Hogeschool PXL
PXL-Digital

Java Advanced
An Introduction to Spring Boot

Author
Nele Custers

September 17, 2023

Contents

1 Kennismaking met Spring Boot **1**
- 1.1 Enterprise toepassingen met Java . . . 1
- 1.2 Bootstrapping van een eenvoudige Spring Boot applicatie . . . 4
  - 1.2.1 Gebruik van Spring Initializr . . . 4
- 1.3 Het demo-project uitvoeren . . . 6
- 1.4 De Maven POM file . . . 7
- 1.5 Inversion of Control (IoC) en dependency injection . . . 9
  - 1.5.1 Spring Beans . . . 9
  - 1.5.2 Application context . . . 9

2 REST **16**
- 2.1 HTTP-verzoekmethoden . . . 16
- 2.2 Spring Boot Starter Web . . . 17
- 2.3 De RestController . . . 19
- 2.4 MusicPlaylist . . . 21
  - 2.4.1 Een liedje toevoegen aan een playlist . . . 21
  - 2.4.2 De playlist opvragen . . . 26
  - 2.4.3 Liedjes van één genre . . . 29
  - 2.4.4 Gegevens van een liedje aanpassen . . . 31
  - 2.4.5 Een liedje verwijderen . . . 34
- 2.5 REST Endpoints . . . 37

3 Unit testing **41**
- 3.1 JUnit . . . 41
- 3.2 Unit test voor een constructor . . . 41
- 3.3 Unit test voor een setter . . . 43
- 3.4 Unit test voor een getter . . . 47

4 Streams **53**
- 4.1 External en internal iterators . . . 53
- 4.2 Intermediate en terminal operations . . . 55
  - 4.2.1 Terminal operation .collect() . . . 55
  - 4.2.2 Intermediate operation .filter() . . . 56
  - 4.2.3 Terminal operation .forEach() . . . 58
  - 4.2.4 Intermediate operation .map() . . . 59
  - 4.2.5 Intermediate operation .sorted() . . . 60
  - 4.2.6 Intermediate operation .distinct() . . . 60
  - 4.2.7 Intermediate operation .limit() . . . 60

4.2.8 Intermediate operation peek() 61
4.2.9 Terminal operation .count() 61
4.3 Intstream, LongStream en DoubleStream 62
4.3.1 sum() 62
4.3.2 range() en rangeClosed() 62
4.3.3 min(), max() en average() 62
4.4 Components and dependency injection 64
4.5 DevOps Tools for Java Developers 65

Chapter 1

Kennismaking met Spring Boot

Learning goals
De junior-collega
1. kan beschrijven wat het Spring framework is
2. kan beschrijven wat Spring Boot is
3. kan de kenmerken van een enterprise toepassing benoemen
4. kan uitleggen wat inversion of control (IoC) is
5. kan uitleggen wat dependency injection is
6. kan beschrijven wat een Spring Bean is
7. kan beschrijven wat de Application Context is
8. kan een nieuwe Spring Boot applicatie aanmaken en opstarten
9. kan Spring Beans aanmaken en gebruiken

1.1 Enterprise toepassingen met Java

Spring Boot is een framework om enterprise toepassingen te bouwen met Java. Enterprise toepassingen zijn toepassingen die bedrijven bouwen of laten bouwen voor hun klanten en medewerkers. Enterprise toepassingen worden ontwikkeld om bedrijfsprocessen te ondersteunen.

Enterprise toepassingen hebben specifieke kenmerken omwille van de complexe aard en specifieke eisen van grootschalige bedrijfssystemen. Enkele belangrijke kenmerken van enterprise toepassingen vanuit het oogpunt van een ontwikkelaar zijn:

- **Schaalbaarheid (scalability):** Enterprise applicaties moeten een groot aantal gebruikers en gegevens aankunnen. Ontwikkelaars moeten de architectuur van de toepassing zodanig ontwerpen dat schalen mogelijk is om een groeiend aantal gebruikers en datavolumes aan te kunnen.
- **Betrouwbaarheid en hoge beschikbaarheid:** Van enterprise applicaties wordt verwacht dat ze 24/7 beschikbaar zijn en snelle responstijden hebben. Ontwikkelaars moeten oplossingen voorzien om problemen en downtime te voorkomen. Ze moeten de betrouwbare werking van de applicatie kunnen garanderen.
- **Beveiliging:** Enterprise applicaties verwerken gevoelige bedrijfsgegevens, waaronder financiële informatie, klantgegevens,... Ontwikkelaars moeten robuuste beveilig-

ingsmaatregelen implementeren, zoals authenticatie, autorisatie, versleuteling van gegevens, audittrails, ..om de gegevens te beschermen tegen ongeautoriseerde toegang.

- **Integratie:** Enterprise applicaties moeten vaak worden geïntegreerd met verschillende bestaande systemen zoals databanken en externe services. Voorbeelden van externe services zijn bijvoorbeeld betaalsystemen en boekhoudpakketten. Ontwikkelaars moeten zorgen voor een veilige en betrouwbare gegevensuitwisseling tussen verschillende enterprise applicaties.
- **Aanpasbaarheid en flexibiliteit:** Enterprise applicaties worden gebruikt door gebruikers uit diverse afdelingen en teams binnen een organisatie, elk met hun eigen unieke workflows en eisen. Ontwikkelaars moeten applicaties bouwen die kunnen worden aangepast en geconfigureerd om aan deze specifieke behoeften te voldoen. Enterprise applicaties ondergaan vaak veranderingen en updates op basis van veranderende zakelijke behoeften. Daarnaast veranderen de workflows en eisen van de gebruikers ook. Ontwikkelaars moeten verandering effectief beheren door versiebeheer, wijzigingstracering en terugrolmechanismen te implementeren. Afhankelijk van de sector moeten enterprise applicaties voldoen aan specifieke wettelijke normen die doorheen de tijd kunnen veranderen.
- **Lange levenscyclus:** Enterprise applicaties hebben meestal een langere levenscyclus dan andere soorten software. Ontwikkelaars moeten zorgen voor voortdurend onderhoud, updates en verbeteringen.
- **Samenwerking en documentatie:** Aangezien meerdere ontwikkelaars en teams werken aan verschillende delen van een enterprise applicatie, zijn duidelijke codeerrichtlijnen, versiebeheer en samenwerkingstools essentieel.
- **Testen en kwaliteitsborging:** Grondig testen is essentieel voor enterprise applicaties om bugs, prestatieproblemen en kwetsbaarheden in de beveiliging te identificeren en aan te pakken. Ontwikkelaars moeten unit test, integratietesten en gebruikersacceptatietesten implementeren. Al deze testen worden geautomatiseerd en bij iedere aanpassing aan de code uitgevoerd.
- **Gebruikerservaring (UX):** Hoewel functionaliteit cruciaal is, is een positieve gebruikerservaring ook belangrijk. Ontwikkelaars moeten streven naar intuïtieve interfaces en workflows zodat de gebruikers productief zijn.

Over het algemeen vereist de ontwikkeling van enterprise applicaties een uitgebreid begrip van bedrijfsprocessen, technische expertise en het vermogen om functionaliteit, performantie, beveiliging en gebruikerservaring in evenwicht te brengen om te voldoen aan de unieke behoeften van grote organisaties.

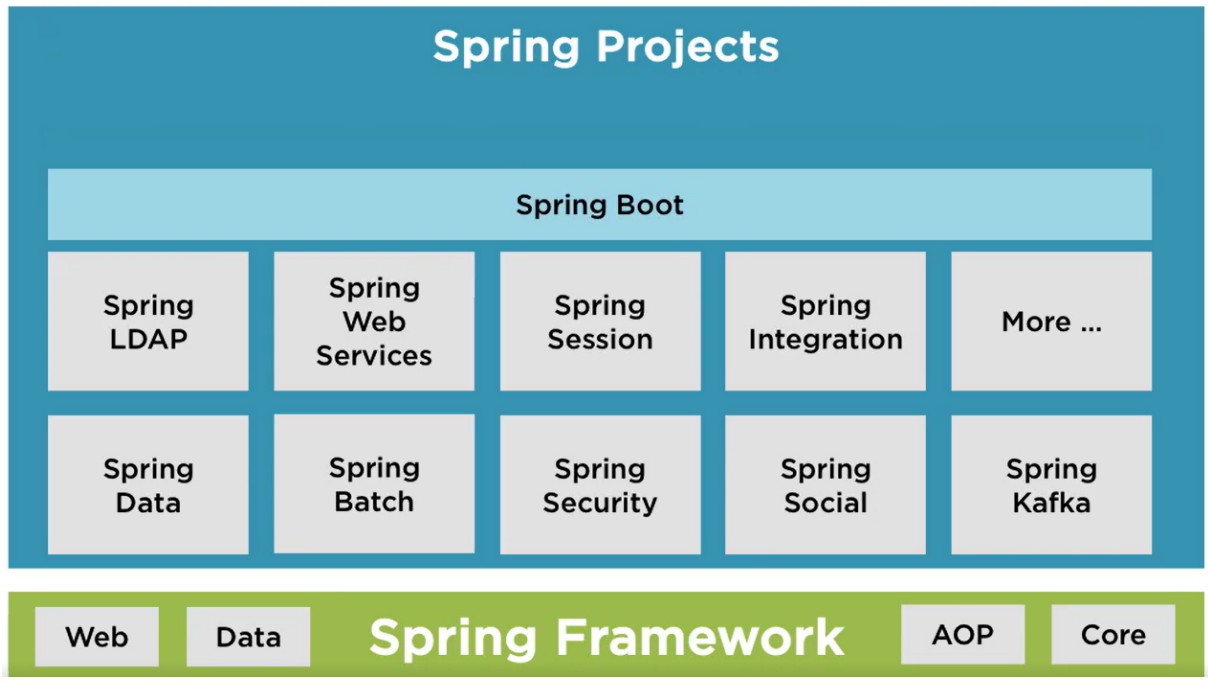
In 1999 besliste Sun Microsystems om de programmeertaal Java uit te bereiden om het ontwikkelen van enterprise applicaties te vergemakkelijken. Met de lancering van J2EE (Java 2 Enterprise Edition) en de applicatie servers om de J2EE-toepassingen op te deployen onstond een schaalbaar en betrouwbaar platform. J2EE, dat werd hernoemd naar Java EE (Java Platform, Enterprise Edition), verwijst naar een verzameling specificaties voor het ontwikkelen van enterprise applicaties. Het bestaat uit een reeks standaarden en API's die ontwikkelaars gebruiken om de enterprise-software te implementeren.

