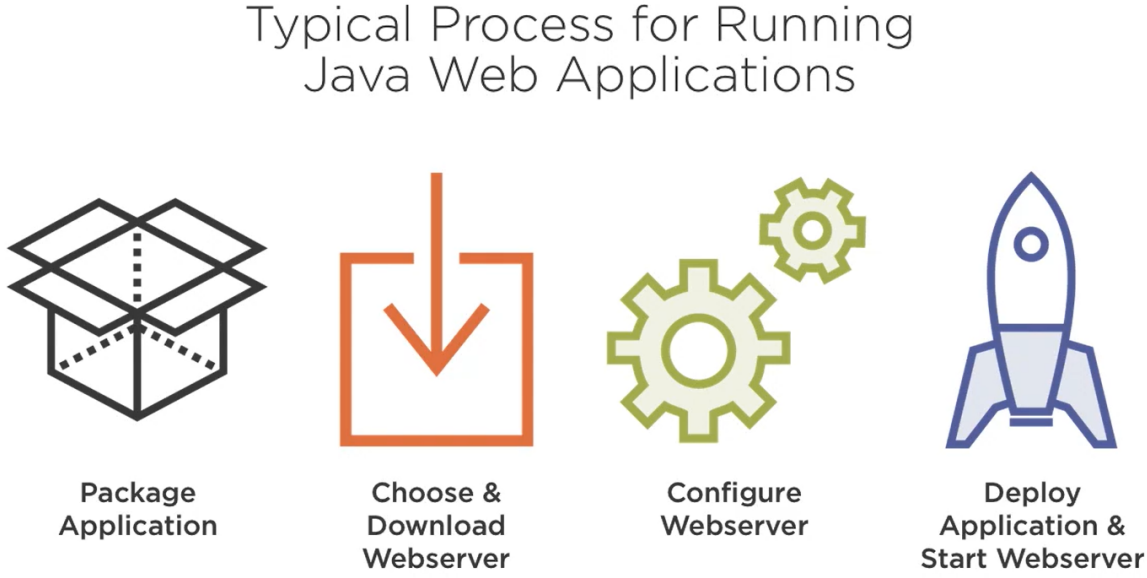
Introduction

Learning goals
The junior-colleague

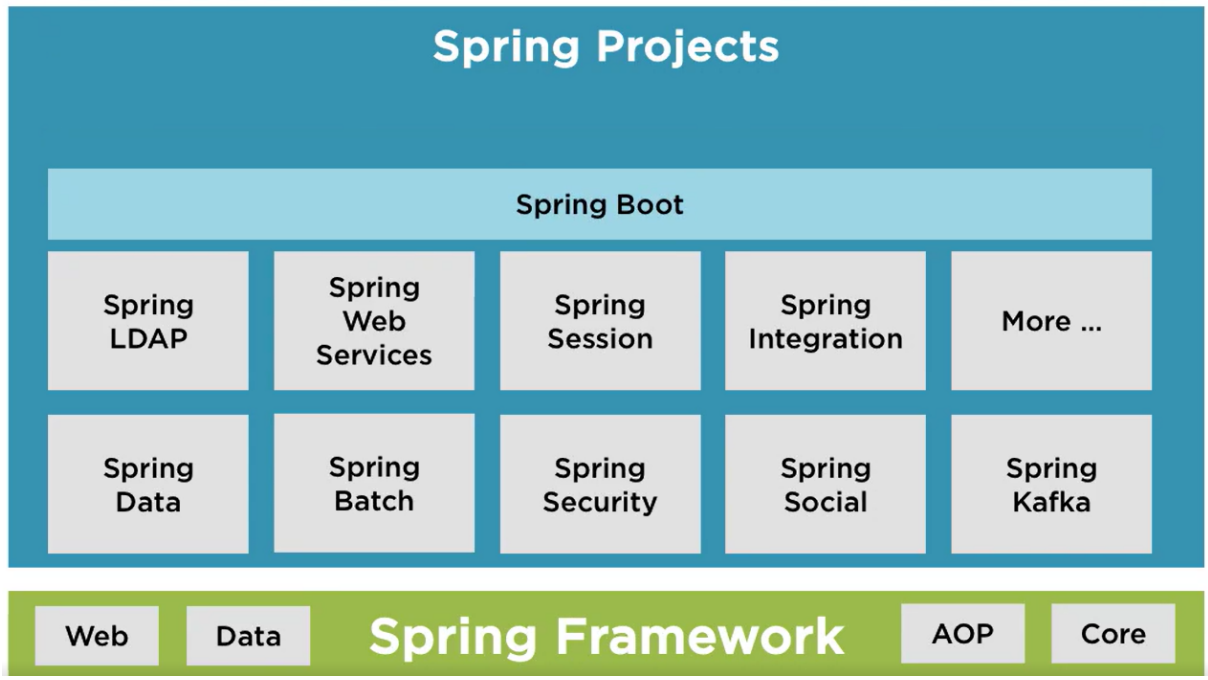
1. can describe what Spring is
2. can describe what Spring Boot is
3. can explain what a three-tier application is
4. can identify the three layers in a Spring Boot application
5. can explain the responsibilities of the layers in a Spring Boot application
6. can explain the architecture of a Spring RESTful web application
7. can explain what a DTO is
8. can explain what an entity-object is
9. can explain what dependency injection is
10. can explain what a Spring bean is
11. can explain what a Spring container is
12. can explain what a Spring Boot starter is

1.1 Enterprise Applications with Java

By 1999, Java had developed a loyal following among application developers and Sun saw an opportunity to extend the language for traditional enterprise workloads. With the launch of J2EE, and another technology that was gaining prominence — the application server — enterprises now had a platform that was designed to meet their needs with capabilities for things like security, scalability and reliability.



Spring came into being in 2003 as a response to the complexity of the early J2EE specifications. While some consider Java EE and its modern-day successor Jakarta EE to be in competition with Spring, they are in fact complementary. The Spring programming model does not embrace the Jakarta EE platform specification; rather, it integrates with carefully selected individual specifications from the traditional EE umbrella. Spring started as a lightweight alternative to Java Enterprise Edition. Rather than develop components as heavyweight Enterprise JavaBeans (EJBs), Spring offered a simpler approach to enterprise Java development, utilizing dependency injection and aspect-oriented programming to achieve the capabilities of EJB with plain old Java objects (POJOs). But while Spring was lightweight in terms of component code, it was heavyweight in terms of configuration. Initially, Spring was configured with XML (and lots of it). It provides everything you need to create Java enterprise applications. Spring offers the flexibility to create many kinds of architectures depending on an application's needs. As of Spring Framework 6.0, Spring requires Java 17+.



Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run java applications.

1.2 What is Spring Boot?

Spring Boot is an open-source Java framework to create production-ready, standalone Spring applications. It's a robust, widely used framework. The creation of this framework was facilitated by the desire to simplify the development of applications on the popular Java EE technology stack from Oracle, which was very complex and difficult to use at the time. With very little configuration, you can create easily your first Spring Boot application.

Let's look at some advantages of Spring Boot for developers

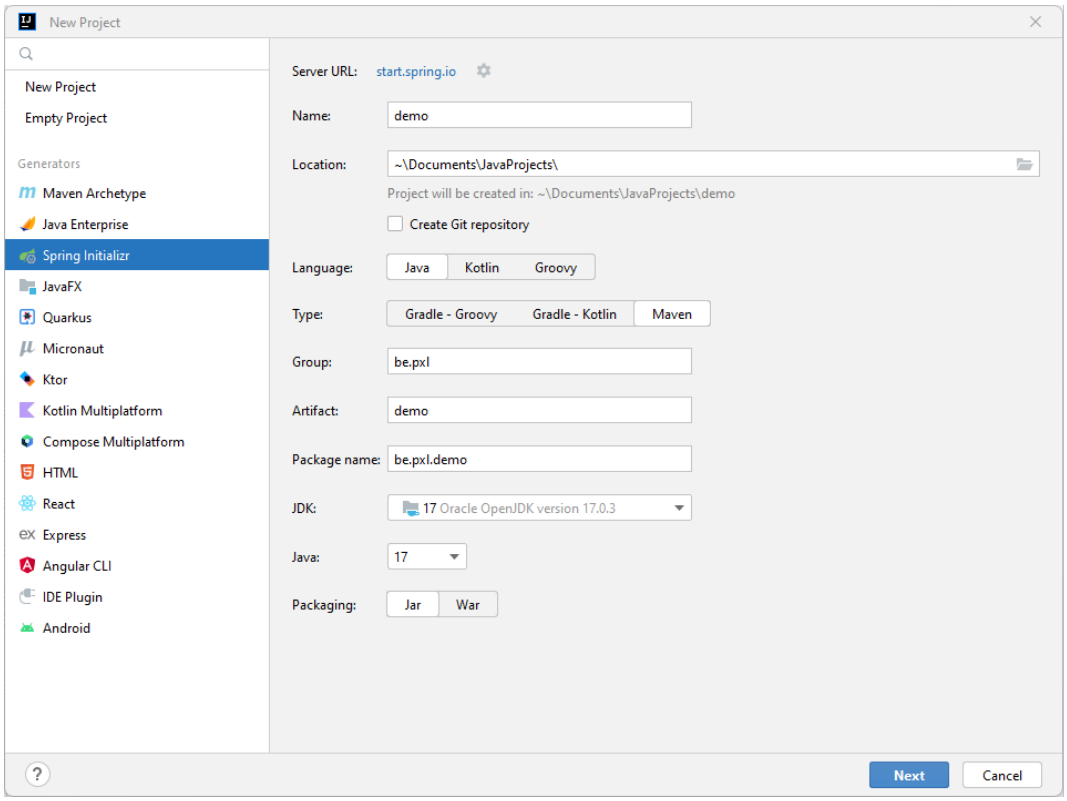
- speed up the process of creating and deploying application
- create standalone applications with less or almost no configuration overhead
- easy to learn framework
- increase productivity of developers

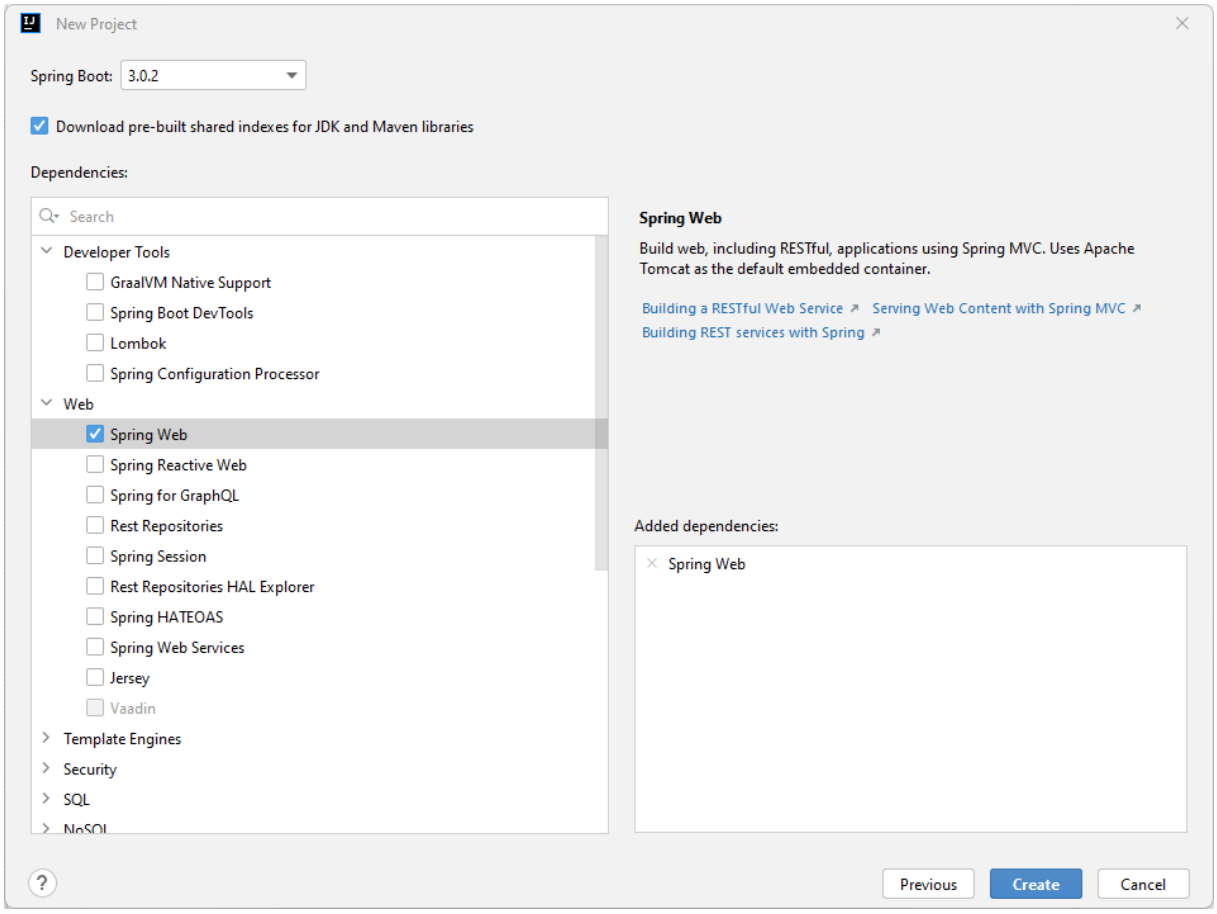
1.3 Bootstrapping a simple application

1.3.1 Using Spring Initializr

Spring Initializr is a web application that can generate a Spring Boot project. The url for this web application is `https://start.spring.io/`. You can select the necessary configuration, including the build tool, language, version of the Spring Boot framework, and any dependencies for your project. IntelliJ IDEA Ultimate provides the Spring Initializr

project wizard that integrates with the Spring Initializr API to generate and import your project directly from the IDE.





We select Spring Web dependency. Spring Web uses Spring MVC. It is used for building RESTful Web Services. Spring MVC provides the annotation @RestController for classes that implement the REST endpoints. To run a RESTful Web Service you need a web container. Spring Boot will automatically add an embedded Tomcat web container to your project. If you prefer another web container, you can update Spring Boot's configuration. Finally Jackson is a popular third-party library for converting Java-objects to JSON and vice versa.

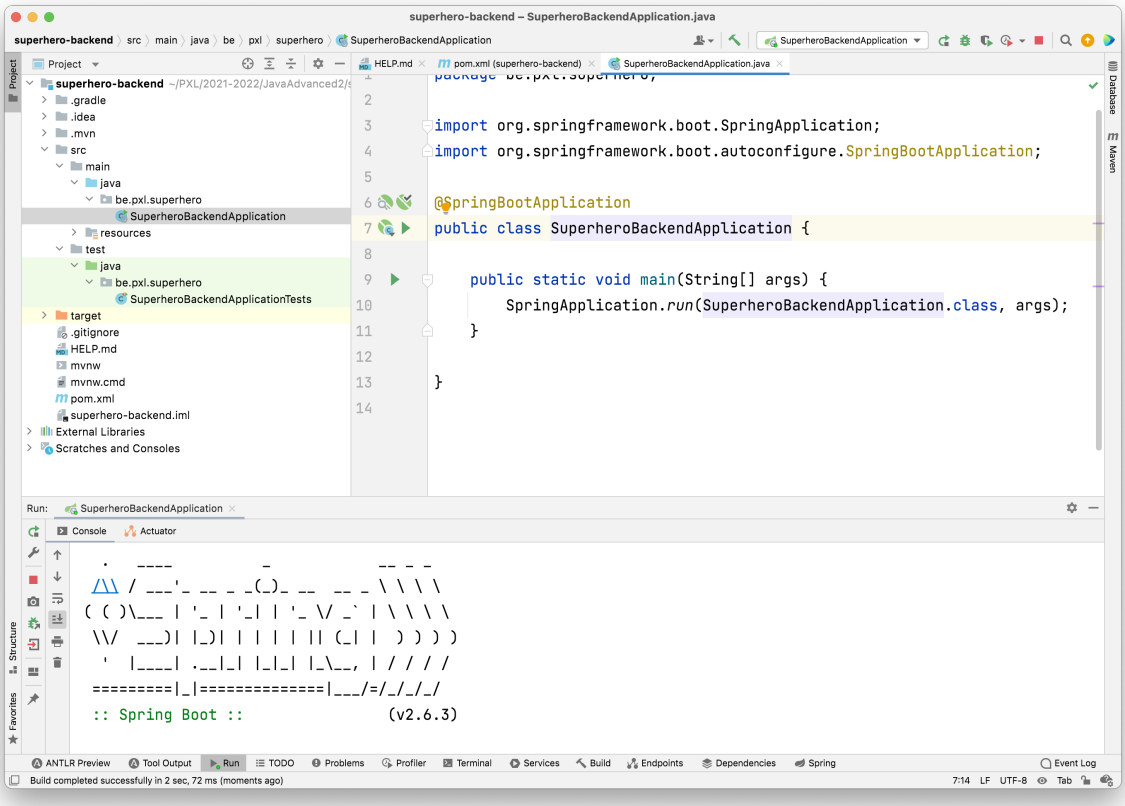
Exercise 1.1. Create the demo project. You can use the wizard in IntelliJ IDEA Ultimate or https://start.spring.io/.

1.4 Running the demo project

The starting point of a Spring Boot application is the class with the main-method and annotated with @SpringBootApplication. This class can be found in the folder /src/main/-java. Spring Boot offers a lot of annotations to reduce the workload of developers.

```
package be.pxl.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {

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

}
```

By running the main-class you start your Spring Boot application.



Currently our Spring Boot application only shows a whitelabel error page. This error page is available when you perform a GET for URL `http://localhost:8080`.

Port 8080 is the default port. If this port is not available you will see an error message in Spring Boot's logging.

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Web server failed to start. Port 8080 was already in use.
```

The port number can be changed in the file application.properties. You have to add the property server.port here with the desired port number.

```
server.port=8081
```

1.4.1 The Maven pom file

POM stands for Project Object Model. It is an XML representation of a Maven project held in a file named pom.xml. This file can be found in your project directory. The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. We will cover Maven in chapter 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    ↪ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
            ↪ https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>be.pxl</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>demo</description>
    <properties>
        <java.version>17</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```

spring-boot-starter-parent is a starter project that provides the default configuration for spring-based applications. Here you choose the version of Spring Boot.

For large projects, managing the dependencies is not always easy. Spring Boot solves this problem by grouping certain dependencies together. These groups of dependencies are called starters. All Spring Boot starters are named following the same naming pattern. The all start with spring-boot-starter-*, where * indicates the purpose and functionality provided by the starter.

spring-boot-starter-web adds all the libraries we need to develop web components. An embedded server will be provided in the Spring Boot project. Therefore the environment where the Spring Boot project is executed does not need to have a pre-installed server. The default embedded server for Spring Boot is Tomcat. The Spring MVC framework which provides all classes for developing RESTful web services is also part of spring-boot-starter-web.

spring-boot-starter-test (with scope test) is the starter for testing Spring Boot applications with libraries including JUnit Jupiter, Hamcrest and Mockito.

1.4.2 A Rest Controller

RestControllers are used for making RESTful web services with the help of the @RestController annotation. The RestController allows to handle all HTTP methods such as GET, POST, DELETE, PUT.

```
package be.pxl.demo.api;

import jakarta.annotation.PostConstruct;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

@RestController
public class GreetingController {

    private final List<String> messages = new ArrayList<>();
    private static final Random RANDOM = new Random();

    @PostConstruct
    public void init() {
```

```
        messages.add("Peek-a-boo!");
        messages.add("Howdy-doody!");
        messages.add("My name's Ralph, and I'm a bad guy.");
        messages.add("I come in peace!");
        messages.add("Put that cookie down!");

    }

    @GetMapping("/hello")
    public String doGreeting() {
        return messages.get(RANDOM.nextInt(messages.size()));
    }
}
```

When Spring's DispatcherServlet receives a request for the URL /hello it looks for the correct @RestController to handle the request. The annotation @RequestMapping is used for mapping all incoming HTTP request URLs to the corresponding controller methods. The DispatcherServlet will be discussed in detail in a later chapter. The Spring container is responsible for managing an instance of the @RestController. When an instance of our GreetingController is created, the init method is called to initialise all possible greetings. The @GetMapping is a specialized version of @RequestMapping and is used for methods handling HTTP GET requests. Therefore, when we call the URL http://localhost:8080/hello, the method doGreeting() in the GreetingController will handle the request and return a greeting message in text format.

Exercise 1.2. Create the package *be.pxl.demo.api*. Add the class **GreetingController** in this package. Restart the Spring Boot application and open the URL http://localhost:8080/hello in a browser.

1.4.3 Auto-configuration

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the dependencies that you have added. To gain some insight in this auto-configuration let's add a line in the application.properties file. This file can be found in the directory /src/main/resources.

