Hogeschool PXL
PXL-Digital

Java Advanced
An Introduction to Spring Boot

Author
Nele Custers

September 23, 2023

Contents

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.

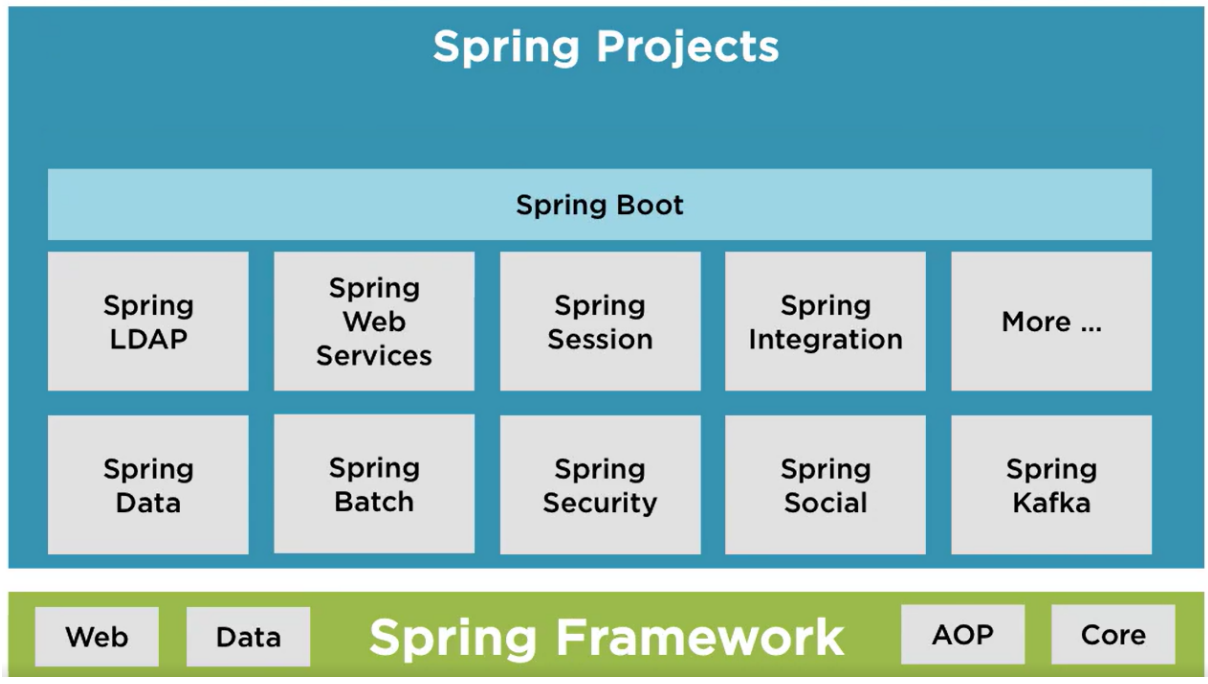
In 1999 besliste Sun Microsystems om de programmeertaal Java uit te bereiden om het ontwikkelen van enterprise applicaties te vergemakkelijken. Met de lancering van J2EE (Java 2 Enterprise Edition) en de applicatie servers om de J2EE-toepassingen op te deployen onstond een schaalbaar en betrouwbaar platform. 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.