Wanneer we spreken over "J2EE-specificaties" of "Java EE-specificaties", dan verwijst dit naar de reeks specificaties die samen Java EE vormen. Deze specificaties zijn de beschrijvingen van hoe bepaalde componenten en functionaliteiten in een enterprise Java-applicatie moeten worden geïmplementeerd. Sun Microsystems en later Oracle, dat in de 2010 Sun Microsystems overnam, zorgen steeds voor een bruikbare implementatie van de specificaties.

Laten we het versturen van mails als voorbeeld nemen. In bijna iedere enterprise applicatie moet het mogelijk zijn om op een eenvoudige manier mails te versturen. Daarom werden de JavaMail API Design Specifications uitgeschreven. Dit kan je zien als een uitgebreide analyse van de vereisten. De reference implementation is voorzien door Oracle zelf. Naast de reference implementation zijn er nog andere spelers op de markt die hun implementatie, vaak met bijkomende functionaliteiten, aanbeiden. Voor de JavaMailAPI kan je bijvoorbeeld kiezen uit Apache Geronimo JavaMail, WildFly (JBoss) JavaMail, Google App Engine JavaMail, ⋮

Het Spring framework ontstond in 2003 als reactie op de complexiteit van de JEE-specificaties. Sommigen zien Spring als een concurrent van Java EE en zijn moderne opvolger Jakarta EE. Maar als je Spring leert kennen zal je merken dat Spring zorgvuldig individuele specificaties uit Jakarta EE integreert. Waar de ontwikkeling van Enterprise JavaBeans (EJB's) in een JEE toepassing eerder omslachtig was, bood Spring een eenvoudigere benadering met Spring Beans. Toch bleeft het Spring framework complex omwille van de configuratie. Deze configuratie gebeurde initieel door (veel) XML bestanden. Vanaf Spring Framework 6.0 is Java 17+ vereist.

Spring Boot is een open-source Java framework dat gebouwd is boven op het Spring Framework. En het biedt uiteraard alles om enterprise applicaties te ontwikkelen in Java. Het biedt een eenvoudigere en snellere manier een toepassing te creëren, te configureren en uit te voeren.



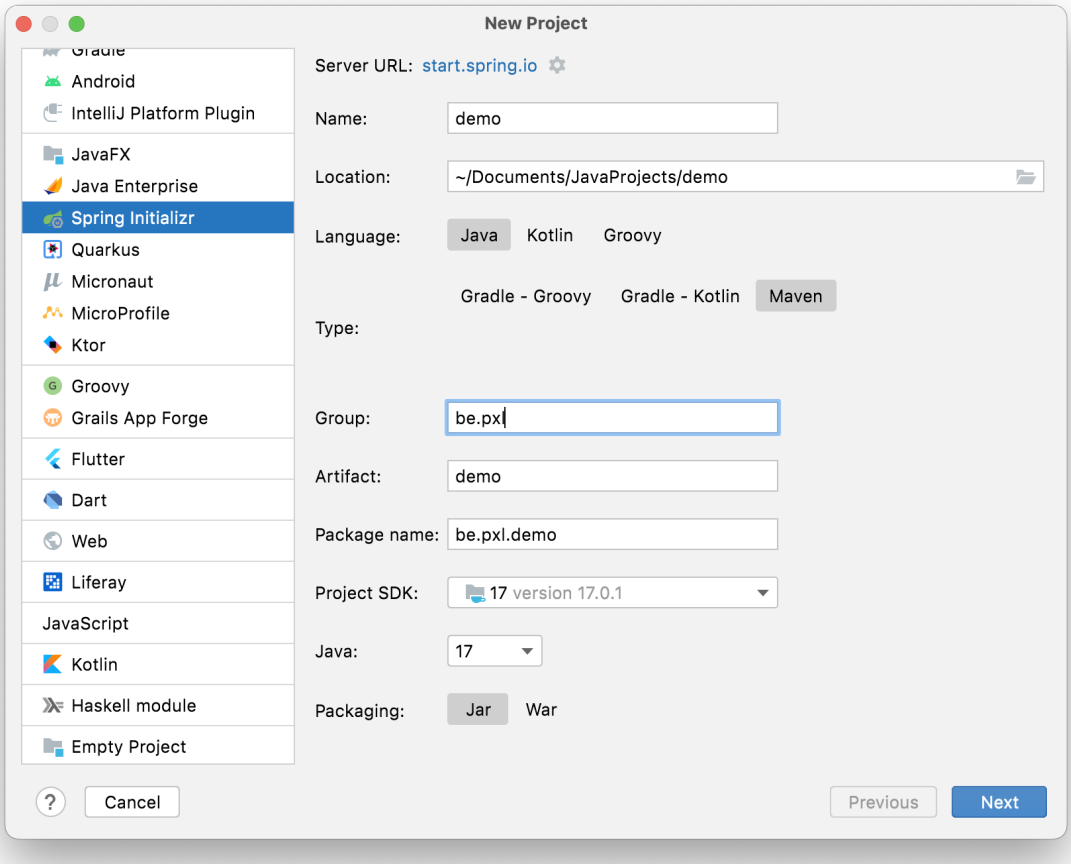
Spring Boot biedt verschillende voordelen voor ontwikkelaars.

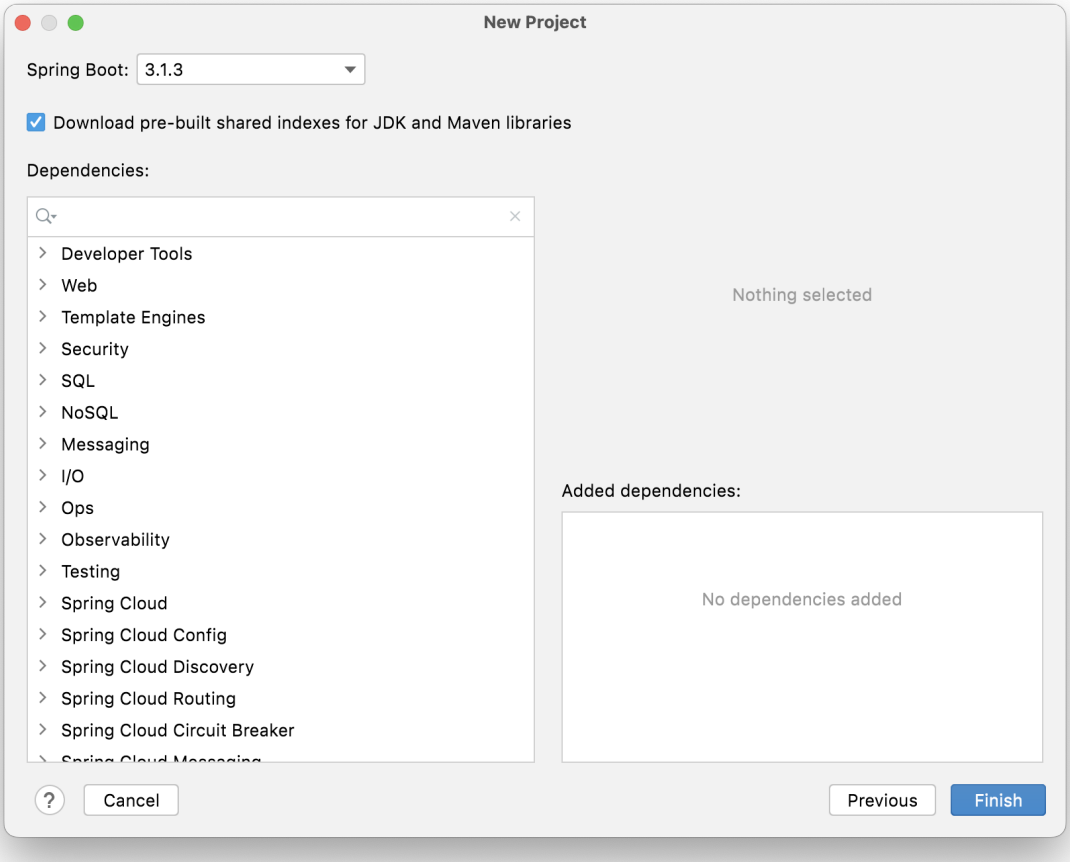
- Gemakkelijker en sneller creëren en implementeren van enterprise applicaties.
- Minder of bijna geen configuratie.
- Eenvoudiger te leren framework.
- Verhoogt de productiviteit.