```
logging.level.org.springframework=debug
```

logging.level.org.springframework is a application properties. A list of all available application properties can be found at https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html.

Exercise 1.3. Add the line above to the application.properties file and restart the Spring Boot application.

In the console you will find all the auto-configuration Spring Boot is doing.

1.4.4 @SpringBootApplication

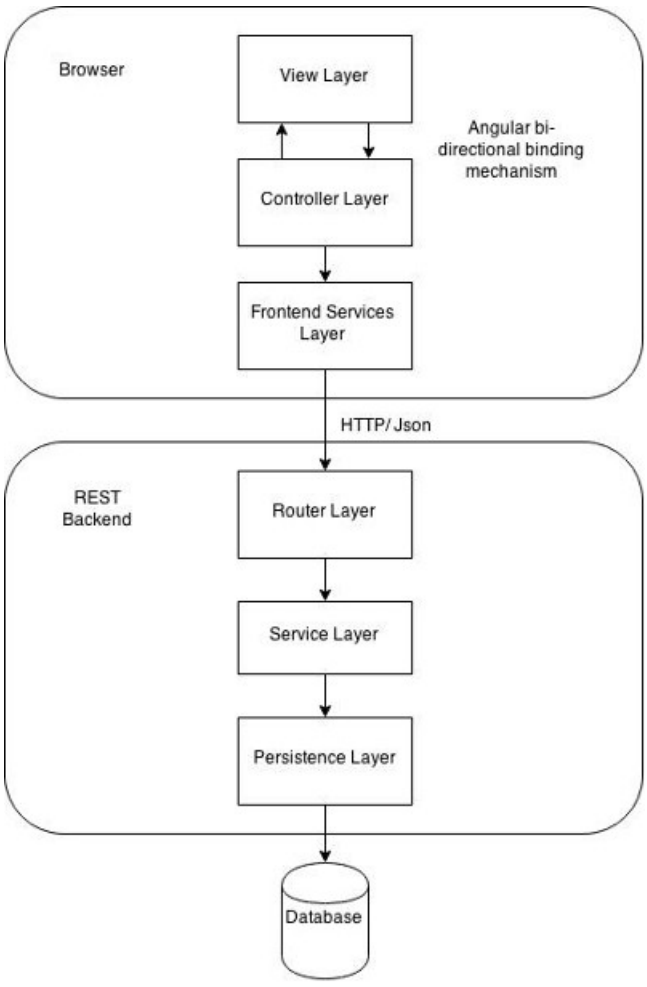
Java annotations are a mechanism for adding metadata information to our source code. An annotation processor processes these annotations at compile time or runtime to provide functionality such as code generation, error checking, etc.

@SpringBootApplication annotation is used to enable following three features:

- @EnableAutoConfiguration: enable Spring Boot's auto-configuration mechanism
- @ComponentScan: enable @Component scan on the package where the application is located
- @Configuration: allow to register extra beans in the context or import additional configuration classes

1.5 Components and dependency injection

All our REST backend applications will consist of 3 layers: router-layer (or API-layer), service- or business-layer and persistence-layer.



- **Router- or presentation layer:** Handling and processing HTTP requests, translating JSON parameters to objects, authentication of users and protecting data from maleficent users, ...

- **Service- or business logic layer:** This layer contains the business logic, the core functionality of your application. All calculations, decisions, evaluations, data processing,... are handled by this layer.
- **Data- or persistence layer:** This layer is responsible for interacting with the database. This layer stores and retrieves the application data in the database.

The typical components for each layer are annotated. These annotated components are managed by Spring. Developers are not responsible for creating the objects of these component classes. The actual objects are created and injected by Spring wherever we need them. In later chapters we will discuss the concept of dependency injection in detail.

The annotations for the Spring components we will use are:

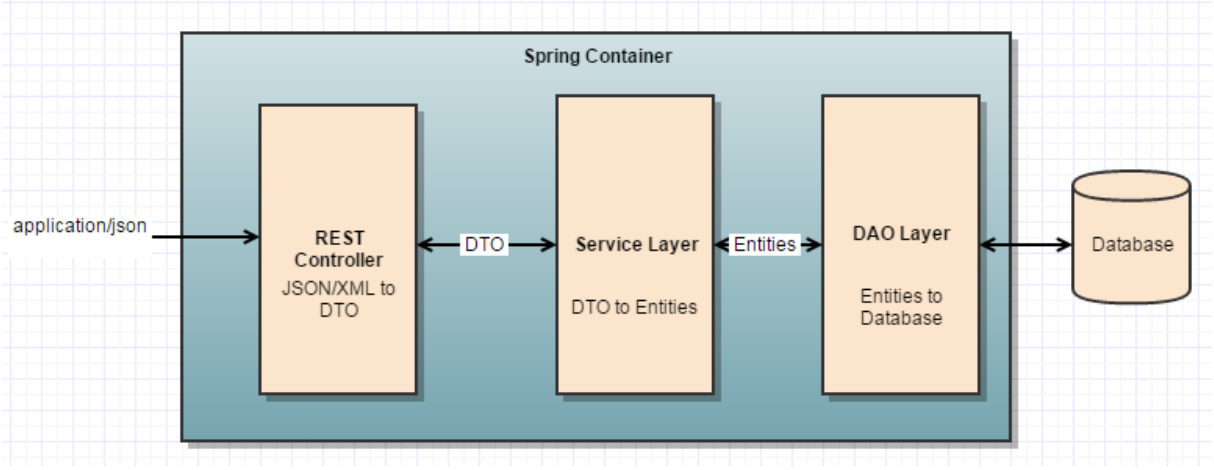
- @Component: generic annotation for all components managed by Spring
- @RestController: indicate the components in the API layer
- @Service: indicate the components in the service layer
- @Repository: indicate the components in the persistence layer

All of these application components (@Component, @Service, @Repository, @Controller, and others) are automatically registered as Spring Beans.

Here is the definition of beans from the Spring framework documentation:

> In Spring, the objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container.

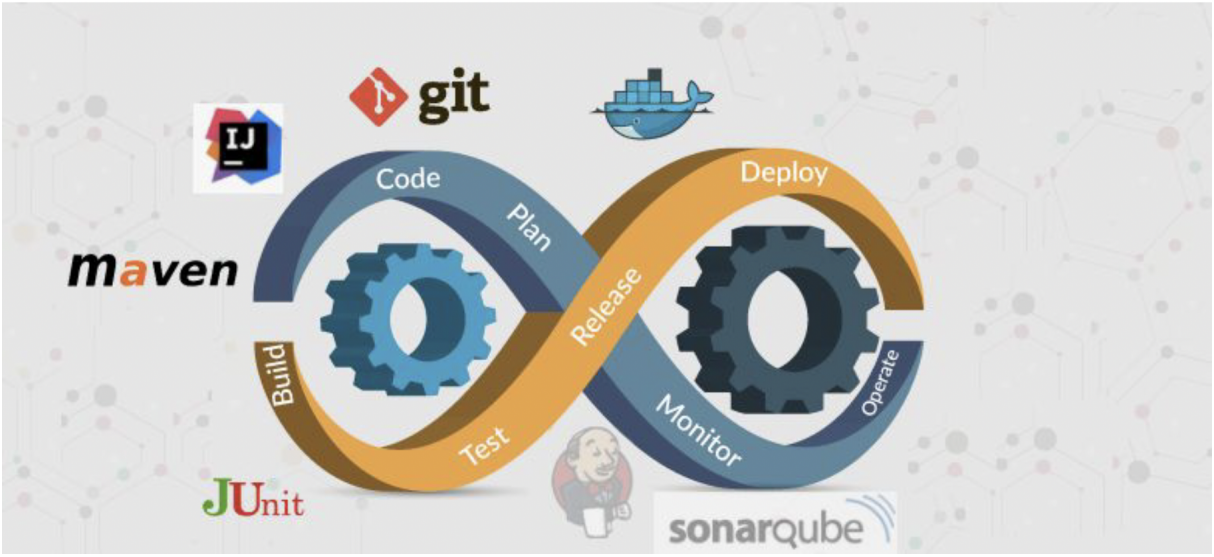
IoC or Inversion of Control is the process in which an object defines its dependencies without creating them. The IoC container is responsible for constructing the dependencies.



The image shows all the layers that we develop in our REST backend applications. The RestController in the API layer provides REST endpoints. The API layer communicates via DTOs or Data Transfer Objects with the service layer. The service layer implements all business logic. DTOs are mapped to entity-objects and passed along the persistence layer. The persistence layer is responsible for storing the entity-objects in the database.

The Spring container is at the core of the Spring Framework. The container is responsible for creating and managing the objects for classes annotated with @Service, @Repository,... The container will provide these objects (beans) when they are needed by other beans.

1.6 DevOps Tools for Java Developers



Chapter 2

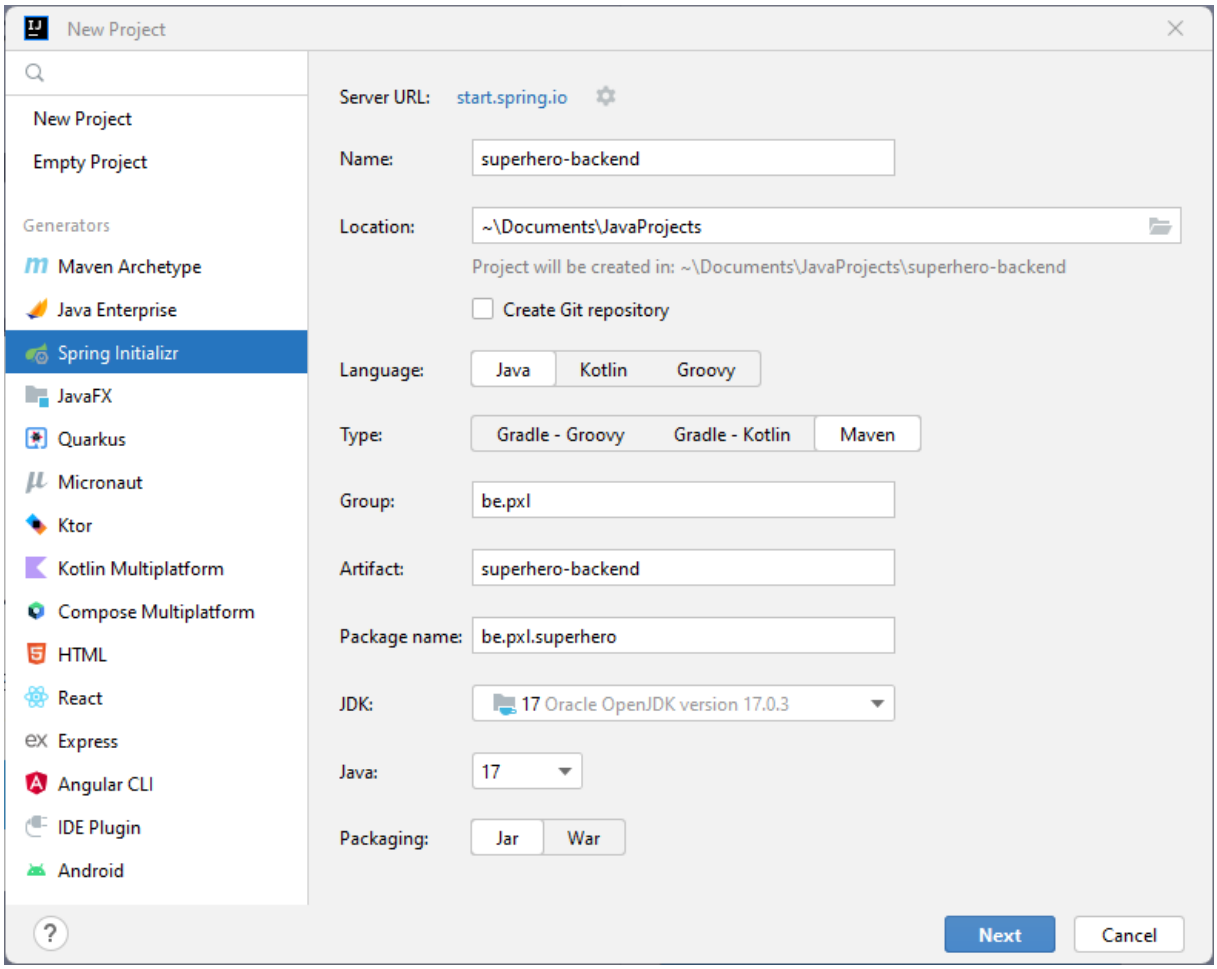
Creating a Spring Boot application

Learning goals
The junior-colleague
1. can identify the different components in a Spring Boot application
2. can explain what REST is
3. can explain what an entity-class is
4. can create a RestController
5. can run a Spring Boot application
6. can enable CORS support in a Spring Boot application
7. ...

2.1 Creating a Spring Boot project

This chapter will guide you through the implementation of your first Spring Boot application. You will create a RESTful webservice for managing the information of a "Super Hero Company". We will implement REST endpoints for creating, updating and deleting superheroes. Later you will add functionality to create, update and delete missions for the superheroes.

We use Spring Intializr to bootstrap this Spring Boot application.



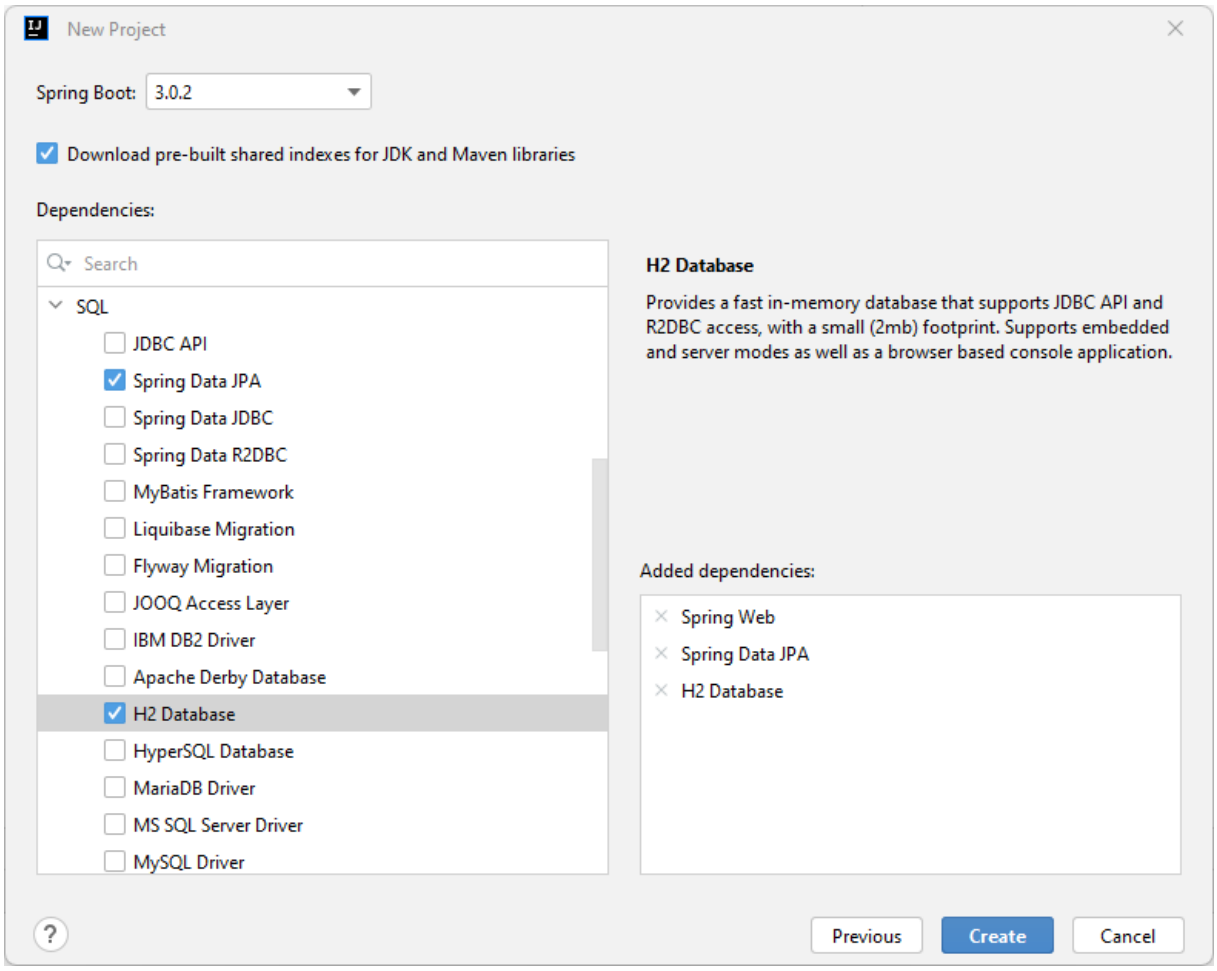
In the first screen of the wizard you have to provide metadata about your project. The information you have to provide is:

- **Group:** unique identification of your project across all projects. A group follows Java's package name rules. It also infers the root package name to use.
- **Artifact:** the name of a project artifact without version. It also infers the name of the project.
- **Name:** display name of the project that also determines the name of your Spring Boot application. For instance, if the name of your project is my-app, the generated project will have a MyAppApplication class
- **Description:** description of the project.
- **Package Name:** root package of the project. If not specified, the value of the group attribute is used
- **Packaging:** project packaging. You can choose either jar or war projects.
- **Java Version:** the Java version to use
- **Language:** the programming language to use

During this course we use Maven as build tool for our projects. In the next chapter we will discuss Maven into detail.

Our backend provides a REST API. A clear explanation about REST can be found at

codecademy. To implement REST endpoints we need some third-party libraries: Spring MVC, Tomcat and Jackson. All these libraries are bundled in one starter: **spring-boot-starter-web**.



Besides Spring Web, we add Spring Data JPA to the project. Using this starter depencency we can use JPA (the Java or Jakarta Persistence API) to access a database. The dependency spring-boot-starter-data-jpa will be added to the pom.xml file. This one starter adds different libraries for easily accessing a database. Finaly the H2 Database dependency is added. This is an in-memory database and you need zero configuration to use this database. This is nice for fast prototyping, but all your data is lost once you restart the application. All database related technologies will be explained in detail during this course.

Exercise 2.1. Create your superhero Spring Boot project. You can use the wizard in IntelliJ IDEA Ultimate or `https://start.spring.io/`.

2.2 Storing and retrieving data

2.2.1 Entity-class Superhero

First we need a class for respresenting the objects that are stored and retrieved from the database. Each object of this class will represent one row of a table in the relational

database. We implement the class Superhero. To save Superhero-objects to the database and retrieve Superhero-objects from the database, we annotate this class and make it an entity-class.

Here is the entity-class Superhero:

```
package be.pxl.superhero.domain;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name="superheroes")
public class Superhero {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    private String superheroName;

    public Superhero() {
        // JPA only
    }

    public Superhero(String firstName, String lastName, String
        ↪ superheroName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public Long getId() {
```

```
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    public void setSuperheroName(String superheroName) {
        this.superheroName = superheroName;
    }

    @Override
    public String toString() {
        return superheroName;
    }
}
```

The annotation @Entity indicates that the class is actually an entity-class. Spring is able to automatically generate the database table (superheroes) with all its fields in the H2 database. The primary key of the table is marked with the annotation @Id. Further, with the annotation @GeneratedValue(strategy = GenerationType.IDENTITY) we don't have to assign the primary keys to the objects. The database itself is responsible for generating and assigning the primary keys.

Exercise 2.2. Add the entity class Superhero to your project. Create the package package *be.pxl.superhero.domain* for this class.

2.2.2 Repository

To execute queries in the database we need an extra class (or interface) called a **repository**. Spring is able to automatically generate database queries. When you extend the interface JpaRepository, simple queries are already available without writing one line of

code. The generic interface JpaRepository only needs to know which data it must store and retrieve. Therefore it needs to know the name of the entity-class and the data type of the primary key of the entity-class.

```
package be.pxl.superhero.repository;

import be.pxl.superhero.domain.Superhero;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface SuperheroRepository extends JpaRepository<Superhero,
    ↪ Long> {
}
```

When you open the documentation of the generic interface JpaRepository, you can see which functionality is available for the developer.

Exercise 2.3. Add the repository-interface SuperheroRepository to the project. Create the package *be.pxl.superhero.repository* for this interface.

2.2.3 Service-layer

The classes in the service-layer are responsible for the business logic. These classes use the repositories to save and retrieve data from the database. It is good practice to provide an interface for every class in the service-layer (service-class). The service-classes may never return entity-objects. We need DTOs (Data Transfer Objects) to pass data from the service-layer to the API-layer.

```
package be.pxl.superhero.service;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroRequest;

import java.util.List;

public interface SuperheroService {

    List<SuperheroDTO> findAllSuperheroes();

    SuperheroDTO findSuperheroById(Long superheroId);

    Long createSuperhero(SuperheroRequest superheroRequest);

    SuperheroDTO updateSuperhero(Long superheroId, SuperheroRequest
        ↪ superheroRequest);

    boolean deleteSuperhero(Long superheroId);
```

```
}
```

```
package be.pxl.superhero.api;

import be.pxl.superhero.domain.Superhero;

public class SuperheroDTO {

    private final Long id;
    private final String firstName;
    private final String lastName;
    private final String superheroName;

    public SuperheroDTO(Superhero superhero) {
       this.id = superhero.getId();
        this.firstName = superhero.getFirstName();
        this.lastName = superhero.getLastName();
        this.superheroName = superhero.getSuperheroName();
    }

    public Long getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

}
```

```
package be.pxl.superhero.api;

public class SuperheroRequest {

    private String firstName;
    private String lastName;
    private String superheroName;

    public SuperheroRequest(String firstName, String lastName, String
```

```
        ↪ superheroName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    public void setSuperheroName(String superheroName) {
        this.superheroName = superheroName;
    }

}
```

The class SuperheroServiceImpl, annotated with @Service, provides the implementation for the interface SuperheroService. All CRUD-operations (create-read-update-delete) for Superhero-objects are provided here. In the class SuperheroServiceImpl all business logic can be implemented. If we, for example, have to check that the superheroname of a superhero is unique, this class is the place to implement these checks. (At the moment we will not yet implement this business rule!)

The SuperheroRepository is autowired in the SuperheroServiceImpl. Hence the service-class can save and retrieve data from the database with the help from this repository.

```
package be.pxl.superhero.service.impl;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.domain.Superhero;
import be.pxl.superhero.exception.ResourceNotFoundException;
import be.pxl.superhero.repository.SuperheroRepository;
```

```java
import be.pxl.superhero.service.SuperheroService;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class SuperheroServiceImpl implements SuperheroService {

    private final SuperheroRepository superheroRepository;

    public SuperheroServiceImpl(SuperheroRepository superheroRepository) {
        this.superheroRepository = superheroRepository;
    }

    public List<SuperheroDTO> findAllSuperheroes() {
        return superheroRepository.findAll()
                .stream().map(SuperheroDTO::new)
                .toList();
    }

    public SuperheroDTO findSuperheroById(Long superheroId) {
        return superheroRepository.findById(superheroId)
                 .map(SuperheroDTO::new)
                .orElseThrow(() -> new
                    ↪ ResourceNotFoundException("Superhero", "ID",
                    ↪ superheroId));
    }

    public Long createSuperhero(SuperheroRequest superheroRequest) {
        Superhero superhero = new Superhero();
        superhero.setFirstName(superheroRequest.getFirstName());
        superhero.setLastName(superheroRequest.getLastName());
        superhero.setSuperheroName(superheroRequest.getSuperheroName());
        Superhero newSuperhero = superheroRepository.save(superhero);
        return newSuperhero.getId();
    }

    public SuperheroDTO updateSuperhero(Long superheroId, SuperheroRequest
        ↪ superheroRequest) {
        return superheroRepository.findById(superheroId).map(superhero -> {
            superhero.setFirstName(superheroRequest.getFirstName());
            superhero.setLastName(superheroRequest.getLastName());
            superhero.setSuperheroName(superheroRequest.getSuperheroName());
            return new SuperheroDTO(superheroRepository.save(superhero));
        }).orElseThrow(() -> new ResourceNotFoundException("Superhero",
            ↪ "id", superheroId));
    }
```

```
    public boolean deleteSuperhero(Long superheroId) {
        return superheroRepository.findById(superheroId)
                .map(superhero -> {
                    superheroRepository.delete(superhero);
                    return true;
                }).orElseThrow(() -> new
                    ↪ ResourceNotFoundException("Superhero", "id",
                    ↪ superheroId));

    }
}
```

```
package be.pxl.superhero.exception;

public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String resource, String field, String
        ↪ value) {
        super("Not found: " + resource + " with " + field + "=" + value);
    }

    public ResourceNotFoundException(String resource, String field, long
        ↪ value) {
        this(resource, field, Long.toString(value));
    }
}
```

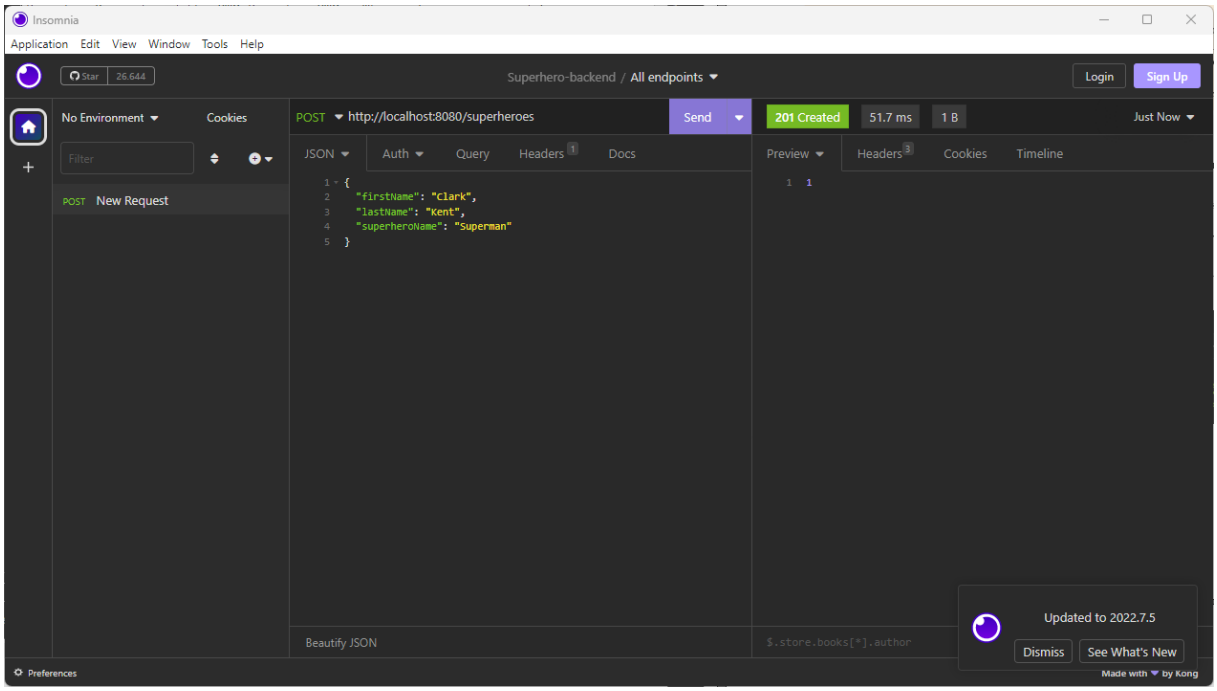
Exercise 2.4. Create the package *be.pxl.superhero.api* and add the request-object SuperheroRequest and the DTO SuperheroDTO. These classes are used for communication with the API-layer. Add the interface SuperheroService en the implementation SuperheroServiceImpl to your project. The interface is located in the package *be.pxl.superhero.service*. The implementation is located in the package *be.pxl.superhero.service.impl*. Finally you add the exception-class ResourceNotFoundException to the package *be.pxl.superhero.exception*.

2.2.4 REST controller

Now we can add the REST endpoints for creating, updating, deleting and retrieving superheroes.

For creating a new superhero we offer a POST-request with a requestbody in JSON-format. This requestbody holds all information about the new superhero. You already added the class SuperheroRequest to the project. This class is used for mapping the data of the requestbody to an object.

To test the implemented REST endpoints, you can use postman (https://www.postman.com/) or insomnia (https://insomnia.rest/).



Look at the method createSuperhero in the class below. The datatype of the parameter is SuperheroRequest and is annotated with @RequestBody. Therefore the body of the HTTP request in JSON-format is mapped to an object of the class (when the fields correspond).

The RestController uses the implementation of SuperheroService to map this SuperheroRequest to a Superhero-entity. Finally, the SuperheroRepository is responsible for saving the Superhero-entity in the database.

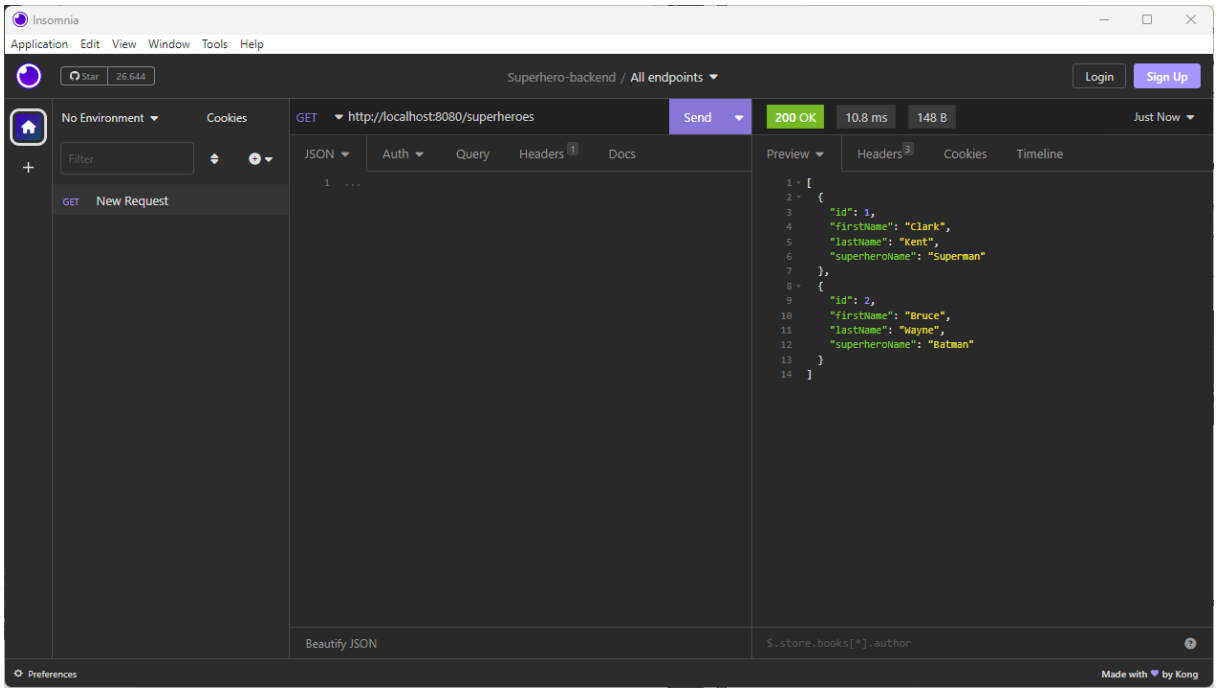
```
package be.pxl.superhero.api;

import be.pxl.superhero.service.SuperheroService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/superheroes")
public class SuperheroController {

    private final SuperheroService superheroService;
```

```
    public SuperheroController(SuperheroService superheroService) {
        this.superheroService = superheroService;
    }