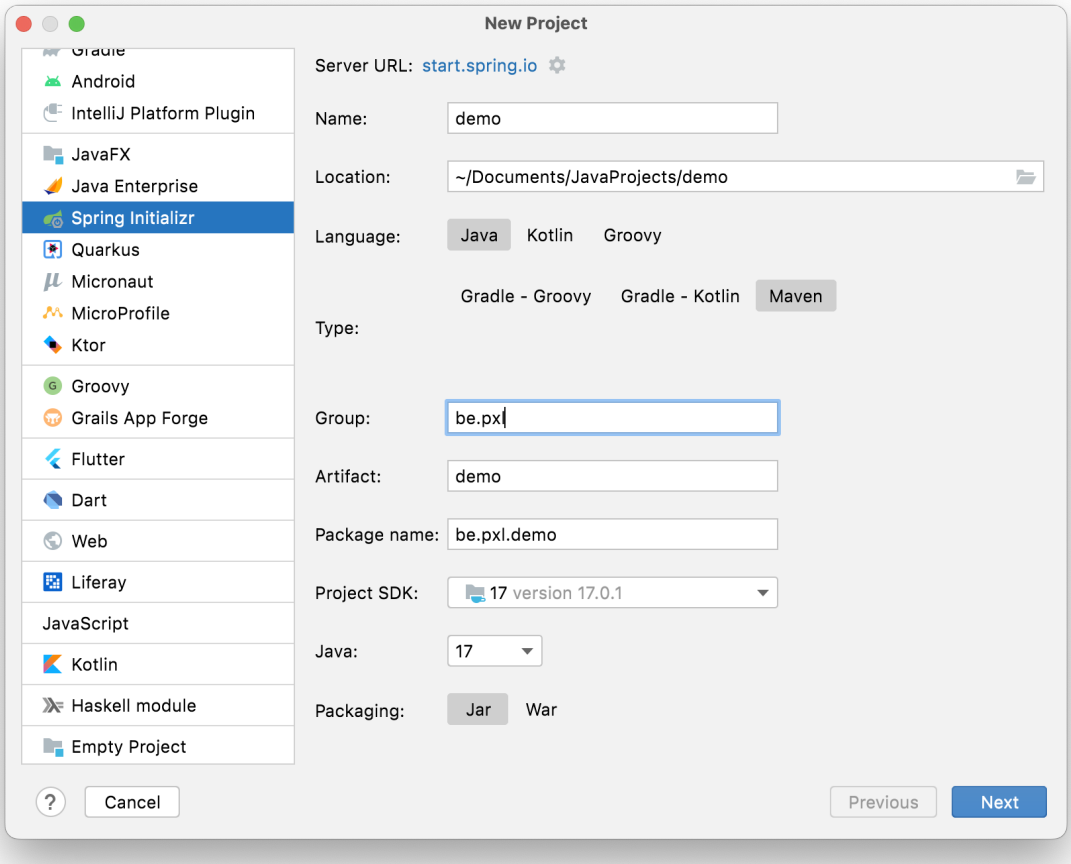
Spring Boot biedt verschillende voordelen voor ontwikkelaars.

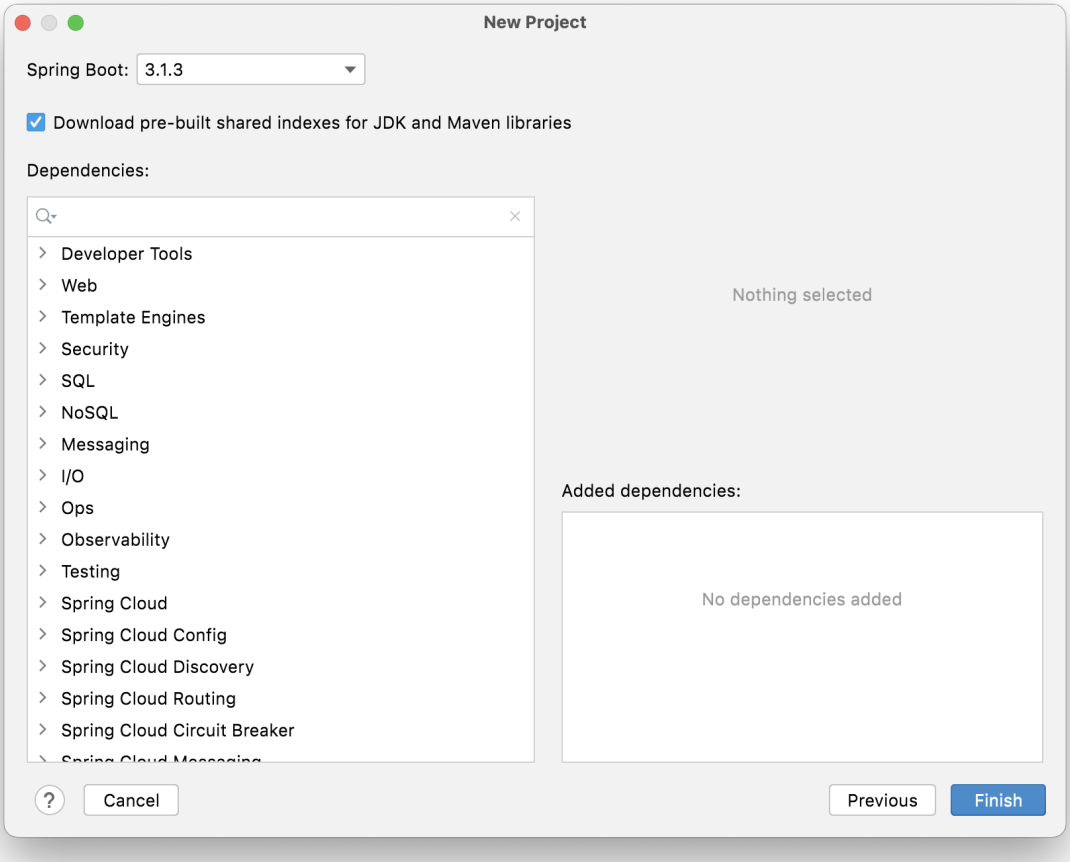
- Gemakkelijker en sneller creëren en implementeren van enterprise applicaties.
- Minder of bijna geen configuratie.
- Eenvoudiger te leren framework.
- Verhoogt de productiviteit.