1.2 Bootstrapping van een eenvoudige Spring Boot applicatie

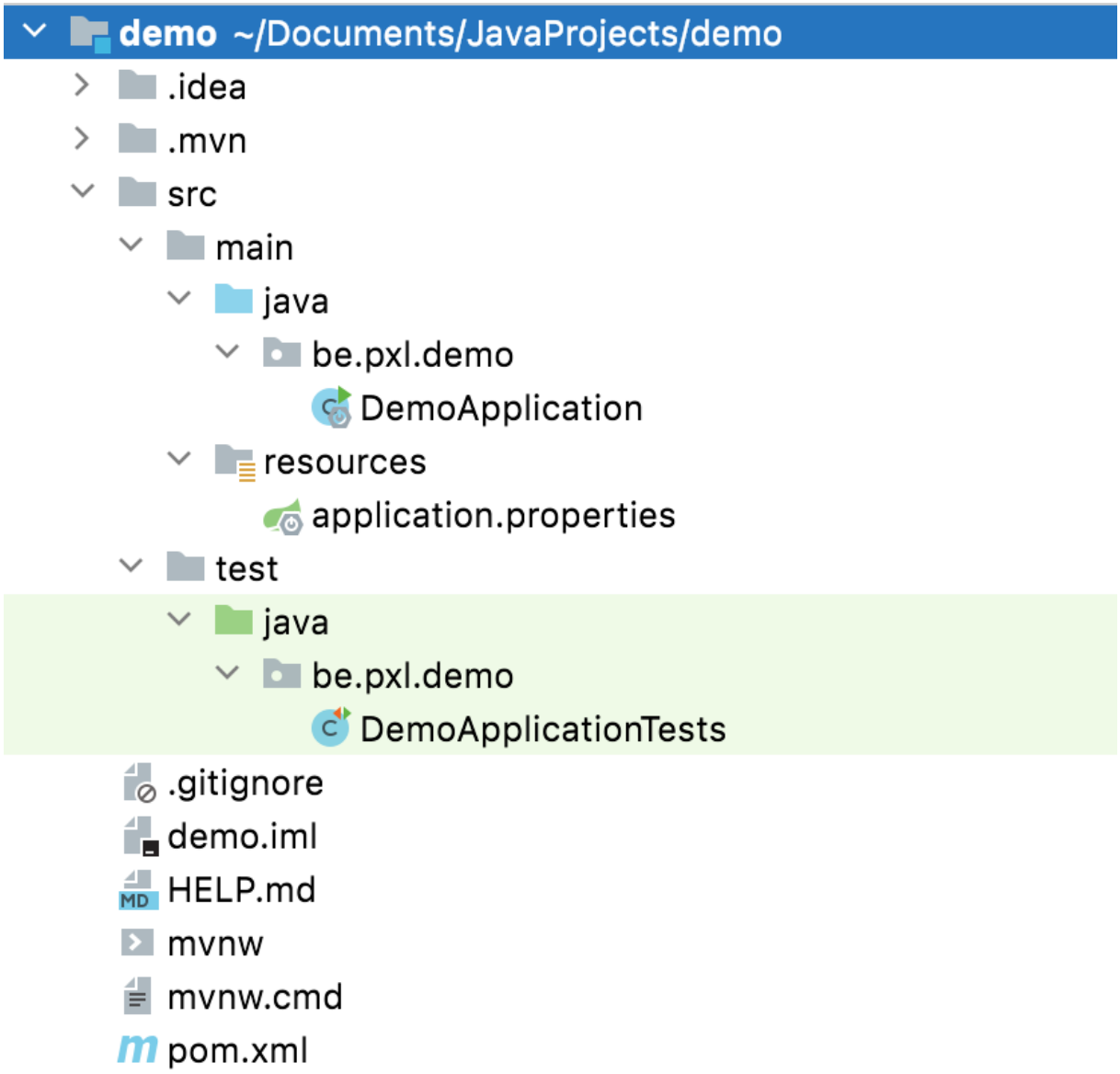
1.2.1 Gebruik van Spring Initializr

Spring Initializr is een webtoepassing waarmee je een Spring Boot-project kunt genereren. De URL voor deze webtoepassing is `https://start.spring.io/`. Je kunt de benodigde configuratie selecteren zoals de buildtool, programmeertaal, versie van het Spring Boot-framework en eventuele afhankelijkheden voor je project. IntelliJ IDEA Ultimate biedt de Spring Initializr-projectwizard die integreert met de Spring Initializr API om je project rechtstreeks in de IDE te genereren en te importeren.





Wanneer je Maven als buildtool kiest krijgt je nieuw project de typische folder-structuur van Maven.



De eigenlijke broncode komt in de src folder. De testen, die we later leren schrijven, komen in de test folder terecht.

Oefening 1.1. Maak het demo-project aan zoals in de screenshots. Je kan de wizard van IntelliJ IDEA Ultimate of de webtoepassing https://start.spring.io/ gebruiken.

1.3 Het demo-project uitvoeren

Het startpunt van een Spring Boot toepassing is de klasse met de main-methode. Deze klasse is geannoteerd met **@SpringBootApplication**. Je vind deze klasse terug in de folder /src/main/java in het package be.pxl.demo. Spring Boot biedt heel veel functionalteit aan in de vorm van annotaties om het werk van de ontwikkelaars te vereenvoudigen.

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

```java
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);
    }
}
```

Start nu de Spring Boot applicatie op.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.1.3)

2023-09-13T09:49:42.340+02:00  INFO 81620 --- [main] be.pxl.demo.DemoApplication: Starting DemoAppl
2023-09-13T09:49:42.343+02:00  INFO 81620 --- main] be.pxl.demo.DemoApplication: No active profile s
2023-09-13T09:49:42.858+02:00  INFO 81620 --- [main] be.pxl.demo.DemoApplication: Started DemoAppli

Process finished with exit code 0
```

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.

1.4 De Maven POM file

POM staat voor Project Object Model. Omdat we voor Maven hebben gekozen als buildtool van onze Spring Boot-toepassing staan hier de project-coördinaten in. Alle dependencies die door ons project worden gebruikt staan hier opgelijst en worden door maven gedownload.

```xml
?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>3.1.3</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</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>
```

De spring-boot-starter-parent is het basisproject dat alle standaardconfiguratie biedt voor op Spring gebaseerde toepassingen. Hier kies je de versie van Spring Boot.

Voor grote projecten is het beheren van afhankelijkheden (dependencies) niet altijd eenvoudig. Spring Boot lost dit probleem op door bepaalde afhankelijkheden samen te bundelen. Deze groepen van afhankelijkheden worden starters genoemd. Alle Spring Boot starters volgen dezelfde richtlijnen voor hun naamgeving. Ze beginnen allemaal met spring-boot-starter-*, waarbij * aangeeft welke functionaliteiten de starter aanbiedt.

spring-boot-starter-test (met scope test) is de starter voor het testen van Spring Boot-toepassingen met o.a. JUnit Jupiter, Hamcrest en Mockito.

1.5 Inversion of Control (IoC) en dependency injection

1.5.1 Spring Beans

Spring Beans zijn componenten die volledig worden beheerd door het Spring Boot framework. Je hoeft zelf geen instanties van deze klassen te maken, Spring Boot genereert de objecten automatisch. Daarnaast beheert Spring Boot ook de objecten. Wanneer een klasse gebruik wil maken van de functionaliteiten van een dergelijke Spring Bean, zal Spring Boot ervoor zorgen dat de instantie van de Spring Bean beschikbaar is voor de betreffende klasse.

We gaan een eerste Spring Bean toevoegen in onze Spring Boot toepassing. De CommandLineRunner interface voorziet de mogelijkheid om een stukje code uit te voeren zodra de Spring Boot applicatie geïnitialiseerd is.