    @GetMapping
    public List<SuperheroDTO> getSuperheroes() {
        return superheroService.findAllSuperheroes();
    }

    @GetMapping("/{superheroId}")
    public SuperheroDTO getSuperheroById(@PathVariable Long superheroId) {
        return superheroService.findSuperheroById(superheroId);
    }

    @PostMapping
    public ResponseEntity<Long> createSuperhero(@RequestBody
        ↪ SuperheroRequest superheroRequest) {
        return new
            ↪ ResponseEntity<>(superheroService.createSuperhero(superheroRequest),
            ↪ HttpStatus.CREATED);
    }

    @PutMapping("/{superheroId}")
    public SuperheroDTO updateSuperhero(@PathVariable Long superheroId,
        ↪ @RequestBody SuperheroRequest superheroRequest) {
        return superheroService.updateSuperhero(superheroId,
            ↪ superheroRequest);
    }

    @DeleteMapping("/{superheroId}")
    public ResponseEntity<Void> deleteSuperhero(@PathVariable Long
        ↪ superheroId) {
        boolean deleted = superheroService.deleteSuperhero(superheroId);
        return deleted? new ResponseEntity<>(HttpStatus.OK) : new
            ↪ ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

Exercise 2.5. Add @RestController SuperheroController to your Spring Boot application. Restart the project and create a new superhero with insomnia or postman. Next you call the REST endpoint to retrieve all superheroes or retrieve the superhero by id.
Here is the json-format to create a superhero:

```
{
    "firstName": "Clark",
```

```
    "lastName": "Kent",
    "superheroName": "Superman"
}
```



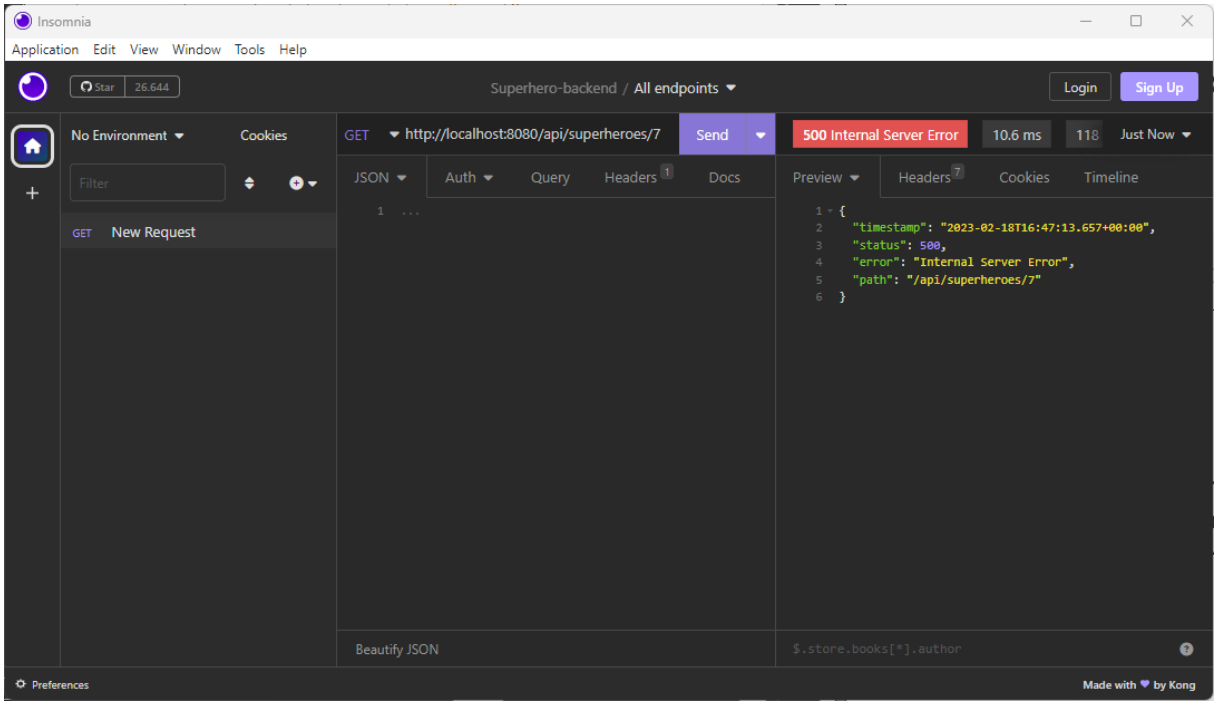
2.3 URL context path

When you want to add a prefix e.g. /api to all the URLs provided by the application, you can add the following key-value-pair in the file application.properties:

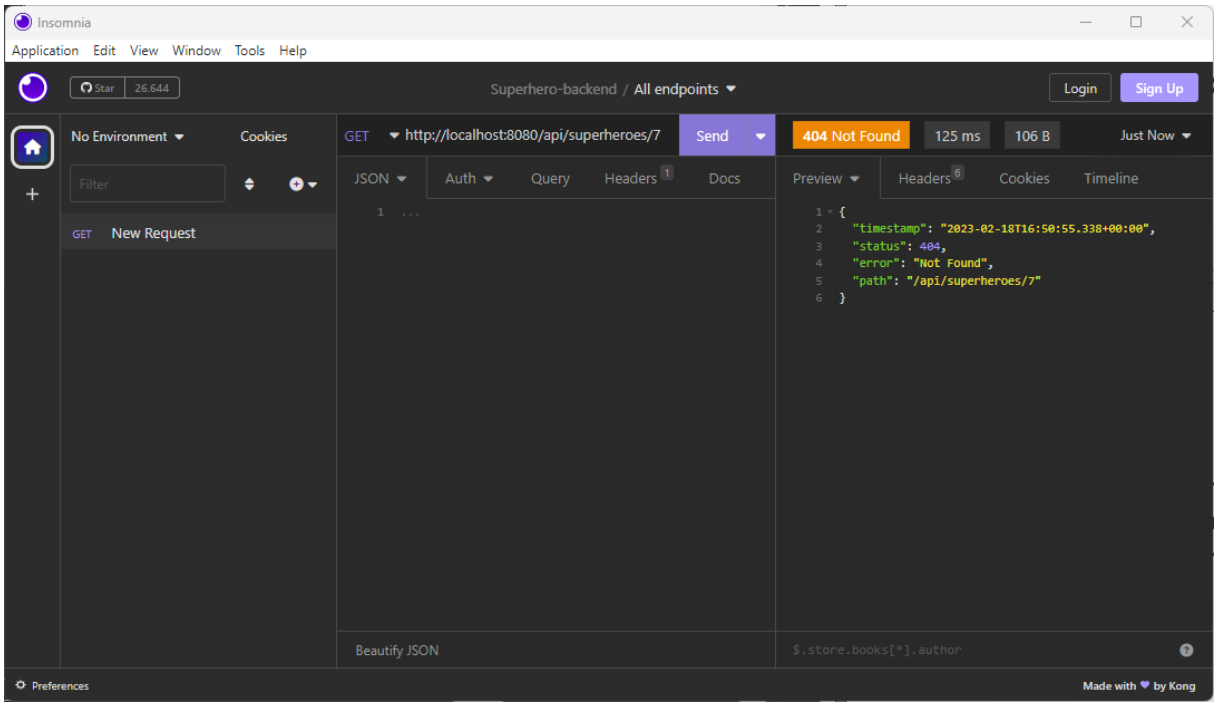
```
server.servlet.context-path=/api
```

Exercise 2.6. Change the context-path of your Spring Boot application. The prefix /api should be used for the application. Restart the application and test the endpoints.

2.4 Resource not found




```java
package be.pxl.superhero.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {
    public ResourceNotFoundException(String resource, String field, String
        ↪ value) {
        super("Not found: " + resource + " with " + field + "=" + value);
    }

    public ResourceNotFoundException(String resource, String field, long
        ↪ value) {
        this(resource, field, Long.toString(value));
    }
}
```



2.5 API documentation

Documentation about your Superhero REST API can be made available with the third-party library. You must add following dependencies to the file pom.xml.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

This documentation in xml-format that can be found by the URL http://localhost:8080/api/v3/api-docs is not user-friendly. However, if you open the URL http://localhost:8080/api/swagger-ui.html in you browser, you can see a user-friendly swagger page where you can even test your API.

H2 database

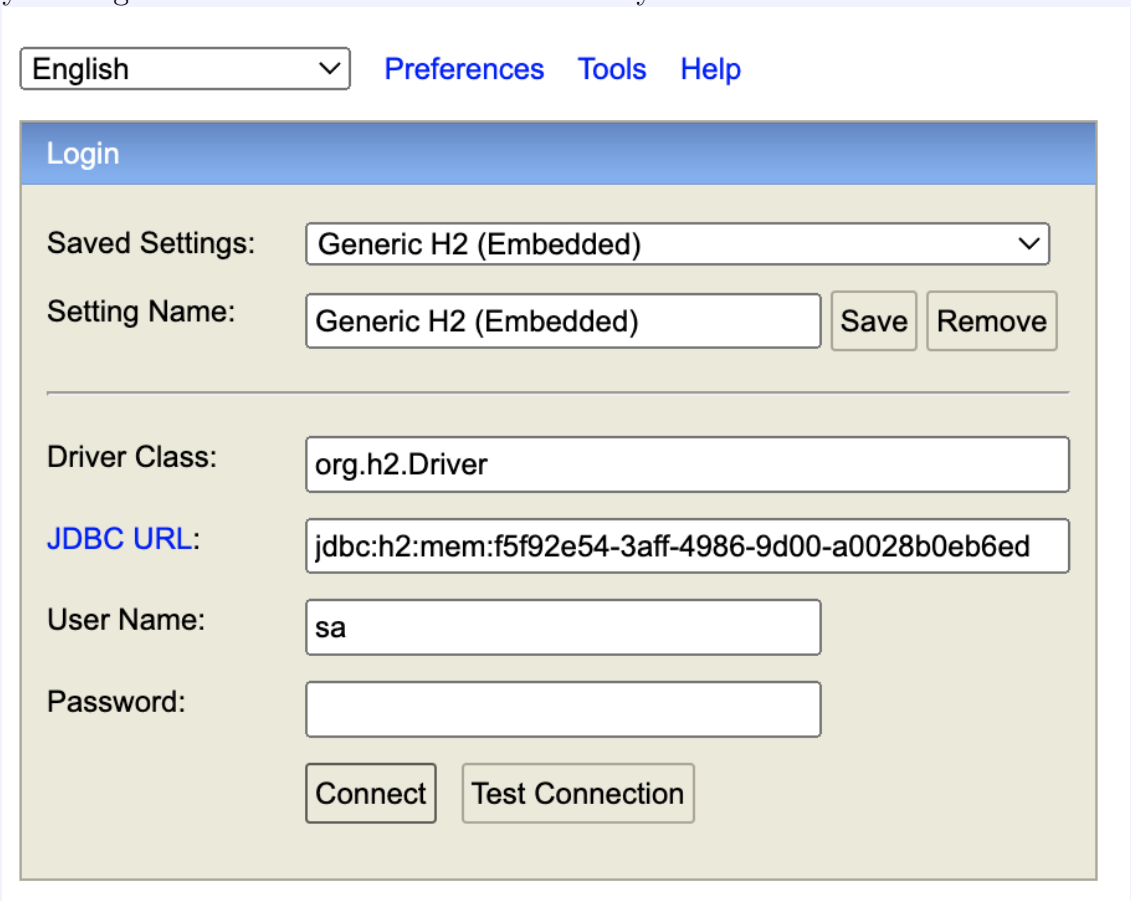
Our H2 in-memory database disappears when you close the application and all data is lost. You can use files to permanently save the data. If you want to inspect the data of your in-memory database, you can add the following property to the file application.properties (located in the resources folder)

```
spring.h2.console.enabled=true
```

When you start the Spring Boot application, you are given a unique identifier for your database.

```
H2 console available at '/h2-console'. Database available at
    ↪ 'jdbc:h2:mem:f5f92e54-3aff-4986-9d00-a0028b0eb6ed'
```

When you open the URL http://localhost:8080/api/h2-console in a browser and fill in the unique id and the username "sa" (the password field is left blank), you are given access to the tables and data of your database.



More information about the H2 database is available at https://www.baeldung.com/spring-boot-h2-database.

2.6 Frontend

A frontend for the superhero application written in Angular can be found at github. (Credits to: https://github.com/shoul10).

Source code
The frontend code is available at: https://github.com/custersnele/superhero-frontend.git

Our backend should support CORS to make the API available for the angular frontend. Cross-origin resource sharing (CORS) is a W3C specification used by most browsers.

Add the following configuration to your project to support CORS.

```
package be.pxl.superhero.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.CorsRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WebMvcConfig implements WebMvcConfigurer {
    private static final long MAX_AGE_SECS = 3600;

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
                .allowedOrigins("*")
                .allowedMethods("HEAD", "OPTIONS", "GET", "POST", "PUT",
                    ↪ "PATCH", "DELETE")
                .maxAge(MAX_AGE_SECS);
    }
}
```

Exercise 2.7. Add CORS support to your project. The class WebMvcConfig will be located in the package *be.pxl.superhero.config*. Restart the Spring Boot application. Download the frontend code from github and open it in a development environment (e.g. WebStorm). Start the frontend application (ng serve) and create, update and delete your superheroes.

2.7 Records

The Java record class type makes DTOs even more easy. It allows us to reduce boilerplate code.

For example, this record class:

```
package be.pxl.superherobackend.api;

public record SuperheroDTO (Long id, String firstName, String lastName,
    ↪ String superheroName) {
}
```

is equal to this traditional Java class:

```
package be.pxl.superherobackend.api;

public class SuperheroDTO {
    private final Long id;
    private final String firstName;
    private final String lastName;
    private final String superheroName;

    public SuperheroDTOFull(Long id, String firstName, String lastName,
        ↪ String superheroName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public Long getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        SuperheroDTO that = (SuperheroDTO) o;

        if (id != null ? !id.equals(that.id) : that.id != null) return
            ↪ false;
        if (firstName != null ? !firstName.equals(that.firstName) :
            ↪ that.firstName != null) return false;
        if (lastName != null ? !lastName.equals(that.lastName) :
            ↪ that.lastName != null) return false;
        return superheroName != null ?
```

```
            ↪ superheroName.equals(that.superheroName) : that.superheroName
            ↪ == null;
    }

    @Override
    public int hashCode() {
        int result = id != null ? id.hashCode() : 0;
        result = 31 * result + (firstName != null ? firstName.hashCode() :
            ↪ 0);
        result = 31 * result + (lastName != null ? lastName.hashCode() : 0);
        result = 31 * result + (superheroName != null ?
            ↪ superheroName.hashCode() : 0);
        return result;
    }

    @Override
    public String toString() {
        return "SuperheroDTO{" +
                "id=" + id +
                ", firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", superheroName='" + superheroName + '\'' +
                '}';
    }
}
```

A record class is a concise way to define an object that is shallowly immutable. The values inside a record are called record components. These are declared in the header of the record. Shallowly immutable means that the references that the immutable instance hold cannot change, but the values inside the referred instance can change.

It is possible to override the constructor in a record class.

Exercise 2.8. Replace SuperheroDTO and SuperheroRequest with record classes. Fix the code and test your application.

Chapter 3

Maven

Learning goals
The junior-colleague
1. can explain what Maven is.
2. can describe the 3 build lifecycles of Maven.
3. can explain that each build lifecycle is made up of phases.
4. can explain that each build phase is made up of plugin goals.
5. can identify the project coordinates.
6. can describe the Maven Standard Directory Layout.
7. can explain what a dependency is.
8. can describe the different dependency scopes.
9. can explain what a transitive dependency is.
10. can explain what a dependency tree is.
11. can execute maven commands from CLI.
12. can manage dependencies with Maven.
13. can create a REST endpoint that provides an image or a file.
14. can run code analysis on a Spring boot project (with SonarQube).
15. can fix bugs, vulnerabilities and code smells detected by SonarQube.

3.1 What is Maven?

Maven is a tool that can be used for building and managing Java-based projects. Every Java project requires certain dependencies, which are automatically downloaded when using Maven.

The main objectives of using Maven for developers are:

- Making the build process easy
- Providing a uniform build system
- Providing information about project quality (for example unit test reports)
- Encouraging better development practices

Maven is much more than a build tool. Maven offers support for automatic source generation, compiling the source code and test sources, packaging final product(s) for different environments, running health checks and reporting. Continuous builds, integration, and

testing can be easily handled by using Maven. A tool commonly used for creating CI/CD pipelines (continuous integration and continuous delivery) is Jenkins. Jenkins cannot replace Maven or vice-versa. Jenkins can use Maven as its build tool. When Jenkins deploys artifacts to remote repositories, they are usually Maven repositories.

According to snyk's 2021 JVM ecosystem report [1] Maven is most popular build tool for the Java ecosystem.

3.2 Installation and Configuration

IntelliJ IDEA has in-built support for Maven. However we want to make Maven available in the command-line interface. Therefore we need to install Maven.

Maven download and documentation pages:
Download page: https://maven.apache.org/download.cgi
Installation instructions: https://maven.apache.org/install.html

Confirm that Maven is installed correctly by executing `$mvn -v` in a terminal window.

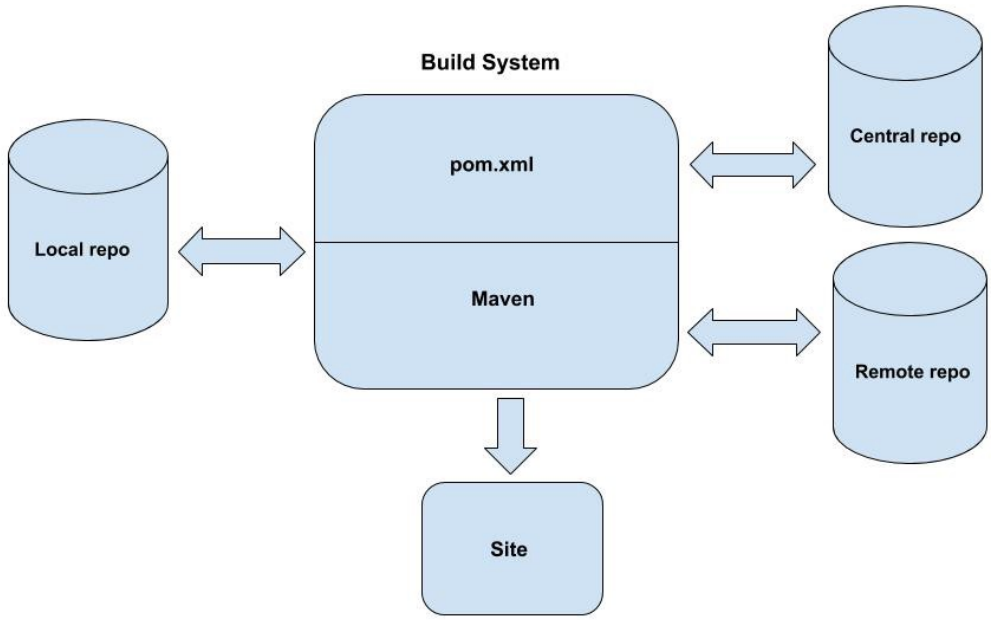
```
$ mvn -v
Apache Maven 3.8.2 (ea98e05a04480131370aa0c110b8c54cf726c06f)
Maven home: /usr/local/Cellar/maven/3.8.2/libexec
Java version: 17.0.1, vendor: Oracle Corporation, runtime:
    ↪ /Library/Java/JavaVirtualMachines/jdk-17.0.1.jdk/Contents/Home
Default locale: nl_BE, platform encoding: UTF-8
OS name: "mac os x", version: "11.5.2", arch: "x86_64", family: "mac"
```

3.3 Maven Standard Directory Layout

When you create a new Maven project, a directory which name matches the artifactId is generated. This directory is known as the project's base directory. Every Maven project has a file named **pom.xml** which is known as the **Project Object Model** (POM). This file describes the project, configures plugins, and declares dependencies. The project's source code and resources are placed in the folder **src/main**. The project's unit tests are located in **src/test**. The target folder is the Maven default output folder. You can delete all the target's folder content with the `$mvn clean` command.

[1] https://snyk.io/jvm-ecosystem-report-2021/

3.4 Maven architecture



Central repository is provided by the Maven community. This Central repository contains a large number of common libraries. Whenever you specify a dependency in pom.xml, Maven will look for it in the local repository first. If the necessary dependency isn't included in the local repository, Maven will download it from the Central repository and copy it to your local repository. Remote repository is a place where developers (or Jenkins) can copy the final packages, so other developer can use these final packages as a dependency in their projects.

Acces to internet is recommended when using Maven, however it is possible to work offline if all dependencies are available in your local repository.

3.5 Maven built-in life cycles

There are three built-in life cycles defined in Maven. The *default* build life cycle is the main build life cycle.

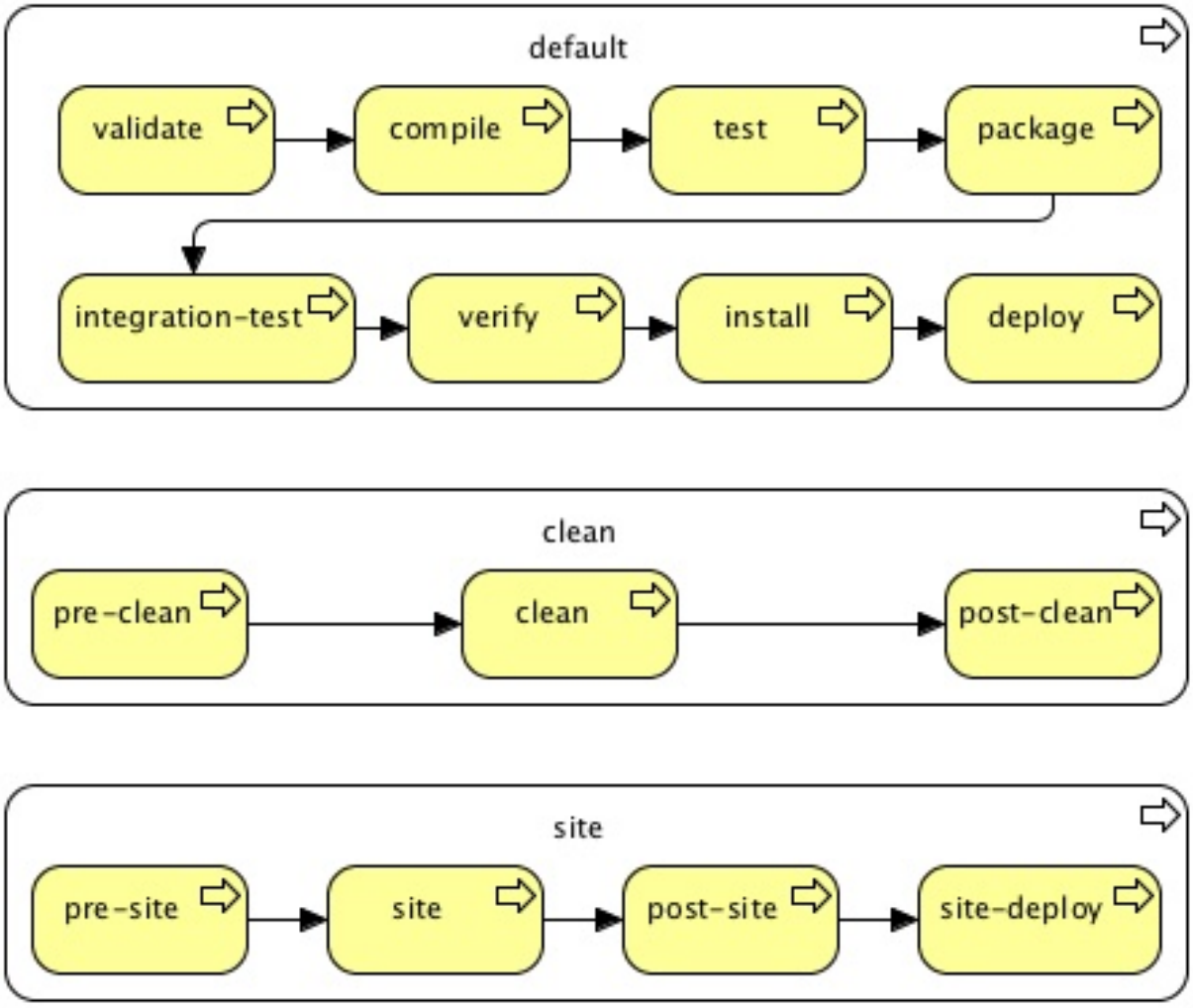
clean	handles the cleanup of directories and files generated during the build process	mvn clean
default	handles the build and distribution of the project	mvn [plugin:goal]* [phase]*
site	create project documentation	mvn site

To run a specific goal, without executing the entire phase (and the preceding phases) the command $mvn [plugin:goal] can be used. Phases on the other hand are executed in a

specific order and all the preceding phases are executed as well!