1.2 Bootstrapping van een eenvoudige Spring Boot applicatie

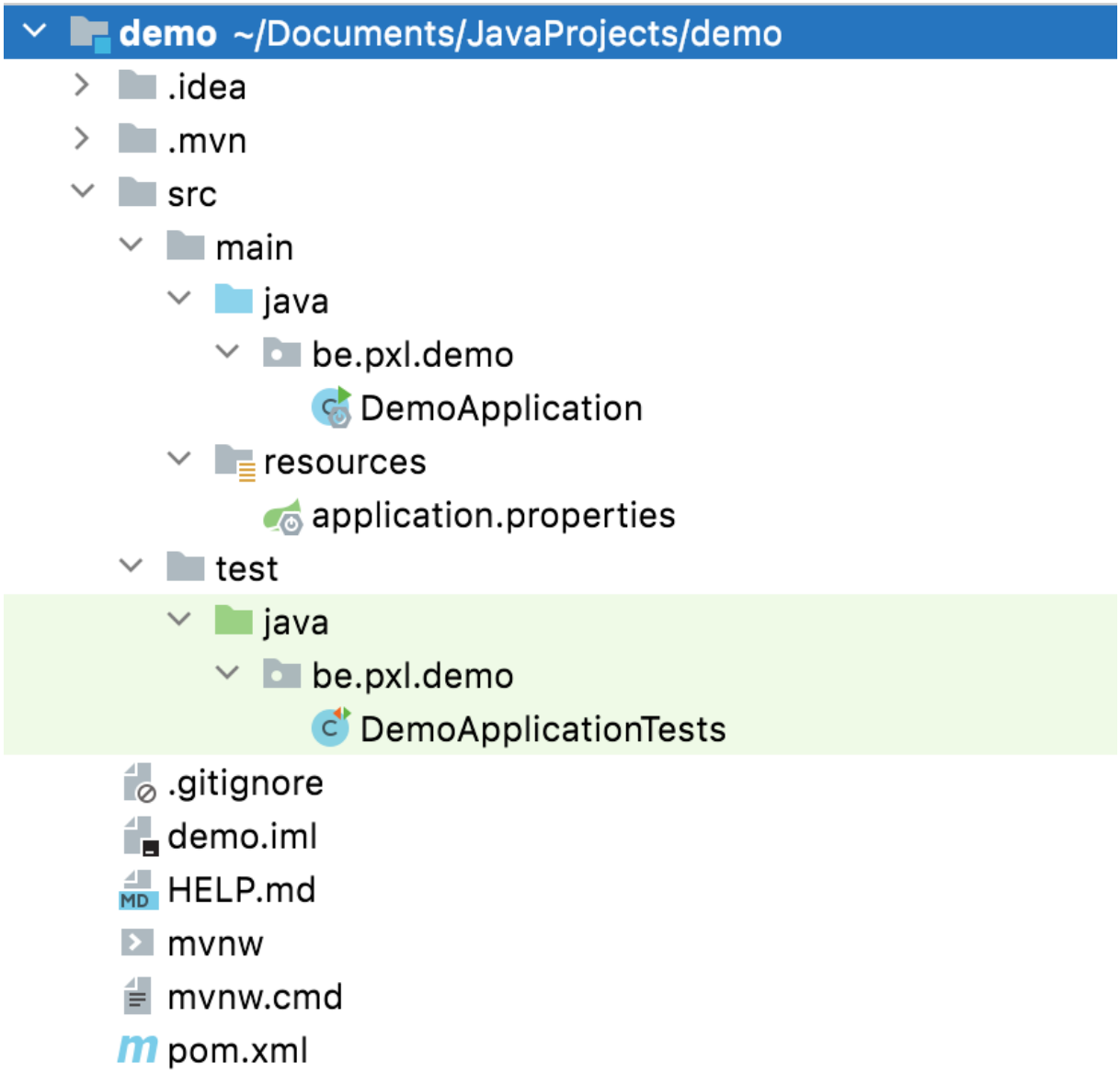
1.2.1 Gebruik van Spring Initializr

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.