```
@Component
public class WelcomeMessage implements CommandLineRunner {

   @Override
   public void run(String... args) throws Exception {
       System.out.println("Welcome to Java Advanced");
   }
}
```

Zodra je de CommandLineRunner interface implementeert in een klasse kan je de run-methode overschrijven. In de run-methode plaats je de code die uitgevoerd moet worden als de applicatie opstart. Zodra de Spring Boot applicatie opstart worden een aantal initialisatie-fasen doorlopen. Eerst wordt de **application context** aangemaakt en alle Spring beans worden geladen. Als de applicatie volledige is geïnitialiseerd wordt de run-methode van de CommandLineRunner automatisch door het Spring boot framework aangeroepen.

1.5.2 Application context

Inversion of Control (IoC) is één van de basisprincipes van het Spring framework. Bij het software engineering principe Inversion of Control wordt het creëren en beheren van objecten de verantwoordelijkheid van een apart onderdeel binnen het programma. Dit onderdeel noemen we een container. Binnen Spring Boot is deze container een object van de klasse ApplicationContext. We spreken daarom kortweg over de application context om deze container binnen het programma te benoemen.

De application context is een slimme doos waarin alle beans, informatie en instellingen voor de Spring Boot-toepassing worden bewaard. Je kunt de application context beschouwen als het brein van de Spring Boot-applicatie dat alles coördineert en beschikbaar maakt voor de verschillende onderdelen van het programma.

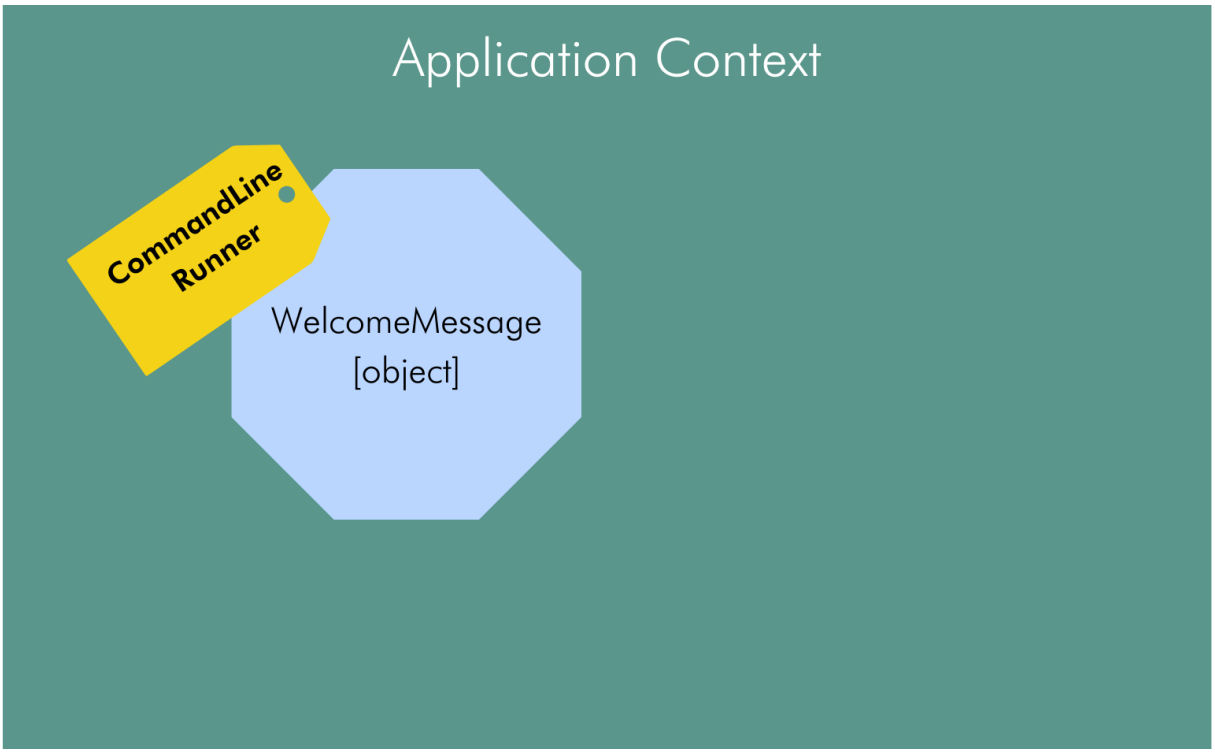


Figure 1.1: De Application Context

Als de Spring Boot-applicatie opstart doorlopen we dus verschillende fasen in de initialisatie. Nadat de Application Context is aangemaakt, worden alle Spring beans geladen en daarna wordt de run-methode van de klassen die de CommandLineRunner interface implemeteren uitgevoerd.

De annotatie @SpringBootApplication zet eigenlijk 3 features van Spring Boot in werking.

- Activeert het auto-configuratie mechanisme van Spring Boot (@EnableAutoConfiguration)
- Activeert het scannen naar componenten met annotatie @Component (en ook @RestController, @Service en @Repository) binnen het package van de Spring Boot applicatie (@ComponentScan)
- Laat toe dat binnen de klasse zelf extra beans worden gedefinieerd die door andere klassen gebruikt kunnen worden. (@Configuration)

We starten eerst met de klasse Pet (huisdier). We willen ons huisdier doorheen de ganse applicatie ter beschikking hebben.

```
package be.pxl.demo;

public class Pet {
    private String name;
    private String breed;

    public Pet(String name, String breed) {
```

```
        this.name = name;
        this.breed = breed;
    }

    public String getBreed() {
        return breed;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Pet{" +
                "name='" + name + '\'' +
                ", breed='" + breed + '\'' +
                '}';
    }
}
```

Nu kunnen we dus in de main-klasse een Spring bean aanmaken voor ons favoriete huisdier.

```
package be.pxl.demo;

import be.pxl.demo.domain.Pet;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.Bean;

@SpringBootApplication // can be replaced by @EnableAutoConfiguration
    ↪ @ComponentScan and @Configuration
public class DemoProjectApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
            ↪ SpringApplication.run(DemoProjectApplication.class, args);
        System.out.println(applicationContext.getApplicationName());
        applicationContext.close();
    }

    @Bean
    public Pet myPet() {
        return new Pet("Scott", "Scotch Collie");
    }
}
```

Het Pet-object wordt bijgehouden in de application context. Elke klasse die nu een Pet-object nodig heeft, kan dit object laten injecteren door de application context. We spreken dan over **dependency injection**. We laten de application context als het ware objecten injecteren in andere objecten. Dat is dus de manier waarop Spring Boot zorgt dat inversion of control mogelijk is.

```
package be.pxl.demo.beans;

import be.pxl.demo.domain.Pet;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class WelcomeMessage implements CommandLineRunner {
    private final Pet myPet;

    @Autowired // annotation not necessary
    public WelcomeMessage(Pet myPet) {
        this.myPet = myPet;
    }

    @Override
   public void run(String... args) throws Exception {
       System.out.println("Welcome to Java Advanced");
        System.out.println(myPet);
   }
}
```

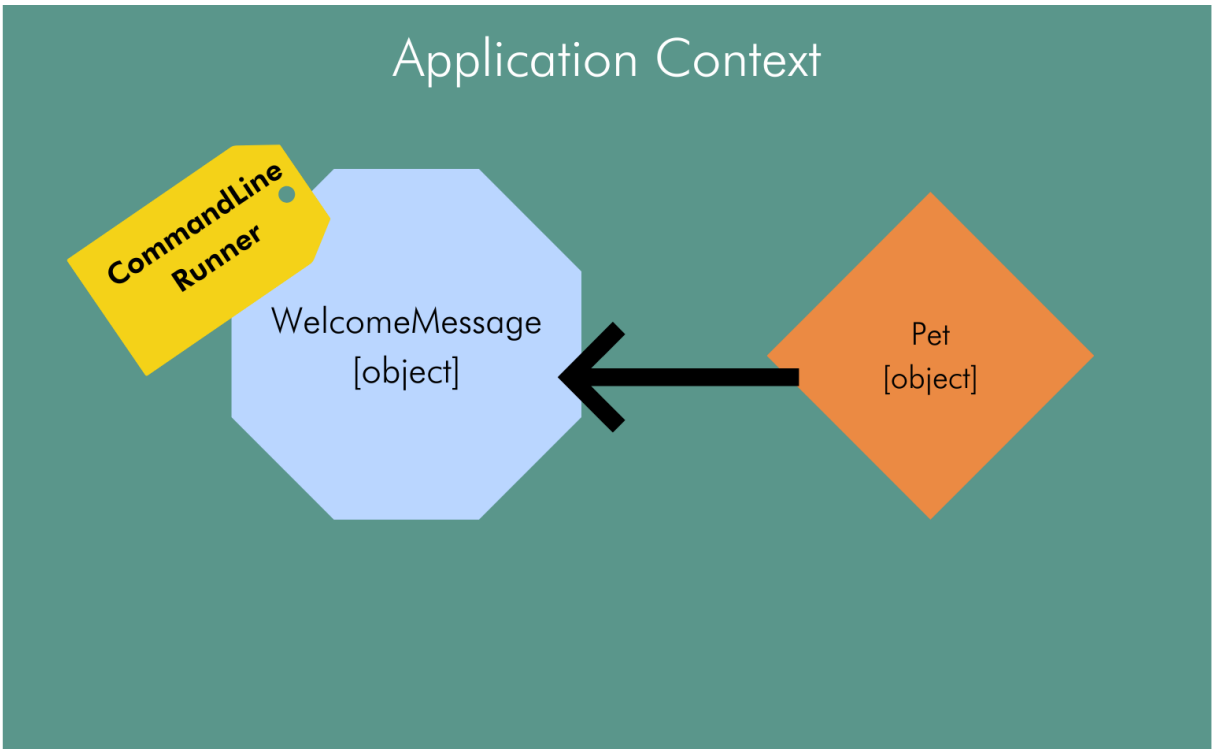


Figure 1.2: De Application Context

Wanneer je het programma uitvoert zal nu het volgende in de console verschijnen:

```
Welcome to Java Advanced
Pet{name='Scott', breed='Scotch Collie'}
```

Om een idee te krijgen van de auto-configuratie die in de Spring Boot applicatie achter de schermen wordt uitgevoerd kan je het loglevel van de applicatie eens aanpassen. Het loglevel van Spring Boot aanpassen doe je door een lijn toe te voegen in application.properties bestand dat je vindt in de folder /src/main/resources.

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

logging.level.org.springframework is één van vele application properties. Een overzicht van beschikbare application properties vind je terug op de documentatie website van Spring Boot https://docs.spring.io/spring-boot/docs/current/reference/html/application-properti
html.

In de logging vind je nu tussen de vele lijnen onderstaande regels:

```
... Creating shared instance of singleton bean 'welcomeMessage'
... Creating shared instance of singleton bean 'myPet'
... Autowiring by type from bean name 'welcomeMessage' via constructor to bean named 'myPet'
```

We geven nog een tweede voorbeeld van dependency injection.

```
package be.pxl.demo.service;

import org.springframework.stereotype.Component;
```

```java
@Component
public class WeatherService {
   public void printWeather() {
      System.out.println("The weather is sunny with a 20% chance of rain");
   }
}
```

We laten nu de instantie van de WeatherService injecteren in onze CommandLineRunnerWelcomeMessage.

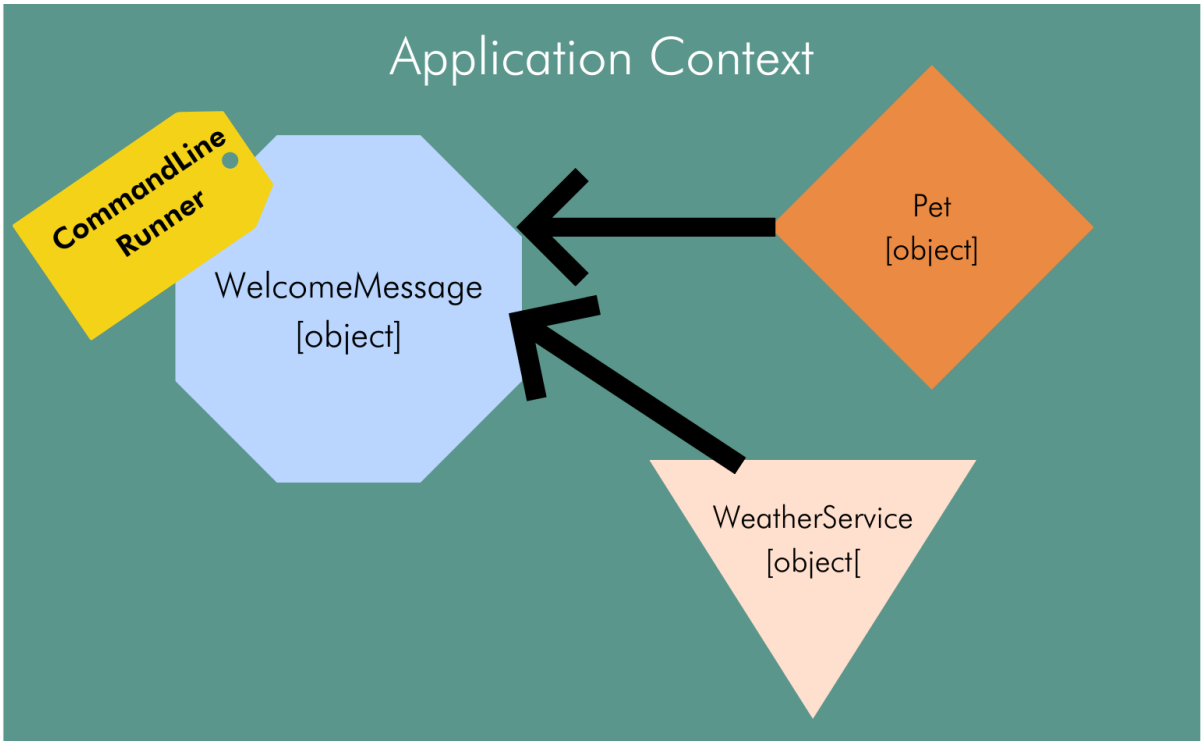


Figure 1.3: De Application Context

```java
package be.pxl.demo.beans;

import be.pxl.demo.domain.Pet;
import be.pxl.demo.service.WeatherService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class WelcomeMessage implements CommandLineRunner {

    private final Pet myPet;
    private final WeatherService weatherService;
```

```
    @Autowired // annotation not necessary
    public WelcomeMessage(Pet myPet, WeatherService weatherService) {
        this.myPet = myPet;
        this.weatherService = weatherService;
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Welcome to Java Advanced");
        System.out.println(myPet);
        weatherService.printWeather();
    }
}
```

Chapter 2

REST

Learning goals
De junior-collega
1. kan beschrijven wat een RESTful web applicatie is
2. kan een POST, GET, PUT en DELETE-verzoek afhandelen in Spring Boot
3. kan beschrijven wat spring-boot-starter-web is
4.

2.1 HTTP-verzoekmethoden

Spring Boot is een goede keuze als je een REST API (of voluit een RESTful web API) wilt ontwikkelen in Java.

REST (Representational State Transfer) is een gestandaardiseerde manier om communicatie tussen verschillende softwaretoepassingen over het internet mogelijk te maken. In een RESTful web applicatie wordt de functionaliteit van de applicatie beschikbaar gesteld als resources, die kunnen worden geïdentificeerd door URI's (Uniform Resource Identifiers). Gebruikers en andere applicaties kunnen met deze resources communiceren via standaard HTTP-verzoekmethoden (HTTP-request, HTTP-method of HTTP-verb). In essentie is HTTP het transportprotocol dat wordt gebruikt om gegevens over te dragen, terwijl REST de verzameling van ontwerpprincipes is die bepalen hoe die gegevens moeten worden georganiseerd en benaderd.

- **GET**: Het GET-verzoek wordt gebruikt om gegevens op te halen van een specifieke resource.

  Voorbeeld URI: GET /api/products/123

  Dit verzoek haalt informatie op over het product met ID 123.

- **POST**: Het POST-verzoek wordt gebruikt om nieuwe gegevens naar een resource te verzenden. Het wordt vaak gebruikt voor het maken van nieuwe resources of het toevoegen van gegevens aan een bestaande resource.

  Voorbeeld URI: POST /api/products

  Dit verzoek voegt een nieuw product toe aan de lijst van producten.

- **PUT**: Het PUT-verzoek wordt gebruikt om gegevens bij te werken voor een specifieke resource of om een nieuwe resource te maken als deze niet bestaat. Het is idempotent, wat betekent dat meerdere PUT-verzoeken hetzelfde resultaat opleveren.

  Voorbeeld URI: PUT /api/products/123

  Dit verzoek bijwerken de informatie van het product met ID 123.

- **DELETE**: Het DELETE-verzoek wordt gebruikt om een resource te verwijderen of te deactiveren.

  Voorbeeld URI: DELETE /api/products/123

  Dit verzoek verwijdert het product met ID 123 uit de lijst van producten.

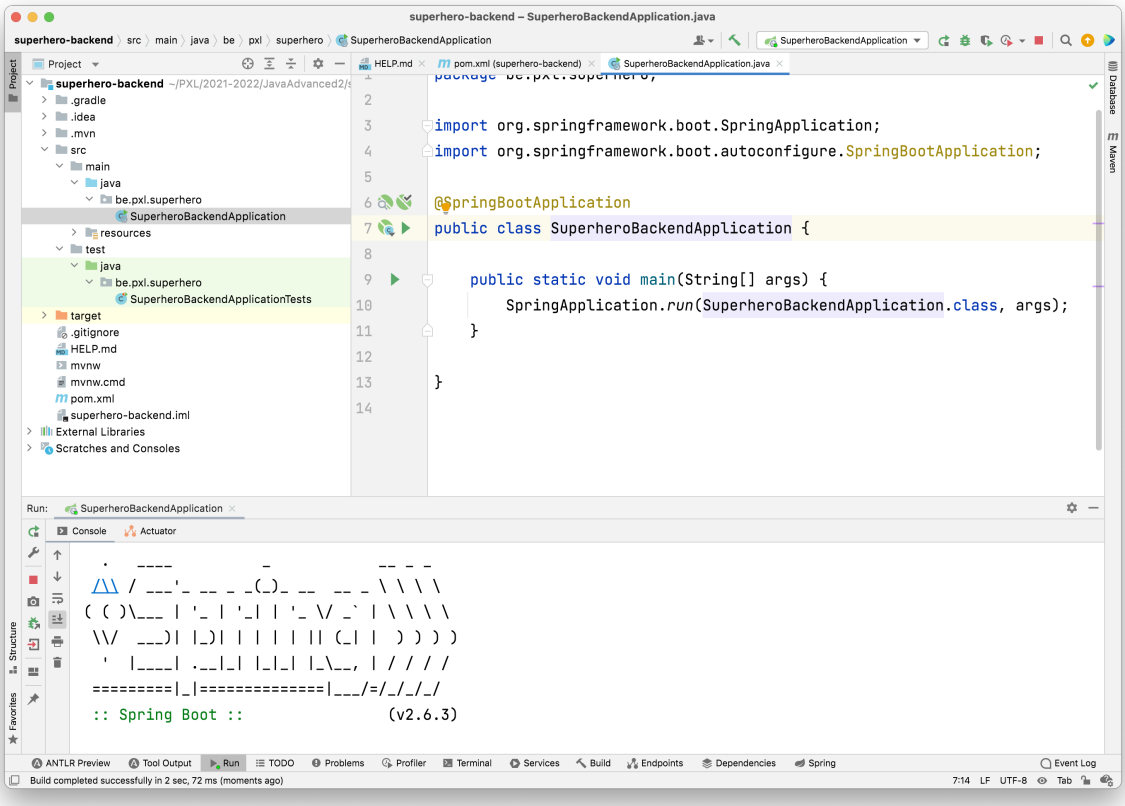
2.2 Spring Boot Starter Web

Spring Boot Starter Web is de verzameling van alle bibliotheken (third party libraries) die we nodig hebben om RESTful web applicaties te bouwen. De verzameling bestaat ondermeer uit Spring MVC, REST, jackson en Tomcat. Apache Tomcat is een populaire open source web server voor Java toepassingen. Als je de dependency spring-boot-start-web toevoegt, start de Tomcat web server op zodra je de Spring boot applicatie opstart. We spreken van de "embedded web server" in Spring boot. Je kan er ook voor kiezen om één van de alternatieve web servers te gebruiken zoals jetty of undertow. Jackson is een populaire library om Java-objecten naar JSON te converteren en vice versa.

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

Je voegt de dependency toe in het bestand pom.xml.

Start nu de Spring Boot applicatie opnieuw op.



Momenteel kan de Spring Boot applicatie enkel een whitelabel error pagina tonen. De error pagina krijg je te zien als je een GET-request uitvoert voor URL `http://localhost:8080`.

Poort 8080 is de default poort van de Tomcat webserver. Als deze poort al gebruikt wordt door een ander programma, krijg je de volgende foutmelding te zien.

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

Description:

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

De poortnummer kan aangepast worden in het bestand application.properties. Je gebruikt de eigenschap **server.port** om de gewenste poortnummer te kiezen.