3.6 A Build Lifecycle is Made Up Of Phases

Each life cycle consists of a sequence of phases. The default build lifecycle consists of 23 phases. In the image below the 8 main phases are included. On the other hand, clean lifecycle consists of 3 phases, while the site lifecycle is made up of 4 phases.

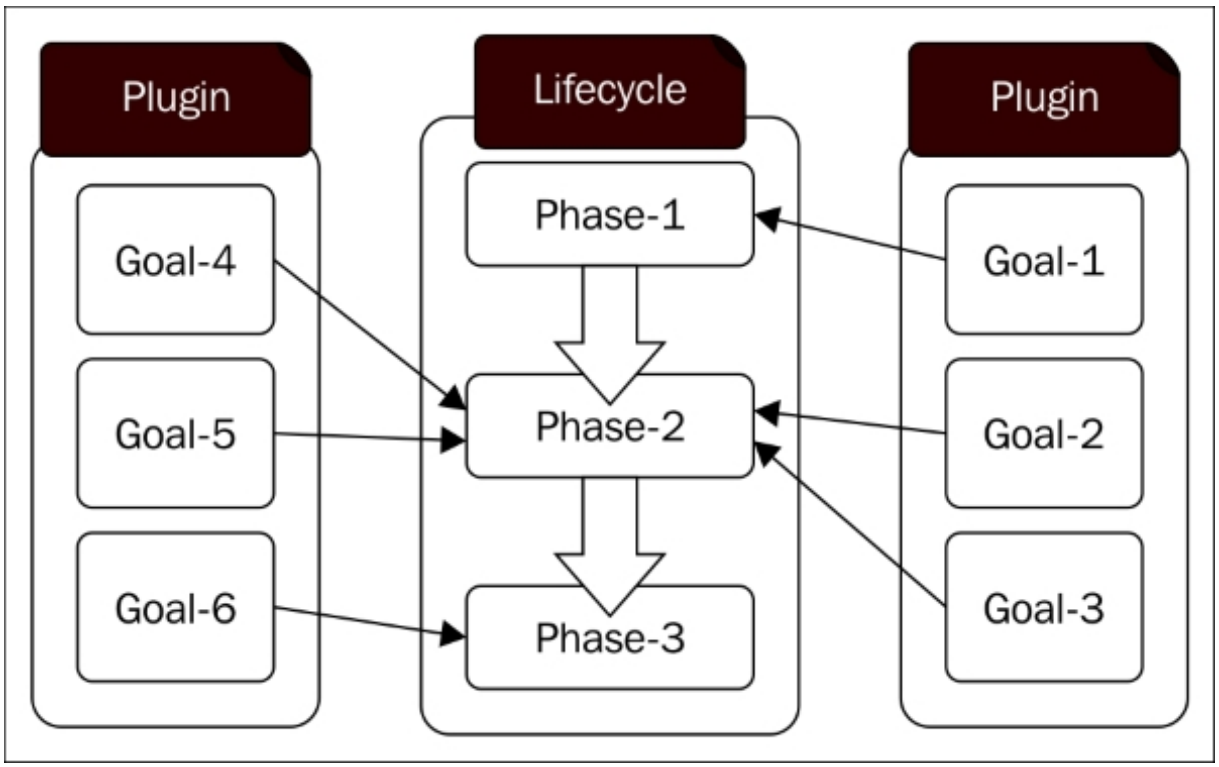


Each phase is responsible for a specific task. Here are the 8 most important phases in the default build lifecycle.

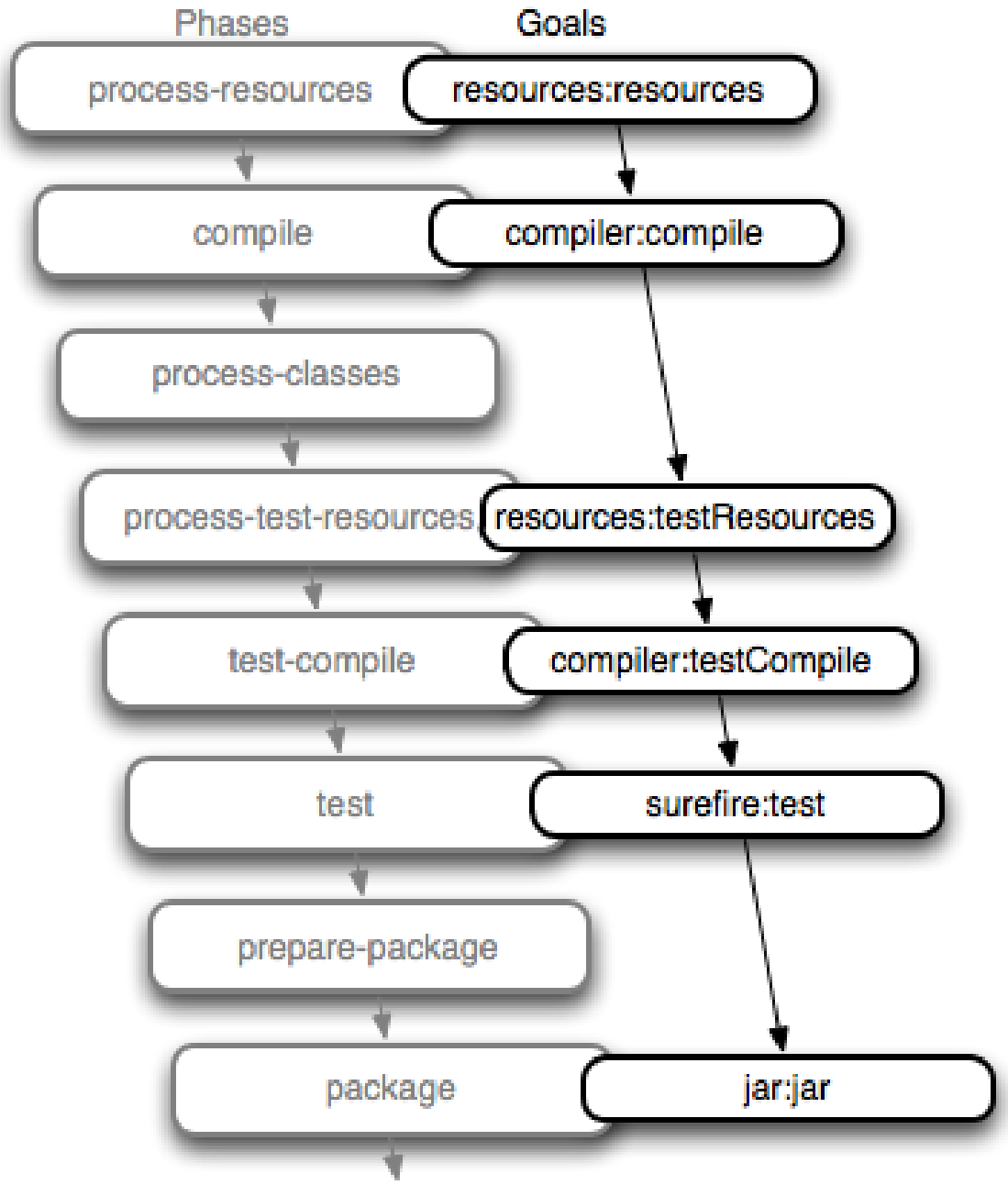
validate	check if all information necessary for the build is available
compile	compile the source code
test	run unit tests
package	package compiled source code into the distributable format, (jar, war, ...)
integration-test	process and deploy the package if needed to run integration tests
verify	run any checks on results of integration tests to ensure quality criteria are met
install	install the package to a local repository
deploy	copy the package to the remote repository

3.7 Plugins and Goals

Maven is actually a plugin execution framework. Every task executed by Maven is actually done by a **goal**, where goals are grouped together in **plugins**. When we run a phase, all the goals bound to the phase are executed in order.



Several phases of the default built-in lifecycles have goals bounded to them. Due to this default configuration you're able to build a Java project without extra configuration.



Note: There are more phases than shown above, this is a partial list

The compile goal from the compiler plugin is bound to the compile phase and is responsible for compiling the source code. The test goal from the surefire plugin is bound to the test phase and is reponsible for running the unit tests.

You can generate an overview of all the phases and the specific goals bounded to these phases by running the command `$mvn help:describe -Dcmd=PHASENAME`

```
$ cd superhero-backend/
$ mvn help:describe -Dcmd=compile
[INFO] Scanning for projects...
[INFO]
```

```
[INFO] ----------------------< be.pxl:superhero-backend
   ↪ >----------------------
[INFO] Building superhero-backend 0.0.1-SNAPSHOT
[INFO] --------------------------------[ jar
   ↪ ]---------------------------------
[INFO]
[INFO] --- maven-help-plugin:3.3.0:describe (default-cli) @
   ↪ superhero-backend ---
[INFO] 'compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile

It is a part of the lifecycle for the POM packaging 'jar'. This lifecycle
   ↪ includes the following phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources:
   ↪ org.apache.maven.plugins:maven-resources-plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources:
   ↪ org.apache.maven.plugins:maven-resources-plugin:2.6:testResources
* test-compile:
   ↪ org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-jar-plugin:2.4:jar
* pre-integration-test: Not defined
* integration-test: Not defined
* post-integration-test: Not defined
* verify: Not defined
* install: org.apache.maven.plugins:maven-install-plugin:2.4:install
* deploy: org.apache.maven.plugins:maven-deploy-plugin:2.7:deploy

[INFO]
   ↪ ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
   ↪ ------------------------------------------------------------------------
[INFO] Total time: 0.896 s
[INFO] Finished at: 2023-02-26T17:19:03+01:00
[INFO]
```

```
↪ ------------------------------------------------------------------------
```

There are two types of plugins:

- **Build plugins**: The goals of build plugins are executed during the build process. If you want to execute additional goals during a phase of the build process, you need to configure the goal in the `<build>` element of the pom.xml.
- **Reporting plugins**: Reporting plugins are executed during the site generation process. These plugins are configured in the `<reporting>` element of the pom.xml.

3.8 Dependencies and scopes

A repository in Maven stores artifacts and dependencies of varying types. There are exactly two types of repositories: local and remote.

The **local repository** is a directory on the machine that runs Maven. By default Maven's local repository is located in the folder $user.home/.m2/repository. As you can see, this is a hidden folder. If you're unable to find the default .m2 folder, you can run the following command: `$mvn help:evaluate -Dexpression=settings.localRepository`

Remote repositories refer to any other type of repository, accessed by a variety of protocols such as file:// and https://.

Exercise 3.1. Locate and open the folder containing your local repository.

If you're looking for a dependency to add, you can use the Maven Central Repository Search [2].

There are two types of dependencies in Maven: direct and transitive. **Direct dependencies** are the dependencies that are defined in the pom.xml under the `<dependencies>` section. **Transitive dependencies** are dependencies of your direct dependencies. This means if your project needs dependency A and A depends on B, then your project needs both A and B. However, for Maven it suffices to add dependency A in the pom.xml. The transitive dependency B is included automatically.

You can generate the dependency tree with direct and transitive dependencies by running `$mvn dependency:tree`.

Sometimes, transitivity brings a very serious problem causing version mismatch issues at runtime. when multiple versions of the same artifact are encountered, Maven picks the "nearest definition". It uses the version of the closest dependency in the dependency tree.

```
A
\-- B
   \-- C
      \-- D 2.0
\-- E
```

[2] https://search.maven.org

```
    \-- D 1.0
```

D 1.0 will be used when building project A because the path from A to D through E is shorter. You can explicitly add a dependency to D 2.0 in A to force the use of D 2.0.

```
A
 \-- B
    \-- C
       \-- D 2.0
 \-- E
    \-- D 1.0
 \-- D 2.0
```

There are 6 dependency scopes which are used to limit the transitivity of dependencies and determine when a dependency should be included in the classpath.

compile	This is the default scope. Depencencies with this scope are available on the classpath for all the build tasks.
provided	Dependencies that are provided at runtime by JDK or a container. The provided dependencies are available at compile-time and in the test classpath.
runtime	The dependencies with this scope are only required at runtime. They are not needed at compile-time and in the test classpath.
test	These dependencies are only needed for executing tests.
system	Similar to provided but specific jar is provided.
import	All dependencies listed in another pom are included.

3.9 Exercise

In the exercise we implement a REST endpoint that returns a PDF file.

```
@GetMapping(value = "/pdfreport", produces =
    ↪ MediaType.APPLICATION_PDF_VALUE)
public @ResponseBody byte[] createPdfReport() {

    ByteArrayInputStream bis = GeneratePdfReport.createReport();

    return bis.readAllBytes();
}
```

In the example above we assume to have a helper class GeneratePdfReport that has a method returning a ByteArrayInputStream with the data of your PDF file. We use the @ResponseBody annotation on the controller method to indicate that the object returned by the method should be marshaled directly to the HTTP response body.

For creating PDF files in Java, we'll use itextpdf: https://mvnrepository.com/artifact/com.itextpdf/itextpdf.

For creating the qr code we will use the ZXing ('Zebra Crossing') API, a popular API for QR code processing in Java. You'll need to include 2 artifacts to be able to generate QR

codes.

```
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>core</artifactId>
    <version>3.4.0</version>
</dependency>
<dependency>
    <groupId>com.google.zxing</groupId>
    <artifactId>javase</artifactId>
    <version>3.4.0</version>
</dependency>
```

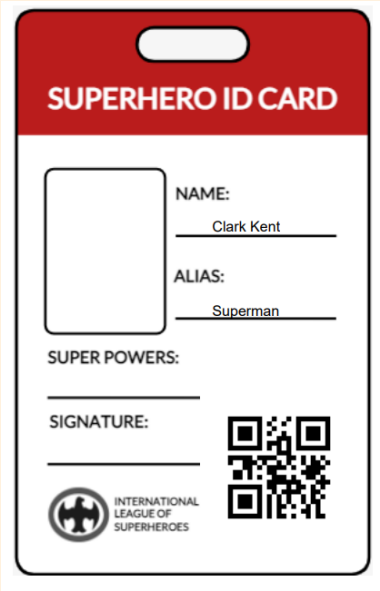
Here is the full code for filling the name and QR code in the PDF template.

```
package be.pxl.superhero.service.impl;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.commons.Cipher;
import com.google.zxing.BarcodeFormat;
import com.google.zxing.WriterException;
import com.google.zxing.client.j2se.MatrixToImageWriter;
import com.google.zxing.common.BitMatrix;
import com.google.zxing.qrcode.QRCodeWriter;
import com.itextpdf.text.DocumentException;
import com.itextpdf.text.Element;
import com.itextpdf.text.Image;
import com.itextpdf.text.Phrase;
import com.itextpdf.text.pdf.ColumnText;
import com.itextpdf.text.pdf.PdfContentByte;
import com.itextpdf.text.pdf.PdfReader;
import com.itextpdf.text.pdf.PdfStamper;
import org.springframework.stereotype.Component;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.net.URISyntaxException;
import java.nio.file.Path;
import java.nio.file.Paths;

@Component
public class SuperheroIdCardGenerator {

    public ByteArrayInputStream superheroIdCard(SuperheroDTO superhero) {

        ByteArrayOutputStream out = new ByteArrayOutputStream();
```

```
        try {
            Path path =
                Paths.get(ClassLoader.getSystemResource("superheroidcard.pdf").toURI());
            PdfReader pdfReader = new PdfReader(path.toUri().toURL());
            PdfStamper pdfStamper = new PdfStamper(pdfReader, out);
            PdfContentByte canvas = pdfStamper.getOverContent(1);
            ColumnText.showTextAligned(canvas, Element.ALIGN_LEFT, new
                ↪ Phrase(superhero.firstName() + " " +
                ↪ superhero.lastName()), 200, 620, 0);
            ColumnText.showTextAligned(canvas, Element.ALIGN_LEFT, new
                ↪ Phrase(superhero.superheroName()), 200, 550, 0);
            byte[] qrCodeImage =
                ↪ getQRCodeImage(Cipher.skipALetter(superhero.superheroName()),
                ↪ 130, 130);
            Image qrCode = Image.getInstance(qrCodeImage);
            qrCode.setAbsolutePosition(190, 360);
            canvas.addImage(qrCode);
            pdfStamper.close();
            pdfReader.close();
        } catch (DocumentException | URISyntaxException | IOException |
            ↪ WriterException ex) {
            // TODO: add logging - see next chapter
            throw new PdfCreationException(ex);
        }
        return new ByteArrayInputStream(out.toByteArray());
    }

    public static byte[] getQRCodeImage(String text, int width, int height)
        ↪ throws WriterException, IOException {
        QRCodeWriter qrCodeWriter = new QRCodeWriter();
        BitMatrix bitMatrix = qrCodeWriter.encode(text,
            ↪ BarcodeFormat.QR_CODE, width, height);

        ByteArrayOutputStream pngOutputStream = new ByteArrayOutputStream();

        MatrixToImageWriter.writeToStream(bitMatrix, "PNG",
            ↪ pngOutputStream);
        return pngOutputStream.toByteArray();
    }
}
```

The PdfReader first opens the PDF template file. The PdfStamper is used for adding the full name, the alias and the QR code (as an image) to the template. The newly generated PDF is returned as a ByteArrayInputStream.

Exercise 3.2. All superheroes need an id card. Your job as a developer is to create a REST endpoint where we can download a pdf with the id card.



The pdf template can be found in blackboard.
☐ Download the pdf template and add it to the resources directory of your superhero-backend project.

We will include the full name, the superhero name and a unique QR code in the template. The generated pdf will be available at the endpoint: `http://localhost:<port>/api/superheroes/<superheroId>/idcard`
The QR code will contain the scrambled alias of a superhero.

☐ Create the utility class Cipher with a static method skipALetter which takes a String as a parameter and returns the scrambled String. To encrypt a sentence you divide each word into half. If a word has an odd number of letters, the first group of letters contains one letter extra. Take the first letter of the first group, followed by the first letter of the second group. Then, write the second letter of the first group and the second letter of the second group, and so on, until all the words are encrypted. For example, if your sentence is 'SECRET CODES', it will be encrypted as 'SREECT CEOSD'.
Write unit tests to test your implementation.

- unit test for one word with even length
- unit test for one word with odd length
- unit test for a sentence with multiple words with odd and even length
- unit test for an empty string (should return an empty string)

☐ Implement the REST endpoint for retrieving a superhero's id card.

3.10 Code quality with SonarQube

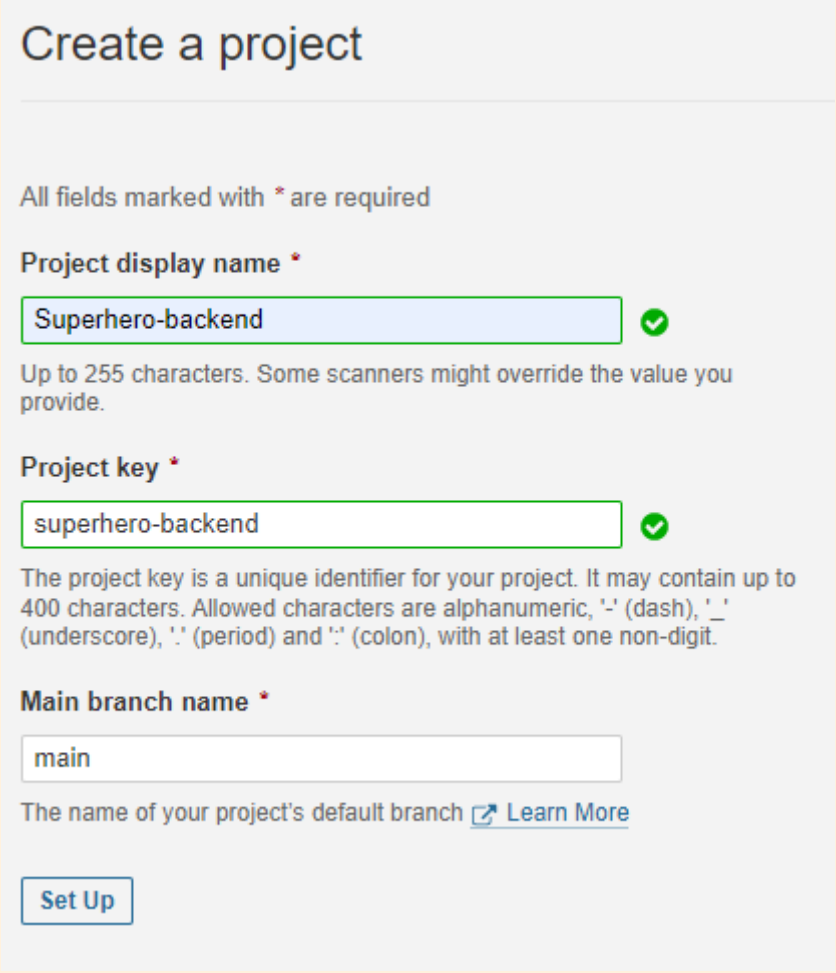
We would like to deliver high quality Java code. SonarQube is a Java analyzer that checks the quality of our code and detects code smells.

Exercise 3.3. Create a docker container with SonarQube. Run the following command in a terminal window:

```
docker run -d --name sonarqube -p 9000:9000 -p 9092:9092 sonarqube
```

Open http://localhost:9000 in a browser. You can login with the default username 'admin' and password 'admin'. Next you have to update the default password (you have to choose a new password!).

Create a new SonarQube project for the superhero-backend application. Click on the button 'Manually', fill out the project details and choose to analyze your project locally. A token will be generated.



Now we can update the pom.xml file of our superhero-backend to run the code analysis.
First we need a sonar profile to provide the url, project key and token.

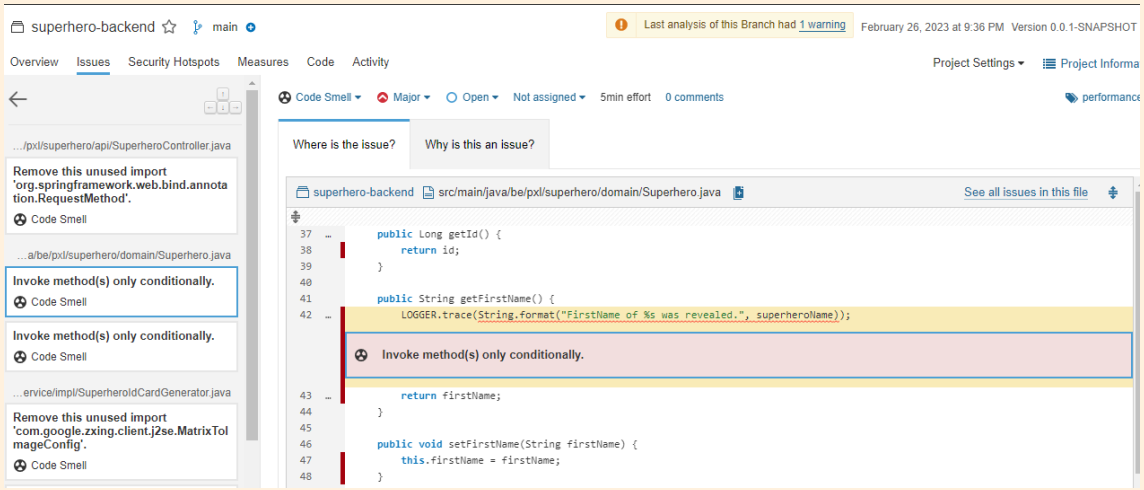
```
<profiles>
    <profile>
        <id>sonar</id>
        <activation>
            <activeByDefault>true</activeByDefault>
        </activation>
        <properties>
            <sonar.host.url>http://localhost:9000</sonar.host.url>
```

```
            <sonar.projectKey>superhero-backend</sonar.projectKey>
            <sonar.login>sqp_6319ae2795ccdc24baa3a994962a8bbaa0b0cc16</sonar.login>
        </properties>
    </profile>
</profiles>
```

Run the following maven command to perform the code analysis:

```
$mvn clean verify sonar:sonar
```

Open the SonarQube dashboard http://localhost:9000/dashboard?id=superhero-backend to get an overview of possible bugs, vulnerabilities and code smells. You can get a more in-depth explanation of the code quality rules including examples at https://rules.sonarsource.com/java. Namely, https://rules.sonarsource.com/java/RSPEC-5786 is an interesting code smell to look at.