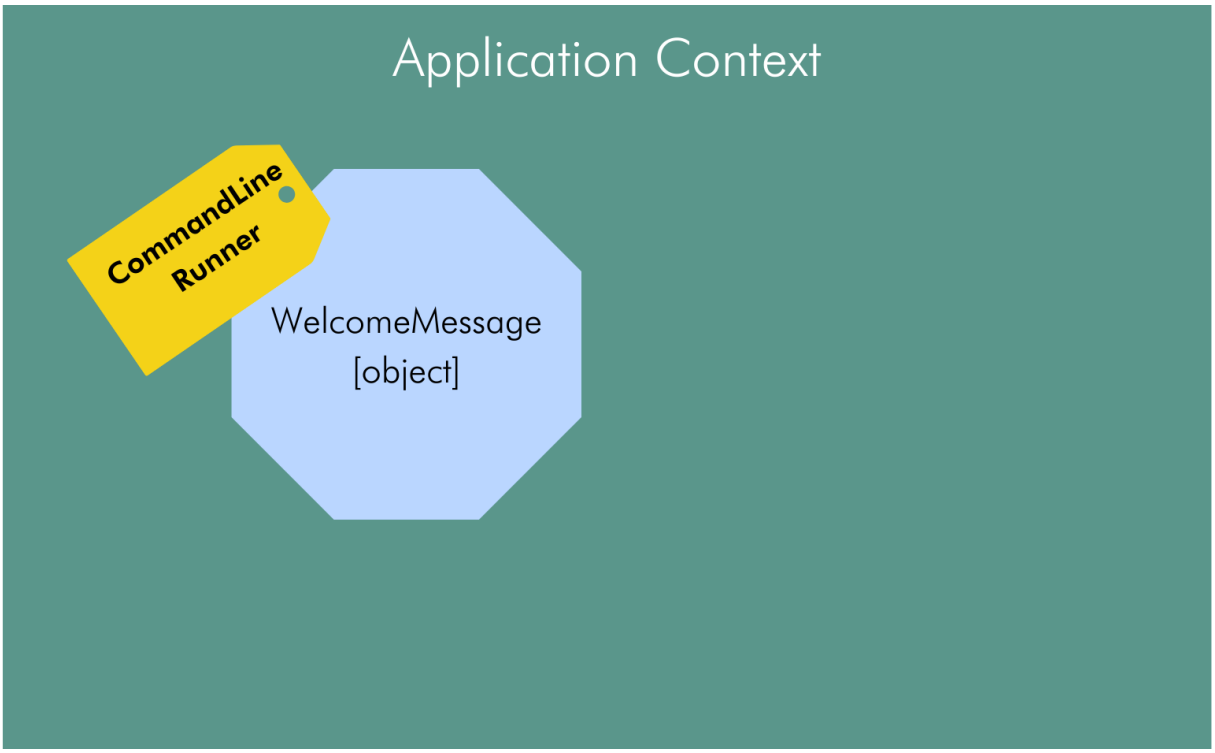


Figure 1.1: De Application Context

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);
   }
}
```

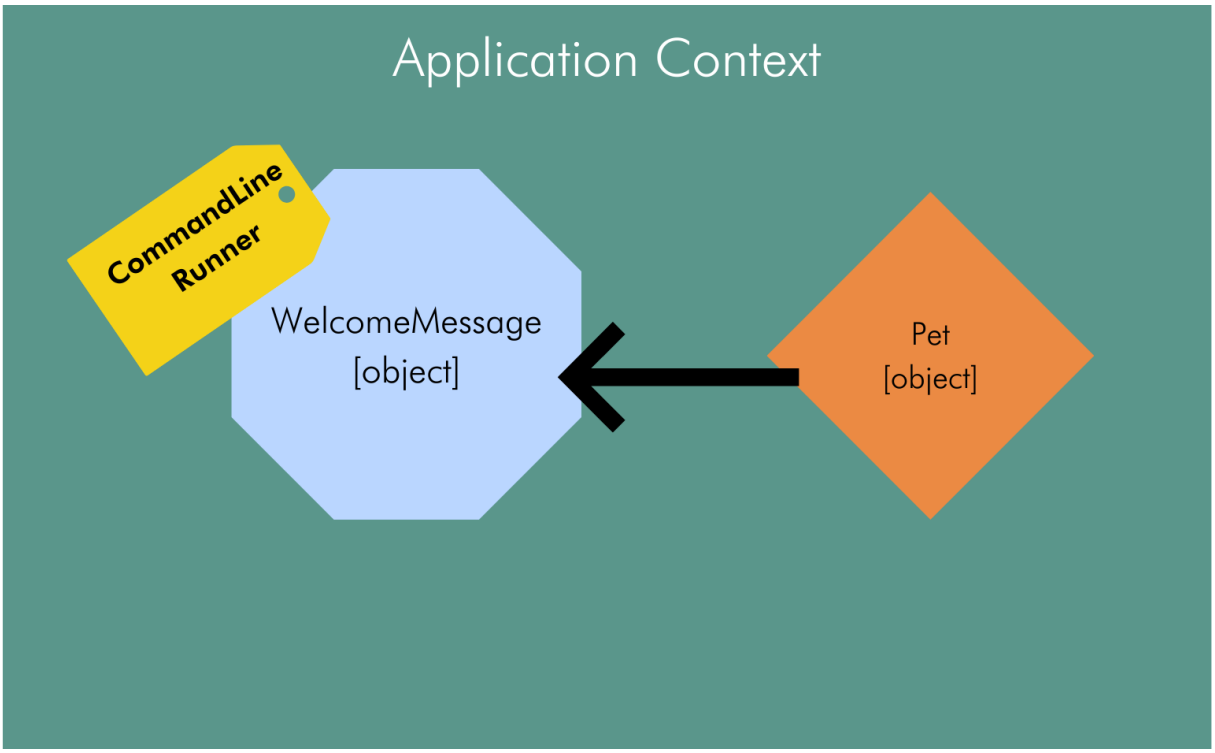


Figure 1.2: De Application Context