```
server.port=8081
```

2.3 De RestController

Spring boot heeft een annotatie voorzien voor de Spring bean die verantwoordelijk is voor het afhandelen van HTTP requests nl. @RestController. Spring boot heeft ook een annotatie @Controller, maar de @RestController zorgt ervoor dat het respons op het HTTP-request automatisch wordt omgezet (geserialiseerd) naar JSON of XML en wordt teruggestuurd naar de client.

```
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
@RequestMapping("/greetings")
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()));
    }
}
```

De @RestController markeert de klasse GreetingController als een REST-controller. De annotatie @RequestMapping("/greetings") specificeert het basispad voor alle requests die door deze controller worden afgehandeld. @GetMapping("/hello") geeft aan dat de goGreeting-methode wordt uitgevoerd wanneer een HTTP GET-verzoek wordt gemaakt naar het pad "/greetings/hello". Het resultaat van deze methode wordt automatisch omgezet in tekst en teruggestuurd als de respons.

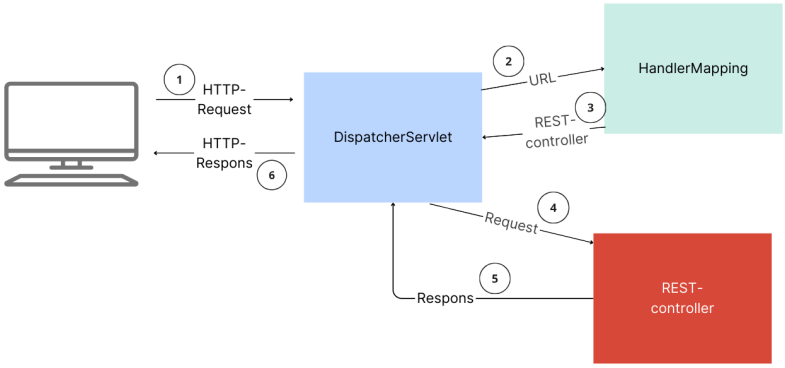


Figure 2.1: Een REST-verzoek afhandelen

De component van Spring Boot die verantwoordelijk is dat een HTTP-request door de juiste REST-controller wordt afgehandeld is de DispatcherServlet. De DispatcherServlet is onderdeel van Spring MVC. De DispatcherServlet bepaalt welke controller een HTTP-request moet afhandelen, hij geeft het request door aan de juiste controller en verwerkt het respons van de controller om een HTTP-respons terug te sturen naar de client.

Om te achterhalen welke REST-controller verantwoordelijk is om een HTTP-request af te handelen raadpleegt de DispatcherServlet de HandlerMapping. De HandlerMapping is als het ware een kaart die URL's koppelt aan specifieke controllerklassen en methoden. Op basis van de URL in het binnenkomende request bepaalt de DispatcherServlet welke controllerklasse en methode verantwoordelijk zijn voor het afhandelen van het request.

Oefening 2.1. Maak het package *be.pxl.demo.api*. Voeg de klasse **GreetingController** toe het package. Start de Spring Boot applicatie en open de URL `http://localhost:8080/greetings/hello` in een browser. Voeg in de REST-controller een methode toe met de URI GET /greetings/daytime die de huidige dag en het tijdstip teruggeeft in het formaat 'Maandag 18 september 2023'.

2.4 MusicPlaylist

We gaan een nieuwe Spring Boot toepassing aanmaken waarmee we een muziek playlist beheren.

Oefening 2.2. Maak een nieuwe Spring boot toepassing MusicPlaylist. We gebruiken Spring MVC om een RESTful web applicatie te maken.

2.4.1 Een liedje toevoegen aan een playlist

POST **/playlist/songs**

Add a new song to the playlist.

Body application/json

```
{
  "title": "Hello",
  "artist": "Adele",
  "duration_seconds": 293,
  "genre": "POP"
}
```

Response application/json

200 ok

Om te beginnen hebben we de klasse Song nodig.

```
public class Song {
    private String title;
    private String artist;
    @JsonProperty("duration_seconds")
    private int durationSeconds;
    private Genre genre;

    // Default constructor
    public Song() {
    }

    // Parameterized constructor
    public Song(String title, String artist, int durationSeconds, Genre
        ↪ genre) {
        this.title = title;
        this.artist = artist;
        this.durationSeconds = durationSeconds;
```

```
        this.genre = genre;
    }

    // Getter and setter methods
    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist() {
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public int getDurationSeconds() {
        return durationSeconds;
    }

    public void setDurationSeconds(int durationSeconds) {
        this.durationSeconds = durationSeconds;
    }

    public Genre getGenre() {
        return genre;
    }

    public void setGenre(Genre genre) {
        this.genre = genre;
    }

    @Override
    public String toString() {
        return "Song{" +
                "title='" + title + '\'' +
                ", artist='" + artist + '\'' +
                ", durationSeconds=" + durationSeconds +
                ", genre='" + genre + '\'' +
                '}';
    }
}
```

Nu gaan we de REST-controller implementeren. We gaan hierin een methode voorzien

die een POST op de URI /playlist/songs kan afhandelen. Initieel gaan we enkel de titel in de loggegevens tonen.

```
@RestController
@RequestMapping("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
    }
}
```

De liedjes die aan de playlist worden toegevoegd willen we bijhouden. Later zullen we ze wegschrijven in een databank, maar voorlopig gaan we ze bijhouden in een lijst. Om dit mogelijk te maken gaan we een nieuwe Spring Bean toevoegen: de MusicPlaylistService.

```
package be.pxl.demo;

import be.pxl.demo.domain.Song;
import org.springframework.stereotype.Service;

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

@Service
public class MusicPlaylistService {
    private final List<Song> myPlaylist = new ArrayList<>();

    public void addSong(Song song) {
        myPlaylist.add(song);
    }
}
```

De MusicPlaylistService wordt geannoteerd met @Service . In onze Spring Boot applicaties gaan we steeds de business-logica implementeren in de service-laag. De @Service annotatie wordt gebruikt voor de componenten (Spring Beans) in de service-laag. Wanneer onze Spring Boot applicatie opstart wordt er exact één instantie van de MusicPlaylistService aangemaakt in de ApplicationContext en deze instantie wordt tijdens de volledige levensduur van de toepassing gebruikt. Dit noemen we de scope van de service en de default scope noemen we **singleton**. Dit betekent dat we één enkele, gedeelde playlist hebben voor alle gebruikers.

Nu gaan we de MusicPlaylistService beschikbaar maken in de MusicPlaylistController. We maken gebruik van **constructor injection**. Zodra de instantie van de MusicPlaylistController door Spring Boot wordt aangemaakt, zal er eerst gezorgd worden dat de instantie MusicPlaylistService aangemaakt wordt. Deze instantie wordt dan achter de schermen meegegeven aan de constructor van de MusicPlaylistController. Zo kan ons MusicPlaylistController-object het MusicPlaylistService-object gebruiken. Omdat er maar één constructor is, is de annotatie @Autowired eigenlijk overbodig.