When you look at the dashboard, you'll see the code coverage is always 0.0%. JaCoCo is needed to generate an extra report to display the code coverage.
First add the property jacoco.version in the pom.xml file:

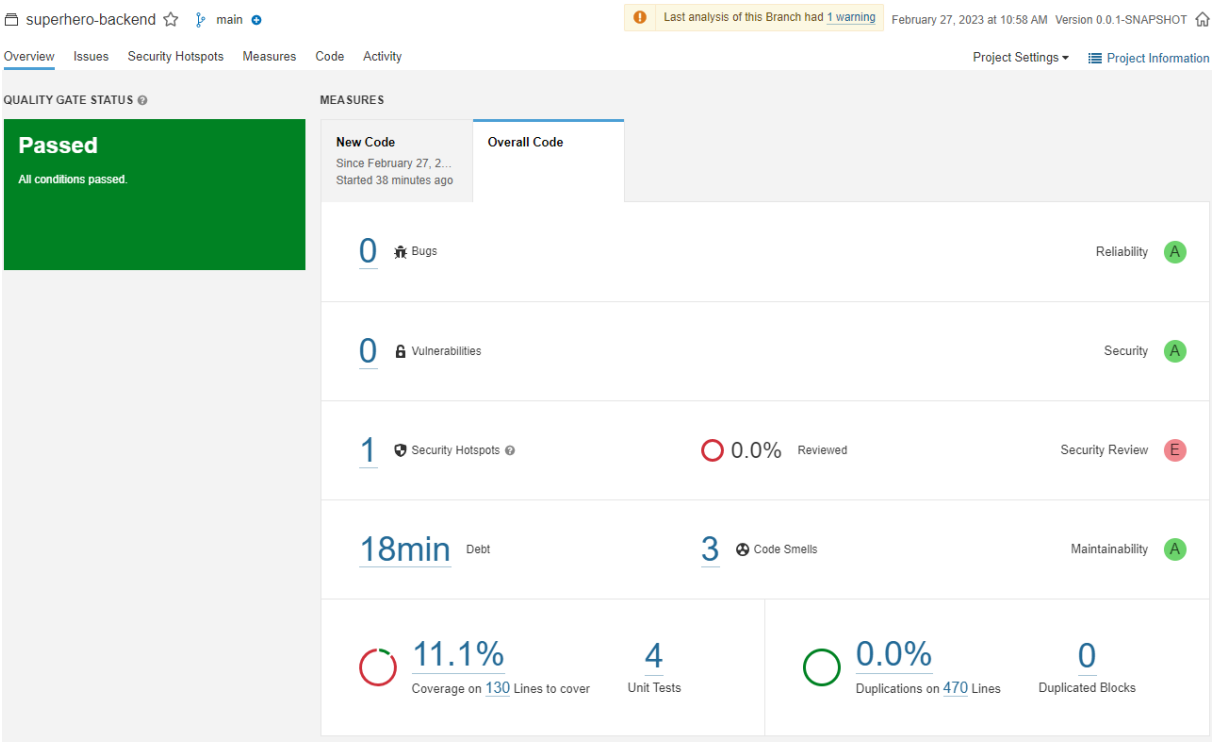
```
<properties>
    <java.version>17</java.version>
    <jacoco.version>0.8.7</jacoco.version>
</properties>
```

Next, add the JaCoCo maven plugin to your project.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>${jacoco.version}</version>
            <executions>
                <execution>
                    <id>prepare-agent</id>
```

```
                <goals>
                    <goal>prepare-agent</goal>
                </goals>
            </execution>
            <execution>
                <id>report</id>
                <phase>test</phase>
                <goals>
                    <goal>report</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
<pluginManagement>
    <plugins>
        ...
    </plugins>
</pluginManagement>
</build>
```

During the test phase JaCoCo will generate a report and save it in the directory target/site/jacoco/jacoco.xml. SonarQube will automatically check this location and the report will be picked up. First remove the empty, automatically created Spring Boot test from your test folder. Run the Maven command again and check your code coverage.



3.11 Maven commands

Here is a brief list of usefull Maven commands.

Maven Command	Description
mvn -v	Prints out the version of Maven you are running.
mvn –version	Same as mvn -v
mvn clean	Clears the target directory into which Maven normally builds your project.
mvn package	Builds the project and packages the resulting JAR file into the target directory.
mvn package -DskipTests	Builds the project and packages the resulting JAR file into the target directory without running the unit tests during the build. You can also use -Dmaven.test.skip=true
mvn clean package	Clears the target directory, builds the project and packages the resulting JAR file into the target directory.
mvn install	Builds the project described by your Maven POM file and installs the resulting artifact (JAR) into your local Maven repository.
mvn -X package	Prints the maven version and runs the build in the debug mode.
mvn -o package	This command is used to run the maven build in the offline mode.
mvn -help	Prints the Maven usage and all the available options.
mvn dependency:tree	Generates the dependency tree of the Maven project.

Chapter 4

Logging

Learning goals
The junior-colleague
1. can explain why a logging framework is used.
2. can configure and use a log4j2 in a Spring boot application.
3. can apply good practices when using a logging framework.

4.1 Logging Framework

A logging framework is a utility specifically designed to standardize the process of logging in your application. Logging is very important for debugging and identifying performance hot spots in an application, as well as getting a sense of how your application operates. The following recommendations about logging can be find in OWASP Logging Guide [1]:

- Why log?
  - identify security incidents
  - identify fraudulent activity
  - identify operational and longterm problems
  - ensure compliance with laws,rules and regulations
- What is commonly logged ?
  - Client requests and server responses
  - Account activities (login, logout, change password etc.)
  - Usage information (transaction types and sizes, generated traffic etc.)
  - Significant operational actions such as application startup and shutdown, application failures, and major application configuration changes. This can be used to identify security compromises and operational failures.

[1] https://owasp.org/www-pdf-archive/OWASP_Logging_Guide.pdf

Much of this info can only be logged by the applications themselves. This is especially true for applications used through encrypted network communications. Therefore we need a standardized process of logging where log-files can be archived easily.

4.2 Log levels

Level	Description
FATAL	the log level that tells that the application encountered an event or entered a state in which one of the crucial business functionality is no longer working.
ERROR	the log level that should be used when the application hits an issue preventing one or more functionalities from properly functioning.
WARN	the WARN level should be used in situations that are unexpected, but the code can continue the work.
INFO	information logged using the INFO log level should be purely informative and not looking into them on a regular basis shouldn't result in missing any important information
DEBUG	should be used for information that may be needed for diagnosing issues and troubleshooting or when running application in the test environment for the purpose of making sure everything is running correctly
TRACE	the most fine-grained information only used in rare cases where you need the full visibility of what is happening in your application

4.3 Logging in Spring Boot

Spring Boot uses Apache Commons logging for all internal logging [2]. Commons logging is a bridge to the logging implementation of your choice. You can choose a logging system you like: log4j2, SLF4J, LogBack, etc.

By default, if you use the 'Starters', Logback is used for logging. It is pre-configured to use console output.

Add the following controller to an existing Spring Boot application and trigger the logging lines by visiting http://localhost:{port}/loglevels.

```
package be.pxl.demo.api;

@RestController
public class LoggingController {

    Logger logger = LoggerFactory.getLogger(LoggingController.class);

    @RequestMapping("/loglevels")
    public String demoLogLevels() {
```

[2] https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.logging

```
        logger.trace("A TRACE Message");
        logger.debug("A DEBUG Message");
        logger.info("An INFO Message");
        logger.warn("A WARN Message");
        logger.error("An ERROR Message");

        return "Howdy! Check out the Logs to see the output...";
    }
}
```

The default logging level of the Logger is preset to INFO, meaning that TRACE and DEBUG messages are not visible.

In the file application.properties you can change the log level by using logging.level.<logger-name>=<level>.

4.4 Log4j2 Configuration Logging

In a production environment, all your logging should be written in files. A decent rolling file policy is necessary to avoid huge log files.

In this course we will use log4j2. End 2021 a critical vulnerability was found in log4j which affected millions of systems [3]. Make sure you use the latest version in your applications!

To start using log4j2 in your Spring boot application, you update the dependencies in pom.xml. Exclude the default logging and add the starter for log4j2.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

In the resources folder you create the file log4j2-spring.xml or log4j2.xml. This is the log4j2 configuration file. Here is an example of a configuration file which can be found at https://www.baeldung.com/spring-boot-logging.

[3] https://cisomag.eccouncil.org/log4j-explained

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout
                pattern="%style{%d{ISO8601}}{black} %highlight{%-5level
                    ↪ }[%style{%t}{bright,blue}]
                    ↪ %style{%C{1.}}{bright,yellow}: %msg%n%throwable" />
        </Console>

        <RollingFile name="RollingFile"
            fileName="./logs/spring-boot-logger-log4j2.log"
            filePattern="./logs/$${date:yyyy-MM}/spring-boot-logger-log4j2-%d{-dd-MMMM-yy
            <PatternLayout>
                <pattern>%d %p %C{1.} [%t] %m%n</pattern>
            </PatternLayout>
            <Policies>
                <!-- rollover on startup, daily and when the file reaches
                    10 MegaBytes -->
                <OnStartupTriggeringPolicy />
                <SizeBasedTriggeringPolicy
                    size="10 MB" />
                <TimeBasedTriggeringPolicy />
            </Policies>
        </RollingFile>
    </Appenders>

    <Loggers>
        <!-- LOG everything at INFO level -->
        <Root level="info">
            <AppenderRef ref="Console" />
            <AppenderRef ref="RollingFile" />
        </Root>

        <!-- LOG "be.pxl.superhero*" at TRACE level -->
        <Logger name="be.pxl.superhero" level="trace"></Logger>
    </Loggers>

</Configuration>
```

The logging level used in the tag <Configuration> is for log4j2 internal events. We created 2 Appenders, one Appender uses the console, the other Appender uses a file. The description of the logging patterns can be found in the Log4j2 documentation (https://logging.apache.org/log4j/2.x/manual/layouts.html).

```
package be.pxl.superhero.domain;
```

```java
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;


@Entity
@Table(name="superheroes")
public class Superhero {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(Superhero.class);

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    private String superheroName;

    public Superhero() {
        // JPA only
    }

    public Superhero(String firstName, String lastName, String
        ↪ superheroName) {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("Creating a new superhero...");
        }
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public Long getId() {
        return id;
    }

    public String getFirstName() {
        if (LOGGER.isTraceEnabled()) {
            LOGGER.trace(String.format("FirstName of %s was revealed.",
                ↪ superheroName));
        }
        return firstName;
```

```
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        if (LOGGER.isFatalEnabled()) {
            LOGGER.fatal(String.format("LastName of %s was revealed.",
                ↪ superheroName));
        }
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    public void setSuperheroName(String superheroName) {
        this.superheroName = superheroName;
    }

    @Override
    public String toString() {
        return superheroName;
    }
}
```

Always declare a static Logger reference, the creation of a Logger comes with an overhead. It is good practice to invoke logging conditionally. This will minimize the impact of the log messages (it will save for example potential String concatenations).

Exercise 4.1. ☐ Add log4j2 to your superhero-backend project.
- ☐ Add the log4j2 configuration file from the textbook to your application.
- ☐ Add logging for the class Superhero and the utility class SuperheroIdCard-Generator.
- ☐ Add a log message with level debug in the main-method of the spring boot application.
- ☐ Run the application and look into the log files. Change log levels in the configuration file and run the application again.

Chapter 5

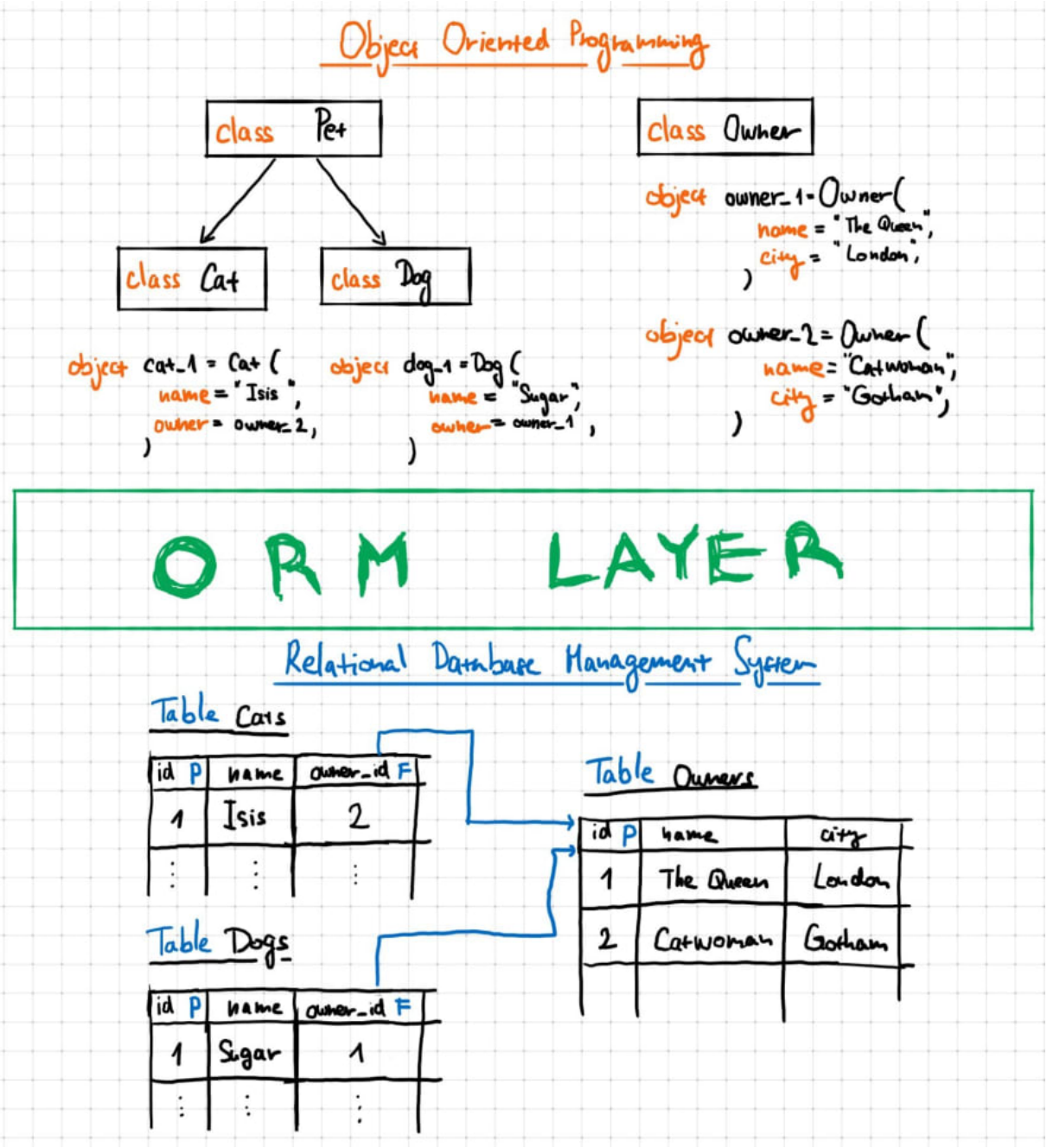
Spring Data JPA

Learning goals
The junior-colleague

1. can describe the concept of ORM.
2. can explain what JPA is, and what it's not.
3. can denominate different JPA providers.
4. can describe what a persistence unit is.
5. can explain the fundamental interfaces of JPA.
6. can explain what the persistence context is.
7. can implement entity classes.
8. can describe the entity objects' lifecycle.
9. can implement different types of relationships between entity classes.
10. can implement CRUD operations.
11. can implement queries.
12. can implement named queries.
13. can explain, identify and solve the N + 1 query problem.

5.1 What is ORM?

Object-Relational Mapping (ORM) is a technique that lets you query and manipulate data from a relational database using an object-oriented programming language.



5.2 What is JPA?

The Java Persistence API (JPA; recently renamed to Jakarta Persistence API) is a specification that defines how to persist data in Java applications. JPA mainly focuses on the ORM layer and managing persistent objects.

JPA is a specification which means JPA consists of implementation guidelines for the Java ORM layer and the syntax. The specification only comes with interfaces, no actual implementation. A reference implementation can be provided but other companies can create and distribute their own implementation of the specification.

2.1. The Entity Class

The entity class must be annotated with the *Entity* annotation or denoted in the XML descriptor as an entity.

The entity class must have a no-arg constructor. The entity class may have other constructors as well. The no-arg constructor must be public or protected.

The entity class must be a top-level class. An enum or interface must not be designated as an entity.

For our exercises and projects we will use Hibernate as the actual implementation of de JPA specification [1].

Spring Data JPA adds a layer on top of JPA. It uses all the features defined by the JPA specification, but adds no-code implementation of the repository pattern and the creation of database queries from method names.

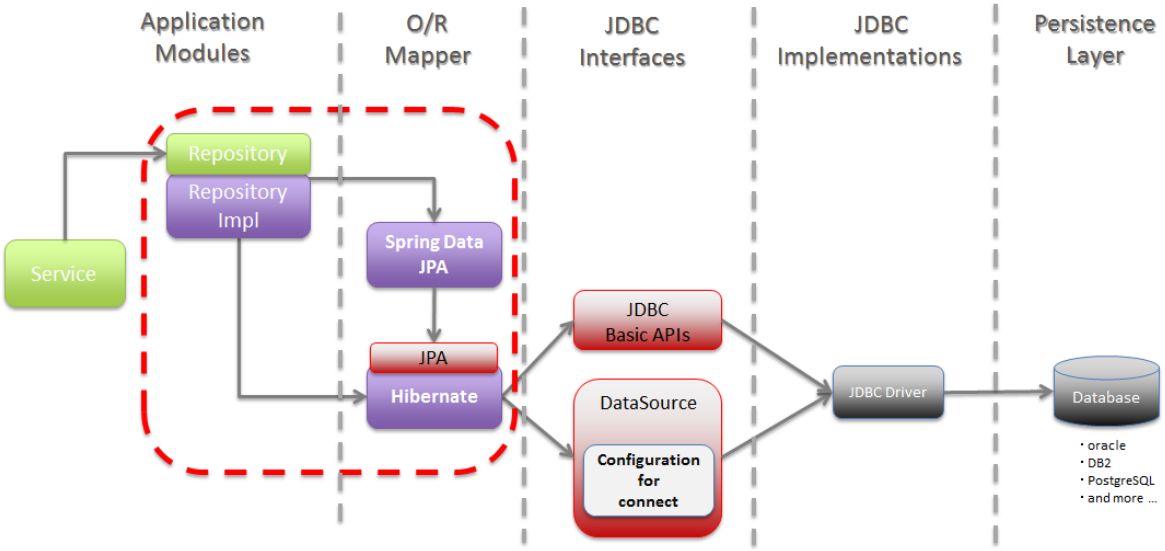
The JpaRepository interface is an extension of CrudRepository. JpaRepository extends PagingAndSortingRepository which in turn extends CrudRepository.

Their main functions are:

- CrudRepository mainly provides CRUD functions.
- PagingAndSortingRepository provides methods to do pagination and sorting records.
- JpaRepository provides some JPA-related methods such as flushing the persistence context and deleting records in a batch.

5.3 Datasource

In Spring Boot a DataSource-object is the preferred means of getting a connection to a database. The interface jakarta.sql.DataSource is implemented by the database driver vendor.



[1] https://hibernate.org/andhttps://hibernate.org/orm/

The datasource can be specified in the application.properties file. Here are some common database properties:

spring.datasource.url	JDBC URL of the database.
spring.datasource.username	Login username of the database.
spring.datasource.password	Login password of the database.
spring.jpa.show-sql	Whether to enable logging of SQL statements. Default: false
spring.jpa.hibernate.ddl-auto	Possible values: none (production), create, create-drop, validate and update. [2]

Alternatively, the data source configuration can be programmatically.

The H2 in-memory database will be replaced with a MySQL database. We will run the MySQL database in a docker container. Each database server has its own specific protocol for transporting requests to, and results from, the server to application programs. Therefore, you need to replace the h2 dependency with the MySQL driver dependency.

```
<dependency>
      <groupId>com.mysql</groupId>
      <artifactId>mysql-connector-j</artifactId>
      <scope>runtime</scope>
</dependency>
```

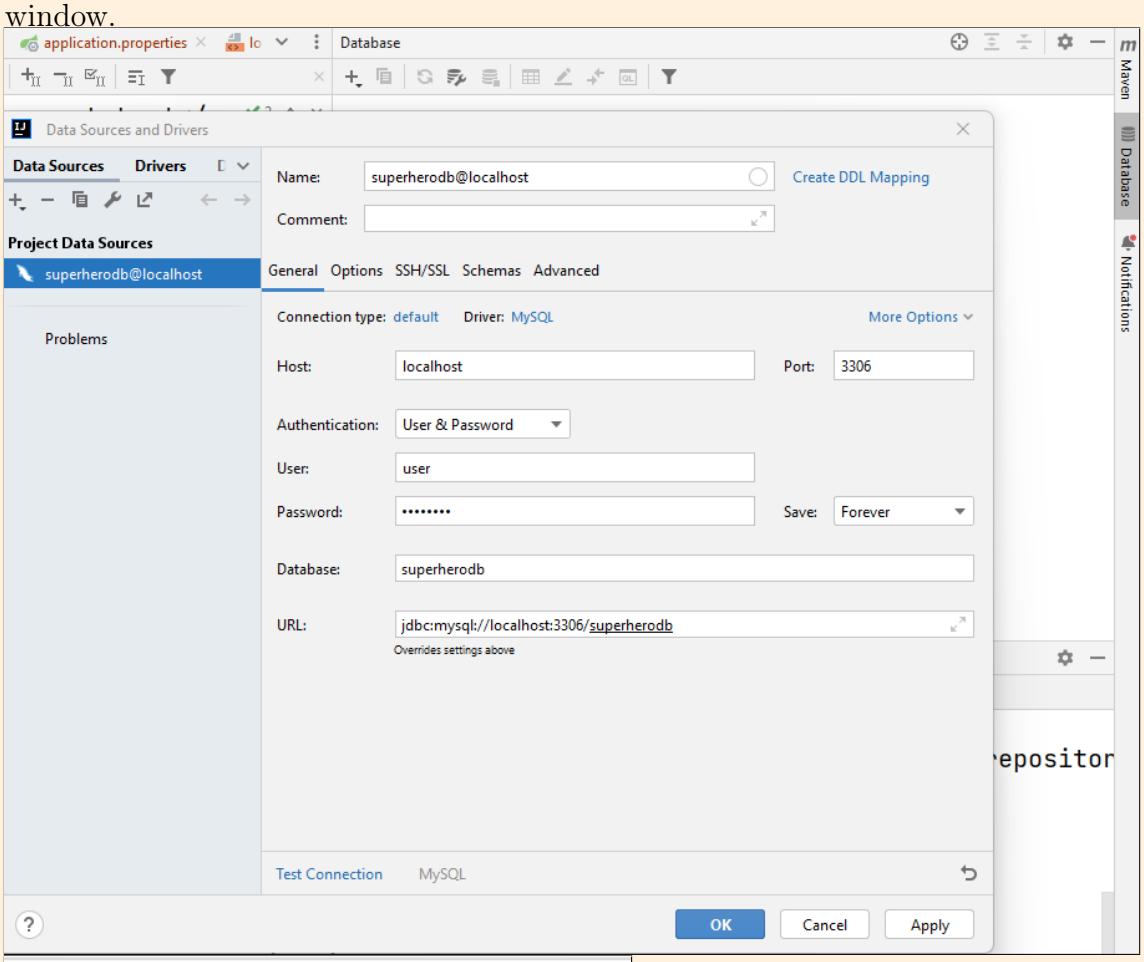
Here is the docker-compose.yml file for creating the database for local development.