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.

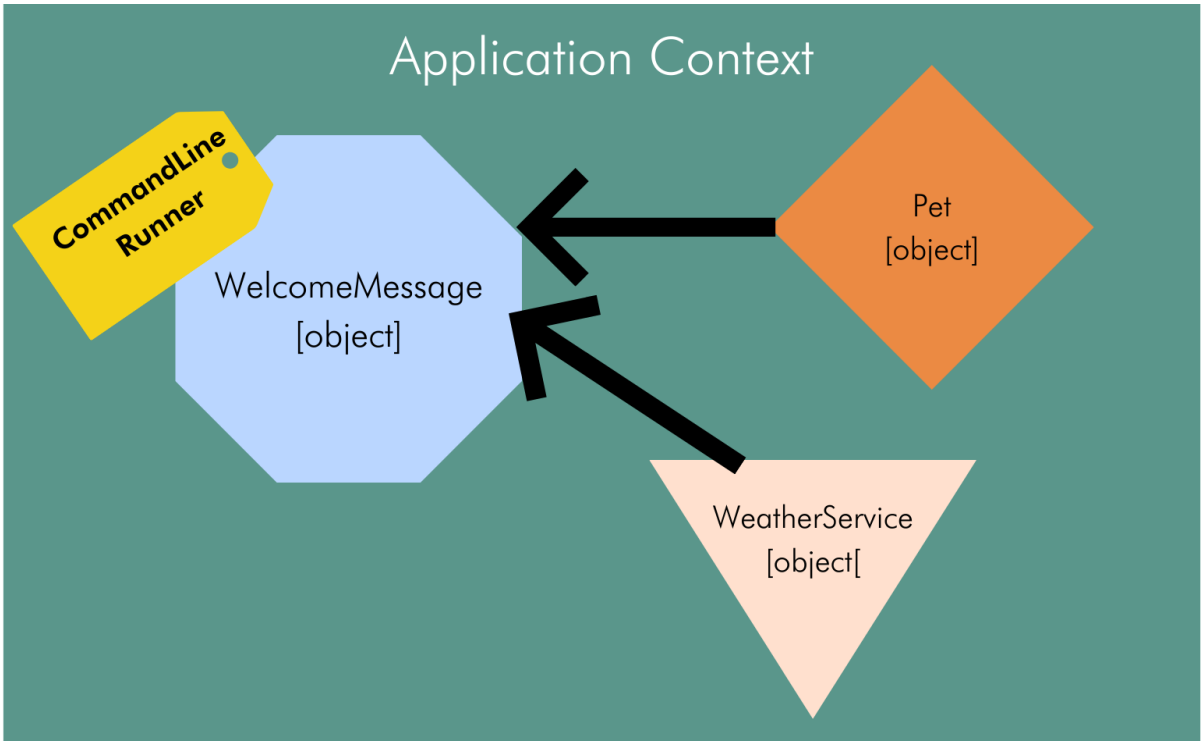


Figure 1.3: De Application Context

```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.