```
@RestController
@RequestMapping("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);
    private final MusicPlaylistService musicPlaylistService;

    @Autowired
    public MusicPlaylistController(MusicPlaylistService
        ↪ musicPlaylistService) {
        this.musicPlaylistService = musicPlaylistService;
    }

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
        musicPlaylistService.addSong(song);
    }
}
```

Test nu het POST-verzoek. Je kan Postman, Insomnia of een andere tool gebruiken om een POST-verzoek naar de toepassing te sturen. De toegevoegde liedjes gaan uiteraard verloren wanneer je de toepassing herstart.

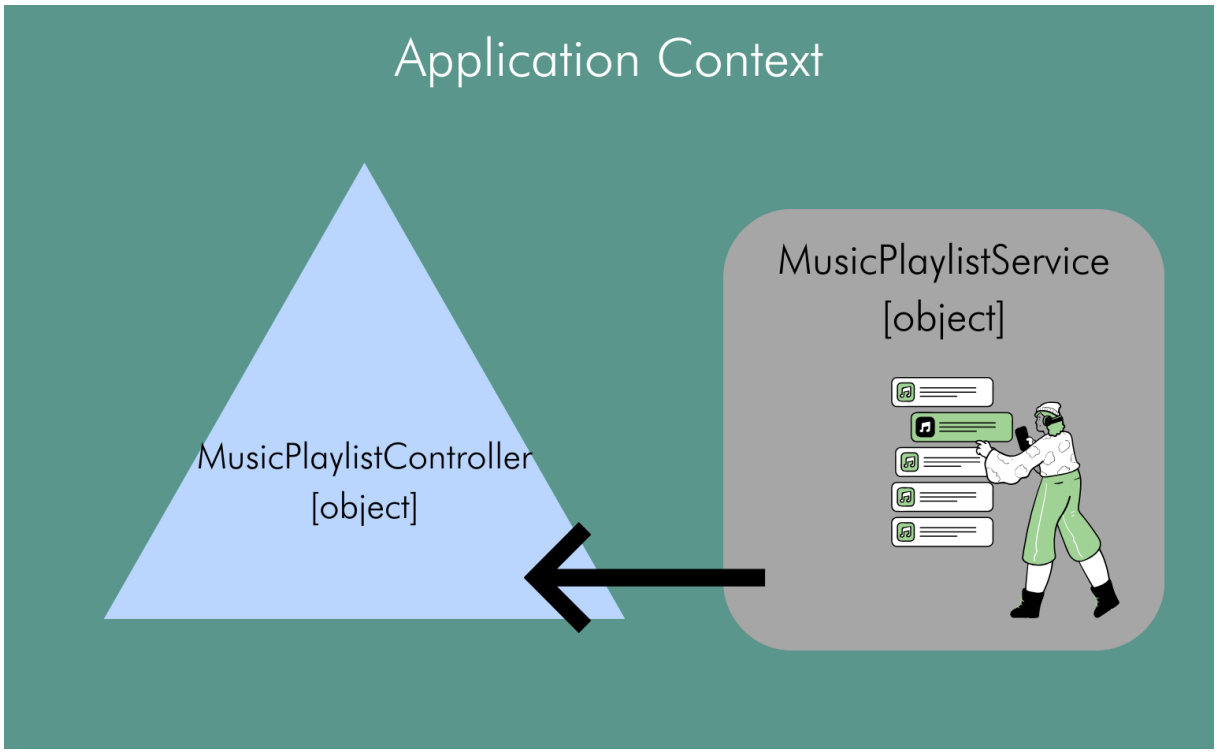


Figure 2.2: Spring Beans in de Application Context

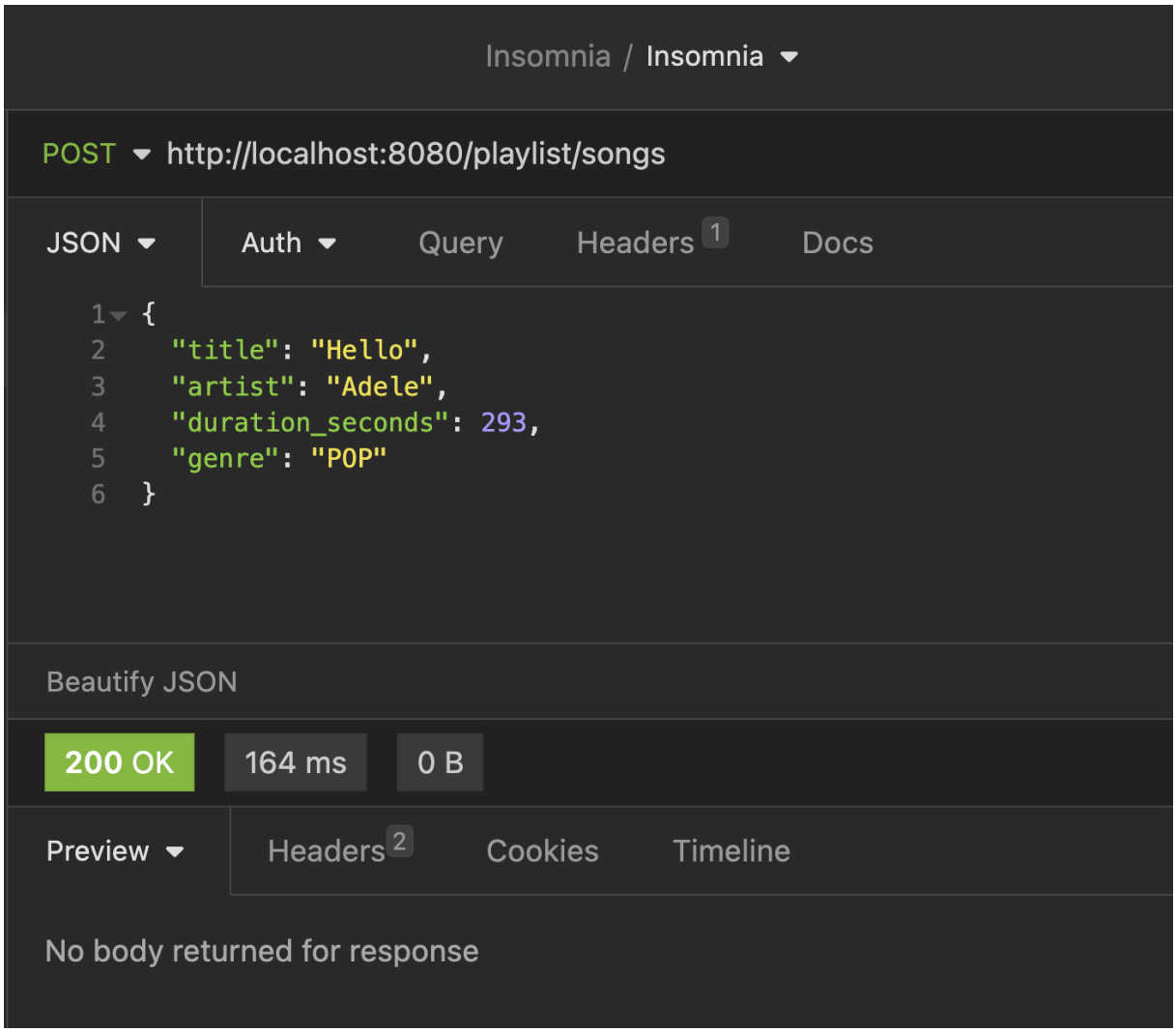


Figure 2.3: POST-verzoek met Insomnia

2.4.2 De playlist opvragen

GET **/playlist/songs**

Retrieve all songs from the playlist

Parameter

no parameter

Body application/json

Response application/json

200 ok


```
[
    {
        "title": "Hello",
        "artist": "Adele",
        "genre": "POP",
        "duration_seconds": 293
    },
    {
        "title": "Shape of You",
        "artist": "Ed Sheeran",
        "genre": "POP",
        "duration_seconds": 233
    },
    {
        "title": "Umbrella",
        "artist": "Rihanna",
        "genre": "RNB",
        "duration_seconds": 264
    }
]
```


```
package be.pxl.demo;

import be.pxl.demo.domain.Song;
import org.springframework.stereotype.Service;

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

@Service
public class MusicPlaylistService {
    private final List<Song> myPlaylist = new ArrayList<>();
```