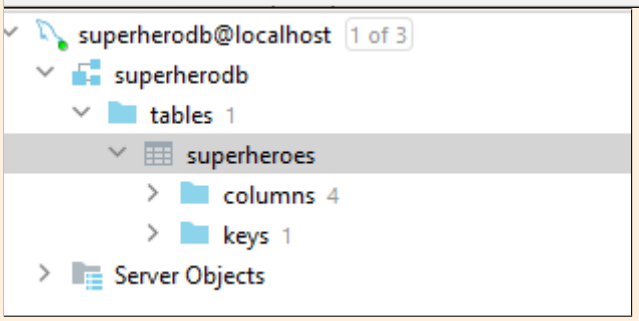
```
version: "3.3"

services:
  superhero-db:
    image: mysql:latest
    ports:
      - "3306:3306"
    environment:
      MYSQL_DATABASE: 'superherodb'
      # So you don't have to use root, but you can if you like
      MYSQL_USER: 'user'
      # You can use whatever password you like
      MYSQL_PASSWORD: 'password'
      # Password for root access
      MYSQL_ROOT_PASSWORD: 'admin'
```

Exercise 5.1. Update the dependencies of the superhero-backend project. In the folder src/main you create a new directory docker. Here you add the docker-compose.yml file. In a terminal window, you navigate to the folder /src/main/docker and execute the command `docker-compose up`.
In the application.properties file you add the following lines:

```
spring.datasource.url=jdbc:mysql://localhost:3306/superherodb
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
```

Start the spring boot application. Add the datasource in intellij's Database tool window.





5.4 The Entity class

```
package be.pxl.superhero.domain;

import jakarta.persistence.*;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

```
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name="superheroes")
public class Superhero {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(Superhero.class);

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String superheroName;
    @Column(name="notes")
    private String description;

    public Superhero() {
        // JPA only
    }

    public Superhero(String firstName, String lastName, String
        ↪ superheroName) {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("Creating a new superhero...");
        }
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public Long getId() {
        return id;
    }

    public String getFirstName() {
        if (LOGGER.isTraceEnabled()) {
            LOGGER.trace(String.format("FirstName of %s was revealed.",
                ↪ superheroName));
        }
        return firstName;
    }
```

```
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        if (LOGGER.isFatalEnabled()) {
            LOGGER.fatal(String.format("LastName of %s was revealed.",
                ↪ superheroName));
        }
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    public void setSuperheroName(String superheroName) {
        this.superheroName = superheroName;
    }

    @Override
    public String toString() {
        return superheroName;
    }
}
```

A JPA entity class is a POJO (Plain Old Java Object) class that is annotated as having the ability to represent objects in the database.

The **@Id** annotation marks a field as a primary key field.

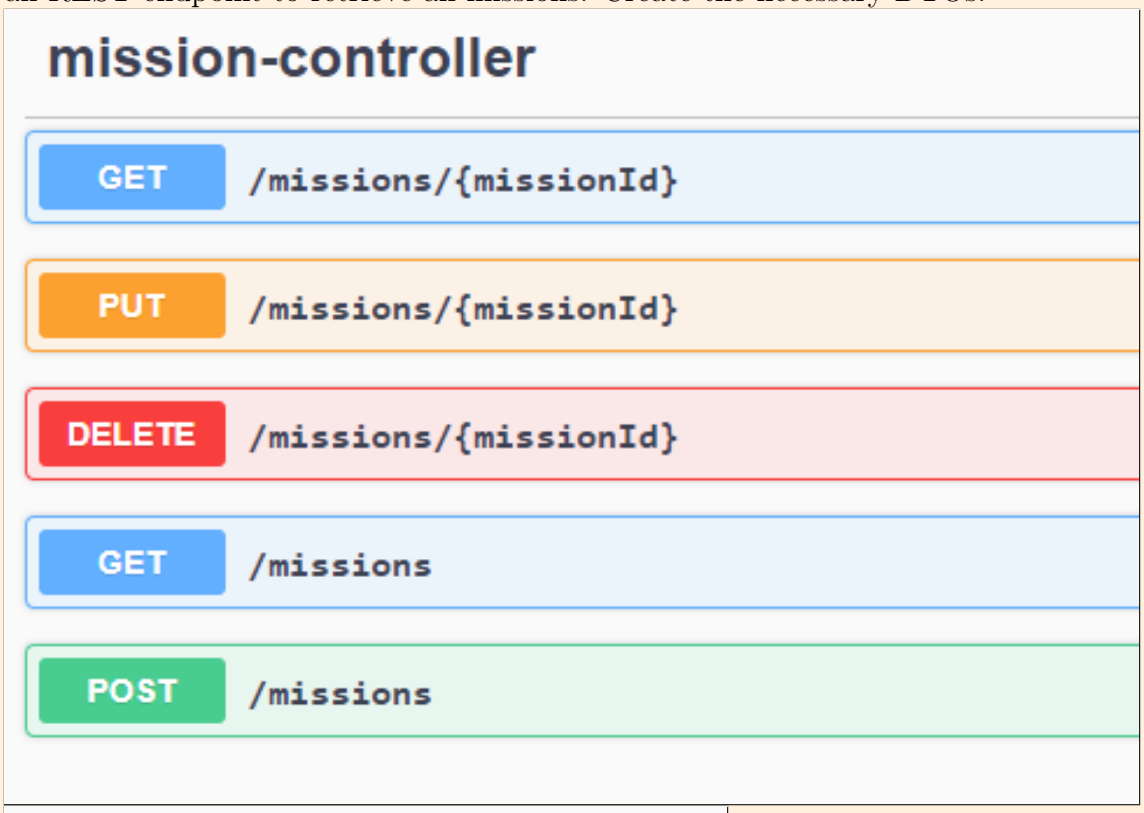
The **@Column** annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

- **name**: specify the name of the column.
- **length**: specify the size of thee column, particularly for a String value.
- **nullable**: mark the column to be NOT NULL when the database schema is generated.
- **unique**: mark the column to contain only unique values.

Exercise 5.2. ☐ Create an entity-class Mission. A mission has following fields:

id	Long	The primary key.
name	String	Unique name of the mission.
completed	boolean	
deleted	boolean	

☐ Create MissionRepository, MissionService with MissionServiceImpl and MissionController to make it possible to create, retrieve, update and delete a mission. Add an REST-endpoint to retrieve all missions. Create the necessary DTOs.

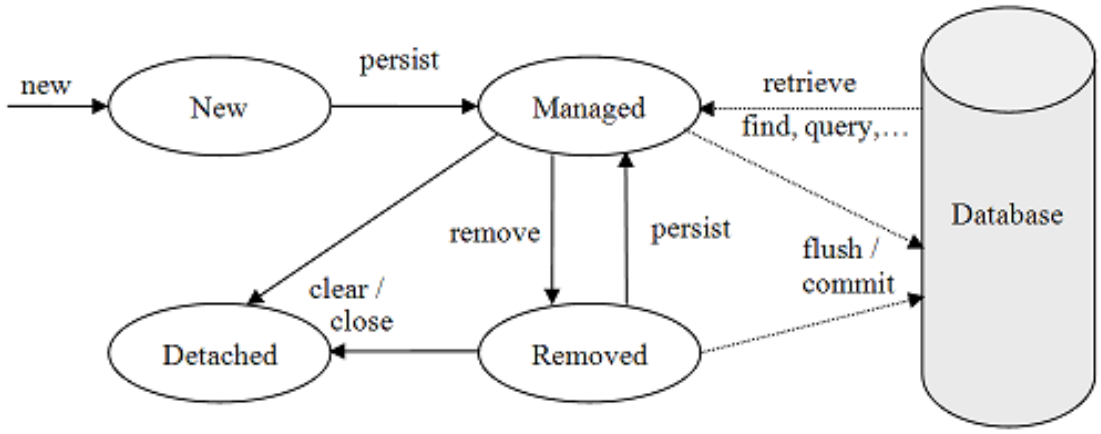


Test all endpoints manually.
Verify the code quality with SonarQube.

5.5 Entity lifecycle

The EntityManager is a core interface of JPA. An instance of EntityManager is used to create and remove entity instances, to find entities by their primary key, and to query over entities. In fact, an instance of the EntityManager is used to interact with the persistence context.

The persistence context is one of the main concepts in JPA. It is a set of all entity object that you are currently using or used recently. You can think of the persistence context as some kind of first-level cache. Each entity object in the persistence context represents a record in the database. The persistence context manages these entity objects and their lifecycle. Each entity object has one of 4 states: new, managed, removed, and detached.



A newly instantiated entity object is in state **new** or **transient**. The entity object hasn't been persisted yet, so there's no database record yet. The persistence context does not know about the entity object yet.

All entity objects that attached to the persistence context are in the lifecycle state **managed**. An entity object becomes managed when it is persisted to the database. Entity object retrieved from the database are also in the managed state. If a managed entity object is modified (updated) within an active transaction, the change is detected by the persistence context and passed on to the database.

When a managed entity is removed within an active transaction, the state changes from managed to removed, and the record is physically deleted from the database (when the transaction commits).

An entity gets detached when it is no longer managed by the persistence context but still represents a record in the database. Detached objects are limited in functionality.

Programmatically we need to use the entitymanager to change the state of the entity object in the persistence context which results in changes or updates in our database.

When using Spring Data JPA you don't have to implement the interaction with the entitymanager. When you create a Repository, the SimpleJpaRepository class provided by Spring Data JPA provides the default implementation. SimpleJpaRepository class internally uses JPA entitymanager.

Exercise 5.3. Open the class SimpleJpaRepository and take a look at its implementation (e.g. save()-method).

5.6 Queries

With Spring Data JPA you can create query methods to select records from the database without writing SQL queries. Behind the scenes, Spring Data JPA will create SQL queries based on the query method and execute the query for us.

The query generation from the method name is a query generation strategy where the invoked query is derived from the name of the query method.

Consider a UserRepository with query method findByEmailAddressAndLastname().

```
public interface UserRepository extends JpaRepository<User, Long> {
  List<User> findByEmailAddressAndLastname(String emailAddress, String
      ↪ lastname);
}
```

Spring Data JPA behind the scene creates a query which translates into the JPQL query: select u from User u where u.emailAddress = ?1 and u.lastname = ?2.

Exercise 5.4. In the interface SuperheroRepository you add a method for retrieving a Superhero by its Supheroname. The method should an Optional.

5.7 Unit testing a repository

Spring Data JPA offers an annotation @DataJpaTest which makes repository testing possible with a minimum of configuration.

Add the h2 dependency with scope test in your pom.xml. This way the unit test for your repository will use the h2 database to test your queries.

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

The builder pattern is used in this example to create entity objects for testing purpose. These entity objects are stored in the in-memory database. There is an IntelliJ IDEA plugin that generates the builder-classes for you (Generate Builder plugin).

The test uses the entity managers, which is injected in the test with the @PersistenceContext annotation. The flush() forces to synchronize the persistence context to the database. The clear() empties the persistence context. Therefore, all entities are detached and can be fetched again from the database.

```java
package be.pxl.superhero.repository;

import be.pxl.superhero.builder.SuperheroBuilder;
import be.pxl.superhero.domain.Superhero;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;

import java.util.Optional;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertTrue;

@DataJpaTest
public class SuperheroRepositoryTest {

    private static final String SUPERHERO_NAME = "Superman";

    @PersistenceContext
    protected EntityManager entityManager;

    @Autowired
    private SuperheroRepository superheroRepository;

    private final Superhero superhero = SuperheroBuilder.aSuperhero()
            .withFirstName("Clark")
            .withLastName("Kent")
            .withSuperheroName(SUPERHERO_NAME)
            .build();

    @BeforeEach
    public void init() {
        superheroRepository.save(superhero);
        entityManager.flush();
        entityManager.clear();
    }

    @Test
    public void returnsSuperheroWithGivenSuperheroName() {
        Optional<Superhero> superheroUnderTest =
            ↪ superheroRepository.findSuperheroBySuperheroName(SUPERHERO_NAME);

        assertTrue(superheroUnderTest.isPresent());
        assertEquals(SUPERHERO_NAME,
```

```
            ↪ superheroUnderTest.get().getSuperheroName());

    }

    @Test
    public void returnsEmptyOptionalWhenNoInstituteWithGivenName() {
        Optional<Superhero> superheroUnderTest =
            ↪ superheroRepository.findSuperheroBySuperheroName("Robin
            ↪ Hood");

        assertTrue(superheroUnderTest.isEmpty());
    }
}
```

Exercise 5.5. You should be able to find a mission by it's name from the database. Add this query in the MissionRepository and test the query.

5.8 Relationships

From a database perspective, table relationships can be:

- one-to-many
- one-to-one
- many-to-many

JPA defines multiple annotations for mapping table relationships. For the superhero-backend we'll use a many-to-many relationship between superheroes and missions. A superhero can participate in multiple missions and multiple superheroes bundle forces in one mission. A link table is necessary to store the data.

Relationships can be either unidirectional of bidirectional. Only one party in a relationship is the owner of the relationship.

5.8.1 Many-to-many bidirectional relationship

The Superhero class is owner of the relationship between Mission and Superhero. Because the relationship is bidirectional, we always maintain a set of a superhero's missions and in a mission we maintain a set of the superheroes participating. When you add a mission to a superhero, you have to make sure the superhero is also added to the mission's list of superheroes. Study the code below.

```
package be.pxl.superhero.domain;

import jakarta.persistence.*;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
```

```
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name="superheroes")
public class Superhero {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(Superhero.class);

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String firstName;
    private String lastName;
    @Column(unique = true)
    private String superheroName;
    @Column(name="notes")
    private String description;
    @ManyToMany
    private List<Mission> missions = new ArrayList<>();

    public Superhero() {
        // JPA only
    }

    public Superhero(String firstName, String lastName, String
        ↪ superheroName) {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug("Creating a new superhero...");
        }
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public Long getId() {
        return id;
    }

    public String getFirstName() {
        if (LOGGER.isTraceEnabled()) {
            LOGGER.trace(String.format("FirstName of %s was revealed.",
                ↪ superheroName));
        }
        return firstName;
    }
```

```
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        if (LOGGER.isFatalEnabled()) {
            LOGGER.fatal(String.format("LastName of %s was revealed.",
                ↪ superheroName));
        }
        return lastName;
    }

    public void addMission(Mission mission) {
        if (mission.isCompleted()) {
            throw new IllegalArgumentException("This mission was already
                ↪ completed.");
        }
        if (onMission(mission)) {
            throw new IllegalArgumentException("This mission was already
                ↪ assigned.");
        }
        missions.add(mission);
        mission.addSuperhero(this);
    }

    public boolean onMission(Mission mission) {
        return missions.stream().anyMatch(m ->
            ↪ m.getName().equals(mission.getName()));
    }

    public List<Mission> getMissions() {
        return missions;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }

    public void setSuperheroName(String superheroName) {
        this.superheroName = superheroName;
    }

    @Override
```

```
    public String toString() {
        return superheroName;
    }
}
```

```
package be.pxl.superhero.domain;

import jakarta.persistence.*;

import java.util.ArrayList;
import java.util.List;

@Entity
public class Mission {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true)
    private String name;

    private boolean completed;

    @ManyToMany(mappedBy = "missions")
    private List<Superhero> superheroes = new ArrayList<>();

    public Long getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    public void addSuperhero(Superhero superhero) {
        superheroes.add(superhero);
```

```
    }

    public List<Superhero> getSuperheroes() {
        return superheroes;
    }


}
```

Exercise 5.6. Create the bidirectional many-to-many relationship between superhero and mission in your project. Write unit tests for the methode addMission() in the class Superhero.

5.9 Transactions

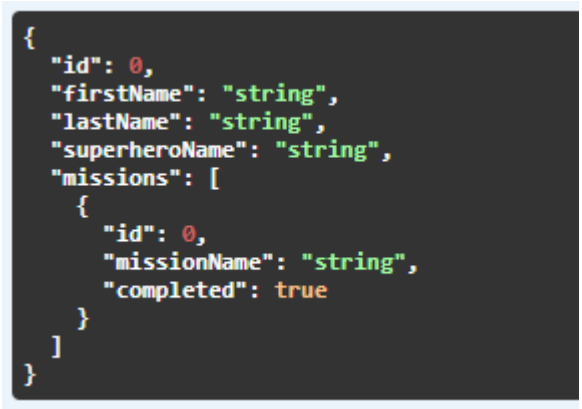
Next, we create the REST-endpoint to add a mission to a superhero.


```
package be.pxl.superhero.service.impl;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.domain.Mission;
import be.pxl.superhero.domain.Superhero;
import be.pxl.superhero.exception.ResourceNotFoundException;
import be.pxl.superhero.repository.MissionRepository;
import be.pxl.superhero.repository.SuperheroRepository;
import be.pxl.superhero.service.SuperheroService;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

@Service
```

```java
public class SuperheroServiceImpl implements SuperheroService {

    private final SuperheroRepository superheroRepository;
    private final MissionRepository missionRepository;

    public SuperheroServiceImpl(SuperheroRepository superheroRepository,
                                MissionRepository missionRepository) {
        this.superheroRepository = superheroRepository;
        this.missionRepository = missionRepository;
    }


    // other methods not included

    @Transactional
    public void addSuperheroToMission(Long superheroId, Long missionId) {
        Superhero superhero =
            ↪ superheroRepository.findById(superheroId).orElseThrow(() ->
            ↪ new ResourceNotFoundException("Superhero", "ID",
            ↪ superheroId));
        Mission mission =
            ↪ missionRepository.findById(missionId).orElseThrow(() -> new
            ↪ ResourceNotFoundException("Mission", "ID", missionId));
        superhero.addMission(mission);
    }
}
```

In the service-layer we retrieve our superhero and mission entity objects (and throw an exception if they don't exist). Then, the mission entity object is added to the superhero's mission and otherwise. Thanks to the @Transactional annotation, all this runs in one transaction. When the transaction is commited, the updated superhero entity is synchronized with the database and the changes are stored in the database. In other words, a record is saved in the link table when no exception occurs. If an exception occurs, the transaction is rolled back.

Lastly, we need DTO's with extra fields with the detailed superhero and mission information.



Chapter 6

Mockito

Learning goals
The junior-colleague
1. can describe the usage of mockito.
2. can implement unit tests with mockito.

6.1 Mocking with Mockito

Mockito is a popular open source framework for mocking objects in software test.

```
package be.pxl.superhero.service;

public class MyService {

    private final OtherService otherService;

    public MyService(OtherService otherService) {
        this.otherService = otherService;
    }

    public int doCalculation(int value) {
        return value * otherService.getValue();
    }
}
```

```
package be.pxl.superhero.service;

public interface OtherService {

    int getValue();
}
```

When writing a unit test for the method doCalculation() we need an object of the class OtherService. We don't need an object of an actual implementation of the interface.

We can just mock the OtherService-object and tell it what it should return when it's called.

```
package be.pxl.superhero.service;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class MyServiceTest {

    @Mock
    private OtherService otherService;
    @InjectMocks
    private MyService myService;

    @Test
    public void doCalculationTest() {
        when(otherService.getValue()).thenReturn(5);
        assertEquals(500, myService.doCalculation(100));
    }
}
```

6.2 Unit testing the service-layer methods

We use Mockito when we test the Service-layer of an application. In this section we discuss more complex unit tests with Mockito.

The name of a superhero must be unique. To accomplish this, we add the unique constraint to our database column. However, it is good practice to implement this business rule in the service layer. Database constraints are the last line of defence, not the only.

```
package be.pxl.superhero.service.impl;

import be.pxl.superhero.api.MissionDTO;
import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroDetailDTO;
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.domain.Mission;
import be.pxl.superhero.domain.Superhero;
import be.pxl.superhero.exception.ResourceNotFoundException;
```

```
import be.pxl.superhero.repository.MissionRepository;
import be.pxl.superhero.repository.SuperheroRepository;
import be.pxl.superhero.service.SuperheroService;
import jakarta.validation.ValidationException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;
import java.util.Optional;

@Service
public class SuperheroServiceImpl implements SuperheroService {

    private final SuperheroRepository superheroRepository;
    private final MissionRepository missionRepository;

    public SuperheroServiceImpl(SuperheroRepository superheroRepository,
                                MissionRepository missionRepository) {
        this.superheroRepository = superheroRepository;
        this.missionRepository = missionRepository;
    }

    public Long createSuperhero(SuperheroRequest superheroRequest) {
        Optional<Superhero> existingSuperhero =
            superheroRepository.findSuperheroBySuperheroName(superheroRequest.getSuperher
        existingSuperhero.ifPresent(s -> {
            throw new ValidationException("This superhero name is already
                ↪ taken.");
        });

        Superhero superhero = new Superhero();
        superhero.setFirstName(superheroRequest.firstName());
        superhero.setLastName(superheroRequest.lastName());
        superhero.setSuperheroName(superheroRequest.superheroName());
        Superhero newSuperhero = superheroRepository.save(superhero);
        return newSuperhero.getId();
    }

    // other methods intentionally left out
}
```

Now we need unit tests for the method *createSuperhero*. There's one problem, we need a SuperheroRepository to thoroughly test the method. The solution is to mock the database layer. Here, mocking means create a dummy layer and no actual database operation will be executed.

In Mockito we can mock methods in 2 different ways:

- Using when-then syntax e.g.: when(..).thenReturn() or when(..).thenAnswer(...)

- Using do-when syntax e.g.: doReturn(..).when()

For mocked objects it is best practice to use when-then option as it provides return type checking and it is more readable.

For mocking void methods, there is no when-then option. We have to use do-when syntax.

```
package be.pxl.superhero.service.impl;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.builder.SuperheroBuilder;
import be.pxl.superhero.domain.Superhero;
import be.pxl.superhero.repository.SuperheroRepository;
import jakarta.validation.ValidationException;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.*;
import org.mockito.junit.jupiter.MockitoExtension;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.mockito.Mockito.when;

@ExtendWith(MockitoExtension.class)
public class SuperheroServiceImplTest {

    private static final String SUPERHERO_NAME = "Superman";
    private static final String FIRST_NAME = "Clark";
    private static final String LAST_NAME = "Kent";
    private static final long SUPERHERO_ID = 115;

    @Mock
    private SuperheroRepository superheroRepository;
    @Mock
    private Superhero newSuperhero;

    @Captor
    private ArgumentCaptor<Superhero> superheroCaptor;

    @InjectMocks
    private SuperheroServiceImpl superheroService;
```

```
    private final Superhero superhero = SuperheroBuilder.aSuperhero()
            .withFirstName(FIRST_NAME)
            .withLastName(LAST_NAME)
            .withSuperheroName(SUPERHERO_NAME)
            .build();

    @Test
    public void throwsValidationExceptionWhenSuperheroNameIsNotUnique() {
        when(superheroRepository.findSuperheroBySuperheroName(SUPERHERO_NAME))
            .thenReturn(Optional.of(superhero));
        SuperheroRequest request = new SuperheroRequest(FIRST_NAME,
            ↪ LAST_NAME, SUPERHERO_NAME);
        ValidationException validationException =
            ↪ assertThrows(ValidationException.class,
            () -> superheroService.createSuperhero(request));
        assertEquals("This superhero name is already taken.",
            ↪ validationException.getMessage());
    }

    @Test
    public void savesSuperheroWhenSuperheroNameIsUnique() {
        when(superheroRepository.findSuperheroBySuperheroName(SUPERHERO_NAME))
            .thenReturn(Optional.empty());
        when(newSuperhero.getId())
            .thenReturn(SUPERHERO_ID);
        when(superheroRepository.save(Mockito.any(Superhero.class)))
            .thenReturn(newSuperhero);
        SuperheroRequest request = new SuperheroRequest(FIRST_NAME,
            ↪ LAST_NAME, SUPERHERO_NAME);
        Long newId = superheroService.createSuperhero(request);
        assertEquals(SUPERHERO_ID, newId);
        Mockito.verify(superheroRepository).save(superheroCaptor.capture());
        Superhero savedSuperhero = superheroCaptor.getValue();
        assertEquals(FIRST_NAME, savedSuperhero.getFirstName());
        assertEquals(LAST_NAME, savedSuperhero.getLastName());
        assertEquals(SUPERHERO_NAME, savedSuperhero.getSuperheroName());
    }

    @Test
    public void findAllSuperheroes() {
        Superhero hero1 = SuperheroBuilder.aSuperhero()
                                        .withSuperheroName("Batman")
                                        .build();
        Superhero hero2 = SuperheroBuilder.aSuperhero()
                                        .withSuperheroName("Superman")
                                        .build();
        when(superheroRepository.findAll()).thenReturn(Arrays.asList(hero1,
```

```
            ↪ hero2));
        List<SuperheroDTO> allSuperheroes =
            ↪ superheroService.findAllSuperheroes();
        assertThat(allSuperheroes.stream().map(SuperheroDTO::superheroName)
                                          .collect(Collectors.toList()))
                .hasSize(2)
                .containsExactly("Batman", "Superman");
    }

}
```

All test classes that use Mockito, are annotated with @ExtendWith(MockitoExtension.class). We want to create a dummy SuperheroRepository, therefore we create it with the annotation @Mock. This dummy SuperheroRepsitory should be used by the SuperheroService under test. We use the annotation @InjectMocks to use all the dummy classes in the SuperheroService instance.

The test throwsValidationExceptionWhenSuperheroNameIsNotUnique is a test for the scenario where the superhero is already saved in the database. Our dummy SuperheroRepository will return this existing superhero when he is retrieved by his superheroName. During the test we call the method createSuperhero with test data but we expect the method to throw a ValidationException. If the exception is not thrown, the test will fail. Finally, the error message of the exception is verified.

The test savesSuperheroWhenSuperheroNameIsUnique is a test for the scenario where the superhero doesn't exist yet. The dummy SuperheroRepository will return an empty Optional. While executing the test, the save method of the SuperheroRepository is called. The result of this save method is used for creating the SuperheroDTO that is returned by the SuperheroService under test. The save-method cannot return null, it would cause our test to end with a NullPointerException. Therefore, we make our dummy SuperheroRepository return something useful. We verify the return value of the method under test (createSuperhero). We verify the save() method is called with the expected argument. This is accomplished with an argument captor.

The third test is a test for the method findAllSuperheroes. We test that the results retrieved from the dummy SuperheroRepository are mapped correctly to SuperheroDTOs. This test uses AssertJ for the assertions. This is a useful library for writing readable assertions.

Exercise 6.1. Add the class SuperheroServiceTest to your project and run (and debug!) the tests. Create a new test class with the name SuperheroServiceUpdateSuperheroTest. Write unit tests for the method updateSuperhero in the class SuperheroServiceTest.

Chapter 7

REST

Learning goals
The junior-colleague
1. can build RESTful services in Spring Boot
2. can explain how HTTP requests are handled by Spring Boot
3. can describe what REST is
4. can explain what a REST request is
5. can explain what a REST response is
6. can provide examples of HTTP response codes
7. can design a RESTful service [**codecademy-rest**]
8. can implement error handling for a RESTful service
9. can implement unit tests for Spring Boot controllers

7.1 CRUD REST API in Spring Boot

We start this chapter with a discussion on SuperheroController. In this class you will find the REST endpoints for CRUD operations for superheroes.

A REST controller in Spring is used to represent REST-endpoints that your application offers to outside systems. In Spring, this can easily be achieved by annotating a class using the @RestController. In order to map the controller to a specific http endpoint path, we will also need to configure the path using the @RequestMapping annotation.