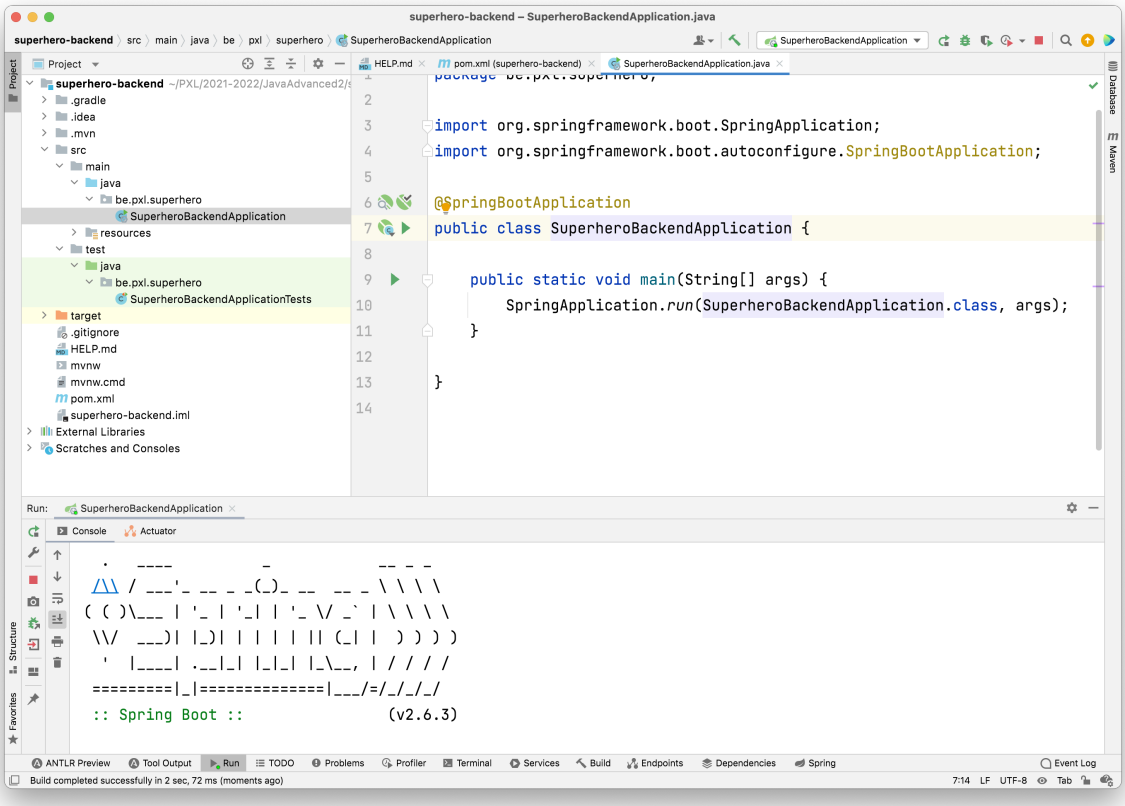
2.2 Spring Boot Starter Web

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.

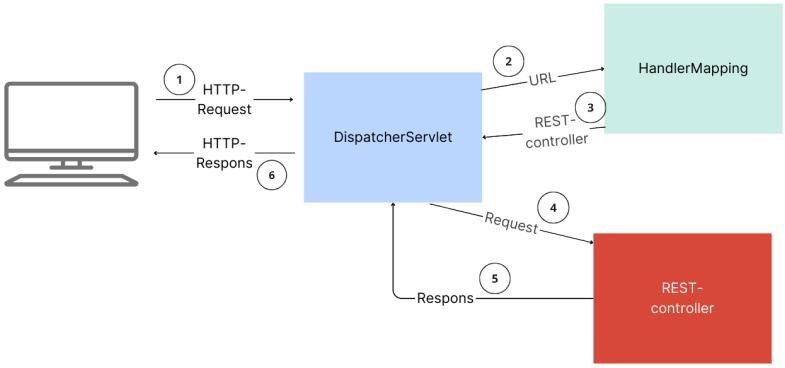


Figure 2.1: Een REST-verzoek afhandelen

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.