```
    public void addSong(Song song) {
        myPlaylist.add(song);
    }

    public List<Song> getSongs() {
        return myPlaylist;
    }
}
```

```
package be.pxl.demo.controller;

import be.pxl.demo.MusicPlaylistService;
import be.pxl.demo.domain.Song;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
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("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);
    private final MusicPlaylistService musicPlaylistService;

    @Autowired
    public MusicPlaylistController(MusicPlaylistService
        ↪ musicPlaylistService) {
        this.musicPlaylistService = musicPlaylistService;
    }

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
        musicPlaylistService.addSong(song);
    }

    @GetMapping
    public List<Song> getSongs() {
```

```
        return musicPlaylistService.getSongs();
    }
}
```

2.4.3 Liedjes van één genre

GET **/playlist/songs/{genre}**

Retrieve all songs from the playlist with the given genre

Parameter

genre the requested genre

Body application/json

Response application/json

200 ok

```
[
    {
        "title": "Hello",
        "artist": "Adele",
        "genre": "POP",
        "duration_seconds": 293
    },
    {
        "title": "Shape of You",
        "artist": "Ed Sheeran",
        "genre": "POP",
        "duration_seconds": 233
    }
]
```

```
package be.pxl.demo.controller;

import be.pxl.demo.MusicPlaylistService;
import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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.RequestBody;
```

```java
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);
    private final MusicPlaylistService musicPlaylistService;

    @Autowired
    public MusicPlaylistController(MusicPlaylistService
        ↪ musicPlaylistService) {
        this.musicPlaylistService = musicPlaylistService;
    }

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
        musicPlaylistService.addSong(song);
    }

    @GetMapping
    public List<Song> getSongs() {
        return musicPlaylistService.getSongs();
    }

    @GetMapping("{genre}")
    public List<Song> getSongs(@PathVariable Genre genre) {
        return musicPlaylistService.getSongsByGenre(genre);
    }
}
```

```java
package be.pxl.demo;

import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.springframework.stereotype.Service;

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

@Service
```

```
public class MusicPlaylistService {
    private final List<Song> myPlaylist = new ArrayList<>();

    public void addSong(Song song) {
        myPlaylist.add(song);
    }

    public List<Song> getSongs() {
        return myPlaylist;
    }

    public List<Song> getSongsByGenre(Genre genre) {
        List<Song> response = new ArrayList<>();
        for (Song song : myPlaylist) {
            if (song.getGenre() == genre) {
                response.add(song);
            }
        }
        return response;
    }
}
```

2.4.4 Gegevens van een liedje aanpassen

Als je een nieuwe song in de playlist toevoegt, dan wordt deze steeds achteraan in de lijst toegevoegd. We kunnen nu de gegevens van het liedje op een gegeven positie in de lijst gaan overschrijven of aanpassen. De index die we meegeven is een waarde van 0 tot 1 minder dan de lengte van de lijst. Later zullen we zien hoe we een duidelijke foutboodschap kunnen geven aan de client als een foutieve index-waarde wordt gegeven.

Om de gegevens van een liedje aan te passen gebruiken we een PUT-verzoek. Je moet steeds alle gegevens van het liedje meegeven in de requestbody.

PUT **/musicplaylist/songs{index}**

Update the song at the given index.

Parameter

code unique identification of a house

Body application/json

```
{
        "title": "Hello",
        "artist": "Adele",
        "genre": "POP",
        "duration_seconds": 293
}
```

Response application/json

200 ok

```
package be.pxl.demo.controller;

import be.pxl.demo.MusicPlaylistService;
import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
```

```
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);
    private final MusicPlaylistService musicPlaylistService;

    @Autowired
    public MusicPlaylistController(MusicPlaylistService
        ↪ musicPlaylistService) {
        this.musicPlaylistService = musicPlaylistService;
    }

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
        musicPlaylistService.addSong(song);
    }

    @GetMapping
    public List<Song> getSongs() {
        return musicPlaylistService.getSongs();
    }

    @GetMapping("/{genre}")
    public List<Song> getSongs(@PathVariable Genre genre) {
        return musicPlaylistService.getSongsByGenre(genre);
    }

    @PutMapping("/{index}")
    public void updateSong(@PathVariable int index, @RequestBody Song song)
        ↪ {
        musicPlaylistService.updateSong(index, song);
    }
}
```

```
package be.pxl.demo;

import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.springframework.stereotype.Service;

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

@Service
public class MusicPlaylistService {
    private final List<Song> myPlaylist = new ArrayList<>();

    public void addSong(Song song) {
```

```
        myPlaylist.add(song);
    }

    public List<Song> getSongs() {
        return myPlaylist;
    }

    public List<Song> getSongsByGenre(Genre genre) {
        List<Song> response = new ArrayList<>();
        for (Song song : myPlaylist) {
            if (song.getGenre() == genre) {
                response.add(song);
            }
        }
        return response;
    }

    public void updateSong(int index, Song song) {
        myPlaylist.set(index, song);
    }

    public void deleteSong(int index) {
        myPlaylist.remove(index);
    }
}
```

We vervangen het liedje op de opgegeven index door een nieuw Song-object met de aangepaste gegevens.

2.4.5 Een liedje verwijderen

Om een liedje op een opgegeven index uit de playlist te verwijderen gaan we een DELETE-verzoek implementeren.

DELETE **/musicplaylist/songs/{index}**

Delete the song at the given index.

Parameter

index position of the song to be deleted

Response application/json

200 ok

```
package be.pxl.demo.controller;
```

```java
import be.pxl.demo.MusicPlaylistService;
import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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("/playlist/songs")
public class MusicPlaylistController {

    private static final Logger LOGGER =
        ↪ LoggerFactory.getLogger(MusicPlaylistController.class);
    private final MusicPlaylistService musicPlaylistService;

    @Autowired
    public MusicPlaylistController(MusicPlaylistService
        ↪ musicPlaylistService) {
        this.musicPlaylistService = musicPlaylistService;
    }

    @PostMapping
    public void addSong(@RequestBody Song song) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("Adding song [" + song.getTitle() + "]");
        }
        musicPlaylistService.addSong(song);
    }

    @GetMapping
    public List<Song> getSongs() {
        return musicPlaylistService.getSongs();
    }

    @GetMapping("/{genre}")
    public List<Song> getSongs(@PathVariable Genre genre) {
        return musicPlaylistService.getSongsByGenre(genre);
    }
```

```
    @PutMapping("/{index}")
    public void updateSong(@PathVariable int index, @RequestBody Song song)
        ↪ {
        musicPlaylistService.updateSong(index, song);
    }

    @DeleteMapping("/{index}")
    public void updateSong(@PathVariable int index) {
        musicPlaylistService.deleteSong(index);
    }
}
```

```
package be.pxl.demo;

import be.pxl.demo.domain.Genre;
import be.pxl.demo.domain.Song;
import org.springframework.stereotype.Service;

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

@Service
public class MusicPlaylistService {
    private final List<Song> myPlaylist = new ArrayList<>();

    public void addSong(Song song) {
        myPlaylist.add(song);
    }

    public List<Song> getSongs() {
        return myPlaylist;
    }

    public List<Song> getSongsByGenre(Genre genre) {
        List<Song> response = new ArrayList<>();
        for (Song song : myPlaylist) {
            if (song.getGenre() == genre) {
                response.add(song);
            }
        }
        return response;
    }

    public void updateSong(int index, Song song) {
        myPlaylist.set(index, song);
    }
```

```
    public void deleteSong(int index) {
        myPlaylist.remove(index);
    }
}
```

Oefening 2.3. **Huizenjacht** We maken een Spring Boot toepassing om het aanbod op de huizenmarkt te beheren. Ontwikkel een RESTful web toepassing met onderstaande endpoints. Je voorziet een component HouseService met één enkele, gedeelde lijst van woningen voor alle gebruikers.
Een huis wordt voorgesteld als een resource met volgende eigenschappen:

- `code` (string): Unieke identificatie van het huis.
- `name` (string): Naam of beschrijving van het huis.
- `status` (enum): Status FOR_SALE of SOLD.
- `city` (string): Locatie van het huis.
- `price` (double): Prijs van het huis.

2.5 REST Endpoints

POST **/houses**

Create a new house.

Parameter

no parameter

Body application/json

```
{
  "code": "HAS_001",
  "name": "Beautiful house in the city",
  "city": "Hasselt",
  "price": 250000
}
```

Response application/json

200 ok

PUT **/houses/{code}**

Update the data (status, price,...) of the house with the given code.

Parameter

code unique identification of a house

Body application/json

```
{
  "status": "SOLD"
  "name": "Beautiful house in the city",
  "price": 320000
}
```

Response application/json

200 ok

GET **/houses**

Retrieve all houses.

Parameter

no parameter

Response application/json

200 ok

```
[
  {
    "code": "GNK_001",
    "name": "Beautiful house in the city",
    "status": "SOLD",
     "city": "Genk",
    "price": 250000
  },
  {
    "code": "HAS_003",
    "name": "Cozy bungalow",
    "status": "FOR_SALE",
     "city": "Hasselt",
    "price": 180000
  }
]
```

DELETE **/houses/{code}**

Delete the house with the given code.

Parameter	
code	unique identification of a house

Response	application/json
200	ok