```
package be.pxl.superhero.rest;

import be.pxl.superhero.api.SuperheroDTO;
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.service.SuperheroService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
```

```
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.validation.Valid;
import java.util.List;

@RestController
@RequestMapping("/superheroes")
public class SuperheroController {

    private final SuperheroService superheroService;

    @Autowired
    public SuperheroController(SuperheroService superheroService) {
        this.superheroService = superheroService;
    }

    @GetMapping
    public List<SuperheroDTO> getSuperheroes() {
        return superheroService.findAllSuperheroes();
    }

    @GetMapping("/{superheroId}")
    public SuperheroDTO getSuperheroById(@PathVariable Long superheroId) {
        return superheroService.findSuperheroById(superheroId);
    }

    @PostMapping
    public ResponseEntity<SuperheroDTO> createSuperhero(@Valid @RequestBody
        ↪ SuperheroRequest superheroRequest) {
        return new
            ↪ ResponseEntity<>(superheroService.createSuperhero(superheroRequest),
            ↪ HttpStatus.CREATED);
    }

    @PutMapping("/{superheroId}")
    public SuperheroDTO updateSuperhero(@PathVariable Long superheroId,
        ↪ @Valid @RequestBody SuperheroRequest superheroRequest) {
        return superheroService.updateSuperhero(superheroId,
            ↪ superheroRequest);
    }

    @DeleteMapping("/{superheroId}")
    public ResponseEntity<Void> deleteSuperhero(@PathVariable Long
        ↪ superheroId) {
        boolean deleted = superheroService.deleteSuperhero(superheroId);
```

```
        return deleted? new ResponseEntity<>(HttpStatus.OK) : new
            ↪ ResponseEntity<>(HttpStatus.BAD_REQUEST);
    }
}
```

The class ResponseEntity is used to represent a response to an HTTP request. In other words, it is used as the return type of controller methods which serve REST requests. It is usually preferred to use a ResponseEntity object as the response because it offers us a lot of control over the response we sent to the REST endpoint caller.

Let us take a look into different HTTP methods, also called HTTP verbs, and their usage:

- **GET:** The HTTP GET method is typically used to fetch items. You usually provide the identifier of the item you would like to fetch as a path variable. This is a variable that you can see in the HTTP URI, as part of the path. For example, if you would like to fetch a superhero with the identifier 212, then the URI would look like: http://localhost:8080/superheroes/212.
- **POST:** The HTTP POST method is typically used to create new instances. The HTTP request body will contain the data for those new instances.
- **PUT:** The HTTP PUT method is used to update an existing instance. The identifier of the instance to be updated is provided as a path variable, the data for the instance is contained in the HTTP request body.
- **DELETE:** The HTTP DELETE method is used to delete an existing instance. The object identifier should be provided as a path variable.

Other HTTP verbs are out of scope for this course.

For every HTTP verb we have an annotation to map map the HTTP POST method to a Spring Controller's method.

The @PathVariable annotation is used to indicate that a method argument is a path variable passed by the REST endpoint caller.

Incoming request bodies are in JSON format. **Jackson** JSON ObjectMapper is the default HTTP Converter used by Spring. Jackson converts the incoming JSON Request body to the Java objects. The @RequestBody annotation is used to mark the parameter which contains the data of incoming request body. The HTTP responses are also in JSON format. Here Jackson is reponsible for converting the Java objects to JSON responses. The process of converting Java objects to JSON is called **marshalling**, or serialization. When you convert JSON to Java objects it is called **unmarshalling**, or deserialization.

When Spring receives an HTTP request, the data must be validated. Spring validation offers standard predefined validators.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

When Spring finds an argument of an HTTP method annotated with @Valid, it automatically validates the argument. In this example, the SuperheroRequest is validated. The @NotBlank annotation checks that a String's trimmed length is not empty. The @Size annotation validates that the length of the string is between min and max.

Here is a list of some common validation annotations.

@NotNull	to say that a field must not be null.
@NotEmpty	to say that a list field must not empty.
@NotBlank	to say that a string field must not be the empty string.
@Min and @Max	to say that a numerical field is only valid when it's value is above or below a certain value.
@Size	to validate the length of a string field.
@Email	to say that a string field must be a valid email address.

```
package be.pxl.superhero.api;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

public class SuperheroRequest {

    private final String firstName;
    @NotBlank(message = "lastName must not be blank")
    @Size(message="lastName must have at least 2 characters.", min=2)
    private final String lastName;
    @NotBlank(message = "Superhero name must not be blank")
    private final String superheroName;

    public SuperheroRequest(String firstName, String lastName, String
        ↪ superheroName) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.superheroName = superheroName;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getSuperheroName() {
        return superheroName;
    }
}
```

When the target argument fails the validation, Spring Boot throws a MethodArgumentNotValidException exception.

7.2 Exception handling

In Spring Boot there are several ways to convert an exception to an HTTP response status. One of the simplest ways is to mark an exception with @ResponseStatus. When Spring catches the marked exception, it uses the settings provided in @ResponseStatus to create the HTTP response.

```
package be.pxl.superhero.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class ResourceNotFoundException extends RuntimeException {

    private final String resourceName;
    private final String fieldName;
    private final Object fieldValue;

    public ResourceNotFoundException( String resourceName, String
        ↪ fieldName, Object fieldValue) {
        super(String.format("%s not found with %s : '%s'", resourceName,
            ↪ fieldName, fieldValue));
        this.resourceName = resourceName;
        this.fieldName = fieldName;
        this.fieldValue = fieldValue;
    }

    public String getResourceName() {
        return resourceName;
    }

    public String getFieldName() {
        return fieldName;
    }

    public Object getFieldValue() {
        return fieldValue;
    }
}
```

When handling the MethodArgumentNotValidException thrown by Spring validation we want to be able to manipulate the response. In this case the solution is using a @ControllerAdvice.

```java
package be.pxl.superhero.rest.advice;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

import java.util.ArrayList;
import java.util.List;

@ControllerAdvice
public class ValidatedExceptionHandler {

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ErrorResponseBody> handleValidationExceptions(
            MethodArgumentNotValidException ex) {
        List<String> errors = new ArrayList<>();
        ex.getBindingResult().getAllErrors().forEach((error) -> {
            String errorMessage = error.getDefaultMessage();
            errors.add(errorMessage);
        });
        return new ResponseEntity<>(new ErrorResponseBody("Validation
            ↪ failed.", errors), HttpStatus.BAD_REQUEST);
    }


    private static class ErrorResponseBody {
        private final String message;
        private final List<String> errors;

        public ErrorResponseBody(String message, List<String> errors) {
            this.message = message;
            this.errors = errors;
        }

        public String getMessage() {
            return message;
        }

        public List<String> getErrors() {
            return errors;
        }
    }
}
```

We implement methods annotated with @ExceptionHandler for handling specific exception classes.

The @ControllerAdvice is a global component in the application for error handling. The actual mechanism is simple but also very flexible: it gives us full control over the body of the response as well as the status code. Several exceptions can be handled by the same method.

There a more possibilities for handling exceptions in Spring Boot than the two discussed here [**exhand**].

7.3 Unit testing

MockMvc is defined as the main entry point for server-side Spring MVC testing. Tests annotated with @WebMvcTest will auto-configure Spring Security and MockMvc. These tests will also auto-configure all classes relevant to MVC tests (e.g. @Controller, @ControllerAdvice, ...). Using the annotation @MockBean we can provide mock objects for the @Service components. By default, @WebMvcTest adds all @Controller beans to the application context. We can specify a subset of controllers by using the controllers attribute, as we have done in the examples discussed here. The method perform in the class MockMvc performs a request and returns a type that allows chaining further actions, such as asserting expectations, on the result. When implementing unit tests, we pass the path parameters using MockMvcRequestBuilders and verify the status response codes and response content using MockMvcResultMatchers and MockMvcResultHandlers.

```
package be.pxl.superhero.rest;

import be.pxl.superhero.api.SuperheroDetailDTO;
import be.pxl.superhero.builder.SuperheroDetailDTOBuilder;
import be.pxl.superhero.exception.ResourceNotFoundException;
import be.pxl.superhero.service.SuperheroService;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static be.pxl.superhero.builder.SuperheroDTOBuilder.FIRSTNAME;
import static be.pxl.superhero.builder.SuperheroDTOBuilder.LASTNAME;
import static be.pxl.superhero.builder.SuperheroDTOBuilder.SUPERHERO_ID;
import static be.pxl.superhero.builder.SuperheroDTOBuilder.SUPERHERO_NAME;
import static org.mockito.Mockito.doThrow;
import static org.mockito.Mockito.when;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
```

```
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(controllers = SuperheroController.class)
public class SuperheroControllerGetSuperheroTest {

    @MockBean
    private SuperheroService superheroService;

    @Autowired
    private MockMvc mockMvc;

    @Test
    public void getSuperheroByIdReturnsExistingSuperhero() throws Exception
        ↪ {
        SuperheroDetailDTO superheroDetailDTO = SuperheroDetailDTOBuilder
                .aSuperheroDetailDTO()
                .withId(SUPERHERO_ID)
                .build();
        when(superheroService.findSuperheroById(SUPERHERO_ID))
                .thenReturn(superheroDetailDTO);
        mockMvc.perform(MockMvcRequestBuilders
                        .get("/superheroes/{id}", SUPERHERO_ID)
                        .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(SUPERHERO_ID))
                .andExpect(jsonPath("$.firstName").value(FIRSTNAME))
                .andExpect(jsonPath("$.lastName").value(LASTNAME))
                .andExpect(jsonPath("$.superheroName").value(SUPERHERO_NAME));
    }

    @Test
    public void
        ↪ getSuperheroByIdReturnsNotFoundWhenResourceNotFoundExceptionIsThrown()
        ↪ throws Exception {
        doThrow(ResourceNotFoundException.class)
                .when(superheroService)
                .findSuperheroById(SUPERHERO_ID);
        mockMvc.perform(MockMvcRequestBuilders
                        .get("/superheroes/{id}", SUPERHERO_ID)
                        .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isNotFound());
    }
}
```

```
package be.pxl.superhero.rest;
```

```java
import be.pxl.superhero.api.SuperheroRequest;
import be.pxl.superhero.builder.SuperheroRequestBuilder;
import be.pxl.superhero.service.SuperheroService;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.junit.jupiter.api.Test;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.request.MockMvcRequestBuilders;

import static be.pxl.superhero.builder.SuperheroRequestBuilder.FIRSTNAME;
import static be.pxl.superhero.builder.SuperheroRequestBuilder.LASTNAME;
import static
    ↪ be.pxl.superhero.builder.SuperheroRequestBuilder.SUPERHERO_NAME;
import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.verify;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultHandlers.print;
import static
    ↪ org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest(controllers = SuperheroController.class)
public class SuperheroControllerCreateSuperheroTest {
    @MockBean
    private SuperheroService superheroService;

    @Autowired
    private MockMvc mockMvc;

    @Captor
    private ArgumentCaptor<SuperheroRequest> superheroRequestCaptor;

    @Test
    public void testLastNameIsRequired() throws Exception {
        SuperheroRequest superheroRequest = SuperheroRequestBuilder
                .aSuperheroRequest()
                .withLastName("")
                .build();
        mockMvc.perform(MockMvcRequestBuilders.post("/superheroes")
                        .content(asJsonString(superheroRequest))
                        .contentType(MediaType.APPLICATION_JSON)
                        .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isBadRequest());
    }
```

```
@Test
public void testLastNameWithOneCharacterIsNotValid() throws Exception {
    SuperheroRequest superheroRequest = SuperheroRequestBuilder
            .aSuperheroRequest()
            .withLastName("A")
            .build();
    mockMvc.perform(MockMvcRequestBuilders.post("/superheroes")
                    .content(asJsonString(superheroRequest))
                    .contentType(MediaType.APPLICATION_JSON)
                    .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isBadRequest());
}


@Test
public void testSuperheroNameIsRequired() throws Exception {
    SuperheroRequest superheroRequest = SuperheroRequestBuilder
            .aSuperheroRequest()
            .withSuperheroName("")
            .build();
    mockMvc.perform(MockMvcRequestBuilders.post("/superheroes")
                    .content(asJsonString(superheroRequest))
                    .contentType(MediaType.APPLICATION_JSON)
                    .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isBadRequest());
}

@Test
public void testSuperheroIsCreatedWhenAllConstraintsSatified() throws
    ↪ Exception {
    SuperheroRequest superheroRequest =
        ↪ SuperheroRequestBuilder.aSuperheroRequest().build();
    mockMvc.perform(MockMvcRequestBuilders.post("/superheroes")
                    .content(asJsonString(superheroRequest))
                    .contentType(MediaType.APPLICATION_JSON)
                    .accept(MediaType.APPLICATION_JSON))
            .andExpect(status().isCreated());
    verify(superheroService).createSuperhero(superheroRequestCaptor.capture());

    SuperheroRequest createdSuperhero =
        ↪ superheroRequestCaptor.getValue();
    assertThat(createdSuperhero.getFirstName()).isEqualTo(FIRSTNAME);
    assertThat(createdSuperhero.getLastName()).isEqualTo(LASTNAME);
    assertThat(createdSuperhero.getSuperheroName()).isEqualTo(SUPERHERO_NAME);
}

public static String asJsonString(final Object obj) {
```

```
        try {
            return new ObjectMapper().writeValueAsString(obj);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

If you want to print the JSON result, you can use the MockMvcResultHandler.print() method.

```
@Test
public void getSuperheroByIdReturnsExistingSuperhero() throws Exception {
    SuperheroDetailDTO superheroDetailDTO = SuperheroDetailDTOBuilder
            .aSuperheroDetailDTO()
            .withId(SUPERHERO_ID)
            .build();
    when(superheroService.findSuperheroById(SUPERHERO_ID))
            .thenReturn(superheroDetailDTO);
    mockMvc.perform(MockMvcRequestBuilders
                    .get("/superheroes/{id}", SUPERHERO_ID)
                    .accept(MediaType.APPLICATION_JSON))
            .andDo(print())
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.id").value(SUPERHERO_ID))
            .andExpect(jsonPath("$.firstName").value(FIRSTNAME))
            .andExpect(jsonPath("$.lastName").value(LASTNAME))
            .andExpect(jsonPath("$.superheroName").value(SUPERHERO_NAME));
}
```

Additional examples can be found in online blogs [**mockmvc**] or Spring documentations.

7.4 Exercise: Superheroes

We expand the superheroes application. First, we implement CRUD operations for missions. Next, we will add an endpoint to add a superhero to a mission.

Change the database configuration to use a mysql or mariadb database, so you will not lose your data every time you restart the Spring boot backend. You may need to introduce multiple DTOs to implement the different responses (e.g. MissionDTO and MissionDetailDTO).

7.4.1 CRUD operations for missions

POST **/api/missions/**

create a new mission

Parameter

no parameter

Body application/json

```
{
    "missionName": "Secret Mission X"
}
```

Response application/json

200 ok

```
{
    "id": 5,
    "missionName": "Secret Mission X",
    "completed": false,
    "deleted": false,
    "superheroes": []
}
```

400 error: validation error

```
    {
    "message": "Validation failed.",
    "errors": [
        "missionName must not be blank"
    ]
    }
```

GET **/api/missions/**

retrieve all missions. Deleted missions are available in the response.

Parameter

no parameter

Response application/json

200 ok


```
[
    {
        "id": 1,
        "missionName": "Sercret Mission X",
        "completed": true,
        "deleted": false
    },
    {
        "id": 2,
        "missionName": "Secret Mission Y",
        "completed": false,
        "deleted": false
    }
]
```

GET **/api/missions/{missionId}**

retrieve details of a mission

Parameter

missionId unique id of a mission

Response application/json

200 ok


```
{
    "id": 3,
    "missionName": "Secret Mission X",
    "completed": false,
    "deleted": false,
    "superheroes": [
        {
            "id": 4,
            "firstName": "Hero 1",
            "lastName": "Karma",
            "superheroName": "June"
        },
        {
            "id": 5,
            "firstName": "Hero 2",
            "lastName": "Leon",
            "superheroName": "Semois"
        }
    ]
}
```


400 error: validation error

PUT **/api/missions/{missionId}**

update a mission

Parameter

missionId unique id of a mission

Body application/json

```
{
    "missionName": "Secret Mission Y",
    "completed": true
}
```

Response application/json

200 ok

```
{
    "id": 3,
    "missionName": "Secret Mission Y",
    "completed": true,
    "deleted": false,
    "superheroes": [
        {
            "id": 4,
            "firstName": "Hero 1",
            "lastName": "Karma",
            "superheroName": "June"
        },
        {
            "id": 5,
            "firstName": "Hero 2",
            "lastName": "Leon",
            "superheroName": "Semois"
        }
    ]
}
```

400 error: validation error

```
    {
    "message": "Validation failed.",
    "errors": [
        "missionName must not be blank"
    ]
    }
```

PUT **/api/missions/{missionId}**

update a mission (additional response)

Parameter

missionId unique id of a mission

Response application/json

404 error: mission not found


```
{
    "timestamp": "2022-04-13T14:39:13.675+00:00",
    "status": 404,
    "error": "Not Found",
    "message": "Mission not found with ID :'5'",
    "path": "/api/missions/5"
}
```


DELETE **/api/missions/{missionId}**

delete a mission. The status of the mission is updated to deleted.

Parameter

missionId unique id of a mission

Response application/json

200 ok

400 error: mission is already deleted

404 error: mission not found


```
{
    "timestamp": "2022-04-13T14:39:13.675+00:00",
    "status": 404,
    "error": "Not Found",
    "message": "Mission not found with ID :'5'",
    "path": "/api/missions/5"
}
```


7.4.2 Add superhero to mission

POST **/api/superheroes/add-superhero-to-mission**

add a superhero to a mission

Parameter

no parameter

Body application/json

```
{
  "missionId": 15,
  "superheroId": 31
}
```

Response application/json

200 ok

400 error: validation error mission is deleted

```
    {
    "message": "Validation failed.",
    "errors": [
        "mission is deleted"
    ]
    }
```

404 error: mission or superhero not found

```
{
    "timestamp": "2022-04-13T14:39:13.675+00:00",
    "status": 404,
    "error": "Not Found",
    "message": "Mission not found with ID :'5'"
}
```

Chapter 8

Servlets

Learning goals
The junior-colleague

1. can explain what a Servlet is
2. can explain how Spring Boot uses Servlet technology (DispatcherServlet!)
3. can explain and use the lifecycle moments of a Servlet
4. can explain the usage of @WebServlet, @WebFilter and @WebListener
5. can explain the usage of HttpSession
6. can create an HTTPServlet to handle requests
7. can use cookies and HTTPSession in a web application
8. can use WebClient to send a simple HTTP request
9. can implement a @WebFilter
10. can implement a @WebListener
11. can explain the Model-View-Controller design pattern
12. can explain the role of DispatcherServlet in Spring
13. can consume Rest services with WebClient
14. can implement a simple view with Thymeleaf

Source for this chapter:
https://github.com/custersnele/PAJ_servlets.git

8.1 What is a Servlet

Relatively few applications still use Servlets directly, but they're still the underlying technology behind the vast majority of Java and JVM web frameworks. A servlet's job is to take a client's request and send back a response. For example, we can use a Servlet to collect input from a user through an HTML form, query records from a database, and create web pages dynamically. Servlets are under the control of another Java application called a Servlet Container. When an application running in a web server receives a request, the server hands the request to the Servlet Container – which in turn passes it to the target Servlet. Java servlets typically run on the HTTP protocol.

When using Spring Boot, the @ServletComponentScan annotation automatically registers the Servlet components for embedded web servers. This annotation supports the following Servlet 3.0 annotations:

- @WebServlet annotation
- @WebFilter annotation
- @WebListener annotation

```
package be.pxl.paj.servlets;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.ServletComponentScan;

@SpringBootApplication
@ServletComponentScan
public class ServletDemoApplication {

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

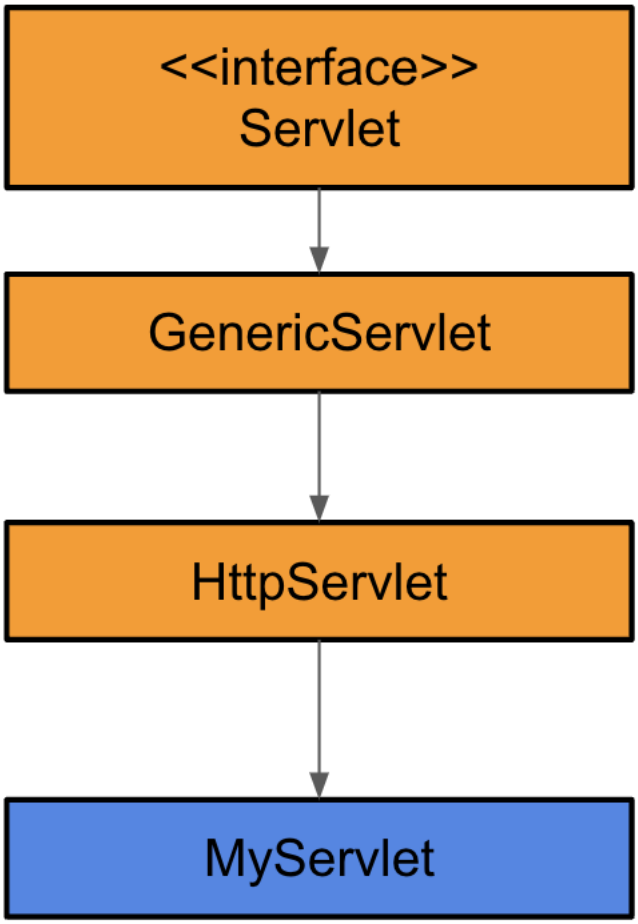
8.2 Servlet class hierarchy

The Servlet interface is the root interface of the servlet class hierarchy. All Servlets need to either directly or indirectly implement the Servlet interface.

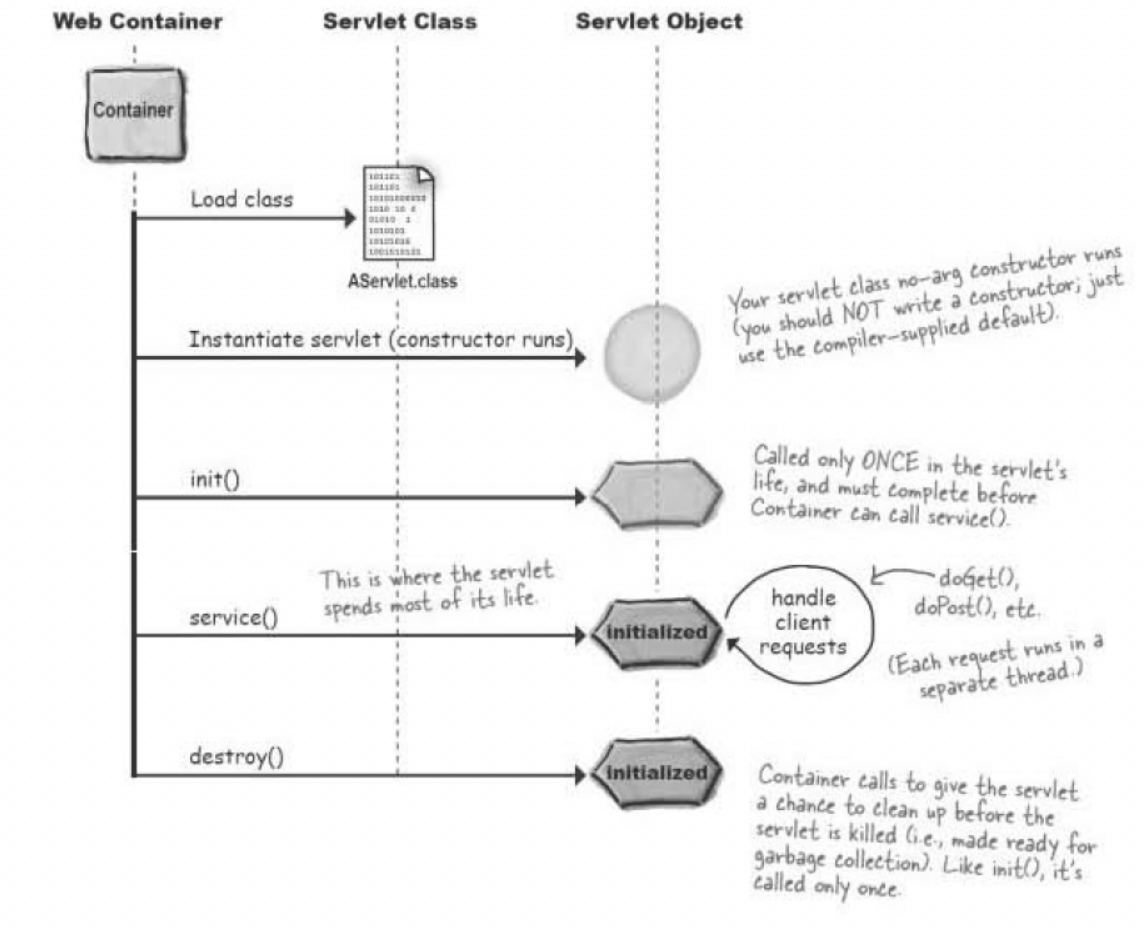
The Servlet interface of the javax.servlet package defines the methods that the Servlet container calls to manage the servlet life cycle.

The GenericServlet class of the Servlet API implements the Servlet interface.

To develop a servlet that communicates using HTTP, we need to extend the HttpServlet class.


<<interface>>
Servlet
GenericServlet
HttpServlet
MyServlet

8.3 Lifecycle of a servlet



The Servlet life cycle mainly goes through three stages:[**servlet_lifecycle**]

- **Loading and initializing a Servlet.** After the Servlet is instantiated successfully, the Servlet container initializes the instantiated Servlet object. The container initializes the Servlet object by invoking the **init()** method. The javax.servlet.ServletConfig interface is implemented by a Servlet container to pass configuration information to a servlet, during initialization of a servlet.
- **Request handling.** After initialization, the Servlet instance is ready to serve the client requests. The Servlet's **service()** method is invoked to handle a client request. When multiple client requests come, the server will create multiple threads. Each client request corresponds to a thread that calls the Servlet's service() method. That's the nice thing about Java, it's multithreaded and different threads (HTTP requests) can make use of the same instance. It would otherwise be too expensive to recreate, init() and destroy() them for every single request. If you encounter threadsafety issues when using servlets, then it is your fault, not Java's nor Servlet's fault.
- **Destroying the Servlet.** When a Servlet container decides to destroy the Servlet, all currently running threads can complete their jobs and next, the Servlet container calls the destroy() method on the Servlet instance.

8.3.1 Example 1

```
package be.pxl.paj.servlets;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Locale;

@WebServlet(value="/DateTimeServlet")
public class DateTimeServlet extends HttpServlet {

    private static final DateTimeFormatter FORMATTER_EN =
        ↪ DateTimeFormatter.ofPattern("EEEE dd/MM/yyyy HH:mm:ss",
        ↪ Locale.ENGLISH);
    private static final DateTimeFormatter FORMATTER_NL =
        ↪ DateTimeFormatter.ofPattern("EEEE dd/MM/yyyy HH:mm:ss", new
        ↪ Locale("nl"));

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        ↪ throws IOException {
        PrintWriter writer = resp.getWriter();
        String language = req.getParameter("lang");
        LocalDateTime dateTime = LocalDateTime.now();
        String dateAsText = "nl".equals(language) ?
                FORMATTER_NL.format(dateTime) :
                    ↪ FORMATTER_EN.format(dateTime);
        writer.println("<html>" +
                "<body>" +
                "<h1 style=\"text-align:center\">" + dateAsText +
                    ↪ "</h1></body>" +
                "</html>");
    }
}
```

8.3.2 Example 2

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
```

```
    <title>Beer selection</title>
</head>
<body>
<h1>Beer Selection</h1>
<form method="POST" action="/SelectBeer.do">
    <p>Select beer characteristics:</p>
    <label for="color">Color:</label>
    <select id="color" name="color" size="1">
        <option value="light">Light</option>
        <option value="amber">Amber</option>
        <option value="brown">Brown</option>
        <option value="dark">Dark</option>
    </select>
    <br/>
    <br/>
    <input type="submit" />
</form>
</body>
</html>
```

```
package be.pxl.paj.servlets;

import be.pxl.paj.servlets.service.BeerExpert;
import org.springframework.beans.factory.annotation.Autowired;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

@WebServlet(value = "/SelectBeer.do")
public class SelectBeerServlet extends HttpServlet {

    private final BeerExpert beerExpert;

    @Autowired
    public SelectBeerServlet(BeerExpert beerExpert) {
        this.beerExpert = beerExpert;
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
        ↪ throws ServletException, IOException {
        String color = req.getParameter("color");
```

```
        List<String> result = beerExpert.getBrands(color);
        PrintWriter writer = resp.getWriter();

        writer.println("<html>" +
                "<body>" +
                "<h1 style=\"text-align:center\">Try these beers:</h1><p>" +
                String.join(", ", result) +
                "</p></body>" +
                "</html>");
    }
}
```

```
package be.pxl.paj.servlets.service;

import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.List;

@Service
public class BeerExpert {

    public List<String> getBrands(String color) {
        List<String> brands = new ArrayList<>();
        if (color.equals("amber")) {
            brands.add("Jack Amber");
            brands.add("Red Moose");
        } else {
            brands.add("Jail Pale Ale");
            brands.add("Gout Stout");
        }
        return brands;
    }
}
```

8.4 Session tracking

Session management is a way to maintain state of an user.

HTTP protocol is a stateless so we need to maintain state using session management techniques. Each time user requests to the server, server treats the request as the new request.

There are several techniques for session management:

- Cookies
- Hidden Form Field

- URL Rewriting
- HttpSession

In this course we will provide examples for using cookies and HttpSession.

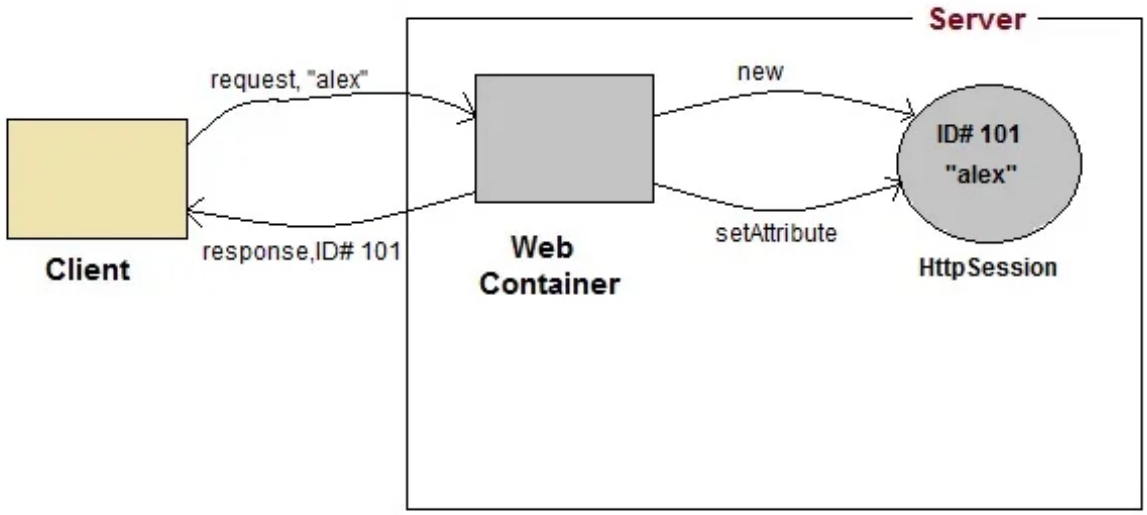
8.4.1 Cookies

The method getCookies() in HttpServletRequest returns an array containing all of the Cookie objects the client sent with this request.

```
package be.pxl.paj.servlets;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(name = "Cookies", value = "/Cookies")
public class UsingCookies extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse
        ↪ response)
            throws IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>");
        out.println("A Web Page");
        out.println("</TITLE>");
        out.println("</HEAD>");
        out.println("<BODY");

        Cookie[] cookies = request.getCookies();
        boolean foundCookie = false;

        for (int loopIndex = 0; loopIndex < cookies.length; loopIndex++) {
            Cookie cookie1 = cookies[loopIndex];
            if (cookie1.getName().equals("color")) {
                out.println("bgcolor = " + cookie1.getValue());
                foundCookie = true;
            }
        }
```

```
        if (!foundCookie) {
            Cookie cookie1 = new Cookie("color", "cyan");
            cookie1.setMaxAge(24 * 60 * 60);
            response.addCookie(cookie1);
        }

        out.println(">");
        out.println("<H1>Setting and Reading Cookies</H1>");
        out.println("This page will set its background color using a cookie
            ↪ when reloaded.");
        out.println("</BODY>");
        out.println("</HTML>");
    }
}
```

8.4.2 HttpSession



HTTP sessions are identified by a unique session ID. A session ID is a pseudo-random number generated at the runtime.

On client's first request, the Web Container generates a unique session ID and gives it back to the client encapsulated in the response. The client sends back the session ID with each request. This way the web container can identify where the request is coming from. The Web Container uses this ID, finds the matching session with the ID and associates the session with the request.

Session hijacking is a known attack for HTTP sessions and can be prevented if all the requests going over the network are enforced to be over a secure connection (meaning, HTTPS).

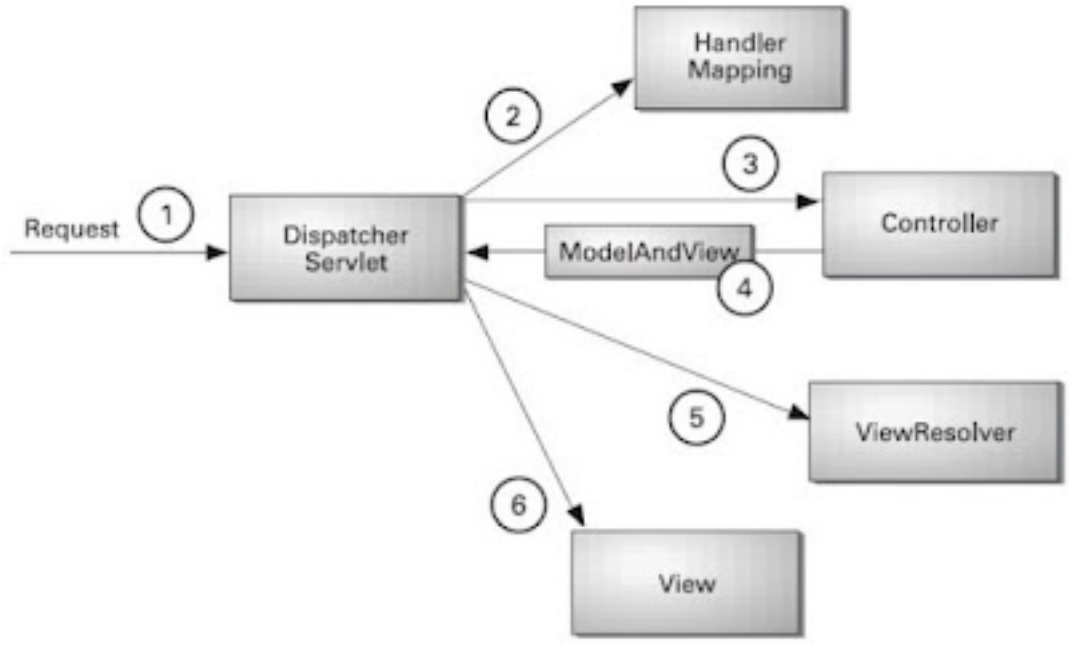
Below is the click_the_button.html file. There are several directories where you can store static html pages. In our demo we've chosen to create a folder named 'static' in the application's resources folder.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Click the Button!</title>
</head>
<body>
<h1>Click the button!</h1>
<form action="/ButtonClicked" method="POST">
    <input type="submit" value="Click me!">
</form>
</body>
</html>
```

```
package be.pxl.paj.servlets;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

@WebServlet(name = "/ButtonClicked", value="/ButtonClicked")
public class ButtonClickedServlet extends HttpServlet {

    @Override
    public void doPost(HttpServletRequest request,
                                    HttpServletResponse response) throws
                                        ↪ IOException, ServletException {

        int clickCount = 0;

        //get the session, which contains user-specific data
        HttpSession session = request.getSession();

        if(session.getAttribute("clickCount") != null){
            //we've already stored the clickCount in a previous request, so
                ↪ get it
            clickCount = (int) session.getAttribute("clickCount");
        }

        clickCount++;

        //store the new clickCount in the session
```

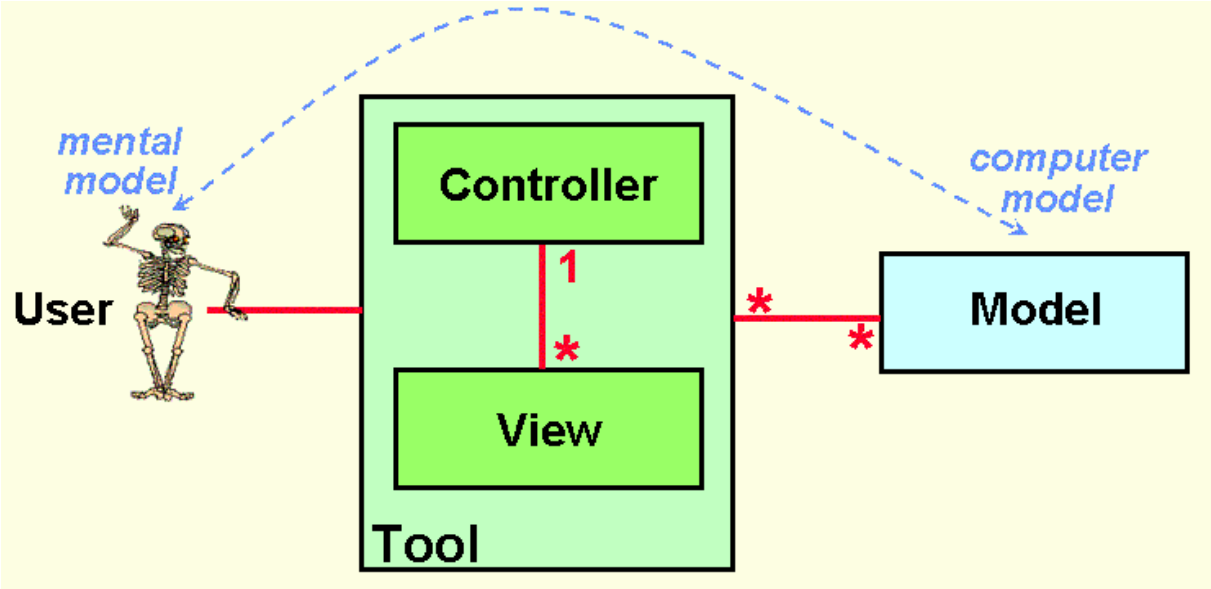
```
        session.setAttribute("clickCount", clickCount);

        //output the clickCount for this user
        response.getOutputStream().println("<h1>You clicked the button " +
            ↪ clickCount + " times.</h1>");
        response.getOutputStream().println("<p>Click <a
            ↪ href=\"/click_the_button.html\">here</a> to go back to the
            ↪ button.</p>");
    }
}
```

8.5 Spring DispatcherServlet



The Spring web framework is built around the MVC (Model-View-Controller) pattern. In Spring Web MVC, all the incoming request are intercepted by the DispatcherServlet that works as a front controller. The DispatcherServlet forwards the request to the controller reponsible for handling the request, i.e. the controller that matches the requested URL. The controller returns a response, besides REST responses, objects of ModelAndView can be returned as well. If a ModelAndView response is returned, the DispatcherServlet invokes the specified view component.

The MVC pattern was first introduced in 1979 by computer scientist Trygve Mikkjel Heyerdahl Reenskaug. He wanted to come up with a solution on how to break up a complex user application into smaller manageable components.



This is a basic breakdown of the MVC pattern:

- Model - A model contains the data of the application.
- Controller - A controller interprets user input and transform it into a model that is represented to the user by the view. The controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering. Here, the @Controller annotation is used to mark the class as the controller.
- View - A view represents the provided information in a particular format. Spring also supports view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

In Spring Web MVC, we have this extra component, the DispatcherServlet class, which works as the front controller. It is responsible to manage the flow of the Spring MVC application.

8.5.1 Views

This section shows how Thymeleaf can be used as an template engine by Spring Boot.

First we add the dependency.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

By default, Spring Boot looks for templates in the directory src/main/resources/templates.

We have two templates, on for submitting a country name and a second for displaying the universities of the chosen country.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="UTF-8">
    <title>Search universities</title>
</head>
<body>
<form th:action="@{/universities}">
    <div>
        <label>Country:</label>
        <input type="text" th:name="country" />
    </div>
    <input type="submit"/>
</form>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<div th:each="university : ${universities}">
    <a th:href="${university.webPages.get(0)}"><p
        ↪ th:text="${university.name}"></a>
</div>
</body>
</html>
```

Let's have a look at the controller.

```
package be.pxl.paj.servlets.rest;

import be.pxl.paj.servlets.domain.University;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.reactive.function.client.WebClient;

@Controller
public class UniversityController {

    private final WebClient webClient =
        ↪ WebClient.create("http://universities.hipolabs.com");

    @GetMapping("/country_selection")
```

```
    public String countrySelection() {
        return "country_selection";
    }

    @RequestMapping("/universities")
    public String addUser(@RequestParam(value = "country") String country,
        ↪ Model model) {
        WebClient.ResponseSpec response = webClient.get()
                .uri(String.format("search?country=%s", country))
                .retrieve();
        University[] universities =
            ↪ response.bodyToMono(University[].class).block();
        model.addAttribute("universities", universities);
        return "universities";
    }
}
```

The University class is a POJO-class.

```
package be.pxl.paj.servlets.domain;

import com.fasterxml.jackson.annotation.JsonProperty;

import java.util.List;

public class University {
    private String name;
    @JsonProperty("web_pages")
    private List<String> webPages;

    public String getName() {
        return name;
    }

    public List<String> getWebPages() {
        return webPages;
    }
}
```

Exercise 8.1. To get familiar with the syntax of thymeleaf, use the interactive tutorial http://itutorial.thymeleaf.org/. Solve the basic exercises.

8.6 @WebFilter

Filters let you intercept the request. And if you can intercept the request, you can also control the response. And, the servlet never knows that someone stepped in between

the client request and the servlet container's invocation of the servlet's service() method. Therefore, if you choose to write and configure a filter, you're able to affect all of your servlets. For example, if you want to add user request tracking to every servlet in your application or manipulate the output from every servlet? No problem. Just write a filter and you don't even have to touch the servlets.

```
package be.pxl.paj.servlets;


import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Collections;

@WebFilter(urlPatterns = { "/SelectBeer.do" })
public class HeaderLogFilter implements Filter {

    private static final Logger LOGGER =
        ↪ LogManager.getLogger(HeaderLogFilter.class);

    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
                         FilterChain chain) throws IOException,
                             ↪ ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse rep = (HttpServletResponse) response;

        LOGGER.info("----- Request ---------");
        Collections.list(req.getHeaderNames()).forEach(n -> LOGGER.info(n +
            ↪ ": " + req.getHeader(n)));

        chain.doFilter(request, response);

        LOGGER.info("----- response ---------");

        rep.getHeaderNames().forEach(n -> LOGGER.info(n + ": " +
            ↪ rep.getHeader(n)));

        LOGGER.info("response status: " + rep.getStatus());
```

```
    }
}
```

8.7 @WebListener

@WebListener annotation is used to define a servlet listener component in a web application.

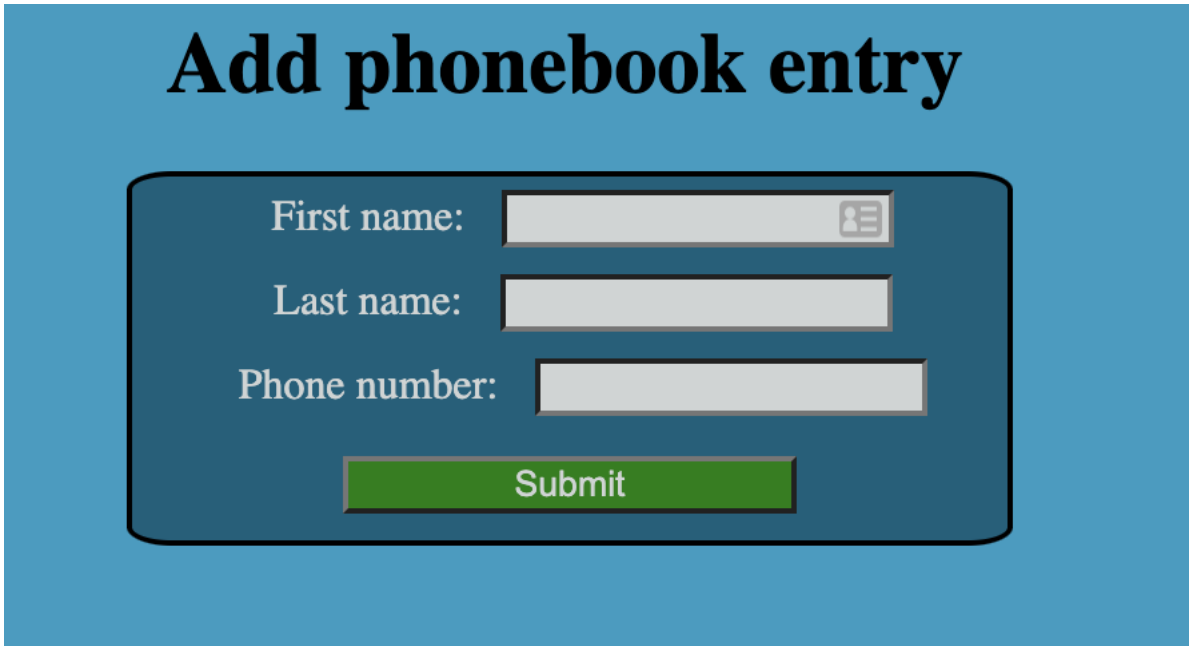
Many listeners are defined for events involving life cycles of ServletContext, HttpSession and ServletRequest. Following listeners are supported for @WebListener:

- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

```
package be.pxl.paj.servlets;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class ExampleContextListener implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent servletContextEvent)
        ↪ {
        System.out.println("ServletDemo starting up!");
    }

    @Override
    public void contextDestroyed(ServletContextEvent servletContextEvent) {
        System.out.println("ServletDemo shutting down!");
    }
}
```

8.8 Exercises

Exercise 8.2. The html-page phonebook_add.html is provided. Create a Servlet to handle the form's POST request. The data is saved to a (in-memory) database. Make sure the phonenumber is unique. Create a Servlet to display all the entries in the phonebook.



Exercise 8.3. Implement the following user stories in the superhero application.

Title: Story 1
As a superhero I can provide worklog for a specific mission.
Description: A thymeleaf form is created to enter the following information: • superhero name - text field • mission - selectbox with all current missions (not deleted or finished) • date • task - textbox First, the information is validated. The superhero is automatically assigned to the mission, if he isn't yet. The task description may not be blank. If the data is valid, it's stored in the database.
Acceptance criteria: Given the superhero is assigned to the mission, when all the requested data is provided, the worklog is saved in the database. Given the superhero is not assigned to the mission, when all the requested data is provided, the superhero is assigned to the mission and the worklog is saved in the database. Given the superhero enters worklog information, when required information is left blank, a meaningful error message is shown and the data can be completed.

Title: Story 2
As a user I can request all the worklogs for a mission.
Description: A REST endpoint is implemented to retrieve all the worklogs for a mission. There is one parameter, superheroname, which filters the worklogs for the given superhero. If no parameter value is provided, all the worklogs are returned. If the mission is completed, the worklogs can be retrieved. If the mission is deleted, the worklogs can not be retrieved.
Acceptance criteria: Given a user requesting the worklogs of a mission, when no parameter is provided and the mission is not deleted or completed, the requested data is returned. Given a user requesting the worklogs of a mission, when a superheroname is provided and the mission is not deleted or completed, the worklogs for entered by the given superhero are returned. Given a user requesting the worklogs of a mission, when the misssion is completed, a requested worklogs are returned. Given a user requesting the worklogs of a mission, when the misssion is deleted, an http status 404 (NOT FOUND) is returned.