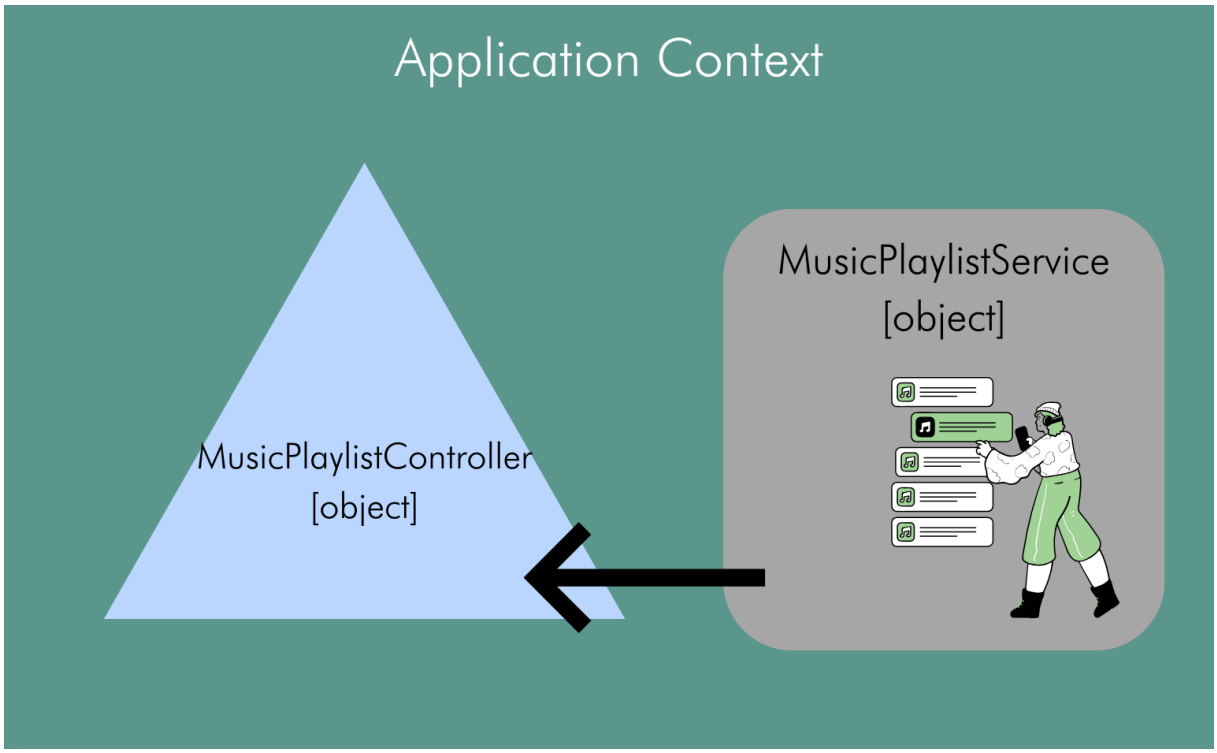


Figure 2.2: Spring Beans in de Application Context

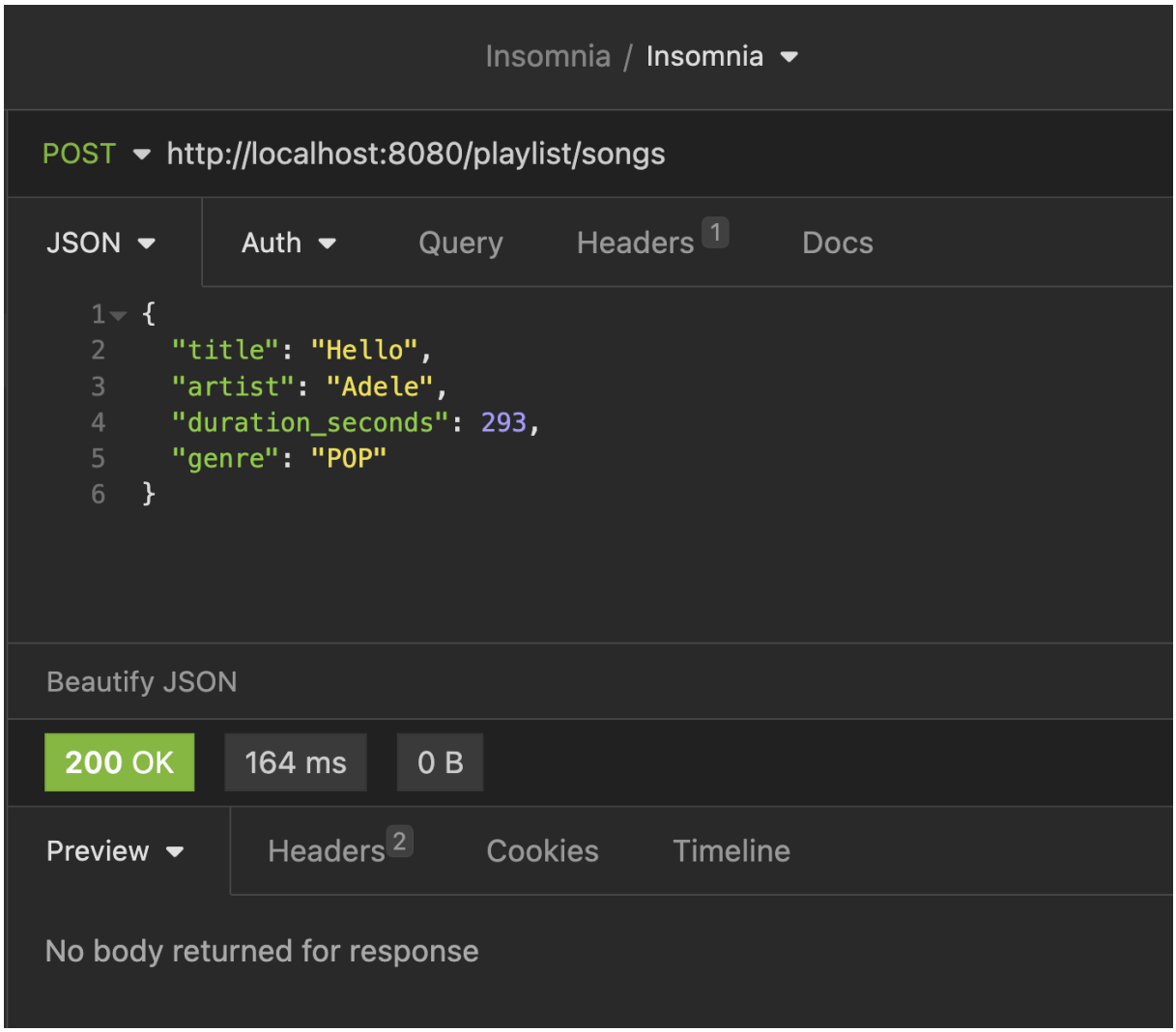


Figure 2.3: POST-verzoek met Insomnia

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

Chapter 3

Foutafhandeling

Tijdens het uitvoeren van een programma kan en zal er vanalles foutgaan. De gebruiker van het programma geeft de datum in in een foutief formaat, een printer is offline, er is onvoldoende schrijfruimte vrij om een bestand weg te schrijven, het programma kreeg onvoldoende geheugen toegewezen,... Fouten die zich voordoen tijdens het uitvoeren van een programma, at runtime dus, verdelen we onder in errors en exceptions. Errors zijn de fouten die meestal veroorzaakt worden door het onderliggende besturingssysteem. Tijdens het verloop van het programma kunnen deze errors niet meer opgelost worden en het programma zal daarom beëindigd worden. Dit is niet het geval voor exceptions. Je kan je code op een defensieve manier schrijven en rekening houden met de mogelijke exceptions die kunnen optreden. "Exception handling" is het proces om deze exceptions op een correcte en liefst gebruiksvriendelijke manier af te handelen. Indien een fout niet correct wordt afgehandeld, zal het programma alsnog voortijdig afgebroken worden, maar met de juiste foutafhandeling kan de uitvoer van het programma gewoon verdergezet worden.

3.1 Compile-time vs runtime errors

Een programmeur schrijft Java code in zijn editor of favoriete IDE. Vervolgens wordt de Java code gecompileerd tot bytecode. Deze bytecode wordt door de JVM (Java Virtual Machine) geïnterpreteerd tot machinecode instructies die worden uitgevoerd door het computersysteem.

Wanneer dus een Java programma wordt opgestart, kunnen er 2 categorieën van problemen voorkomen. Het kan zijn dat het Java programma niet gecompileerd kan worden. In dat geval spreken we van een compile-time error. Indien het programma succesvol gecompileerd wordt, kan er zich tijdens het uitvoeren van de code een probleem voordoen, in dat geval spreken we van runtime errors of exceptions.

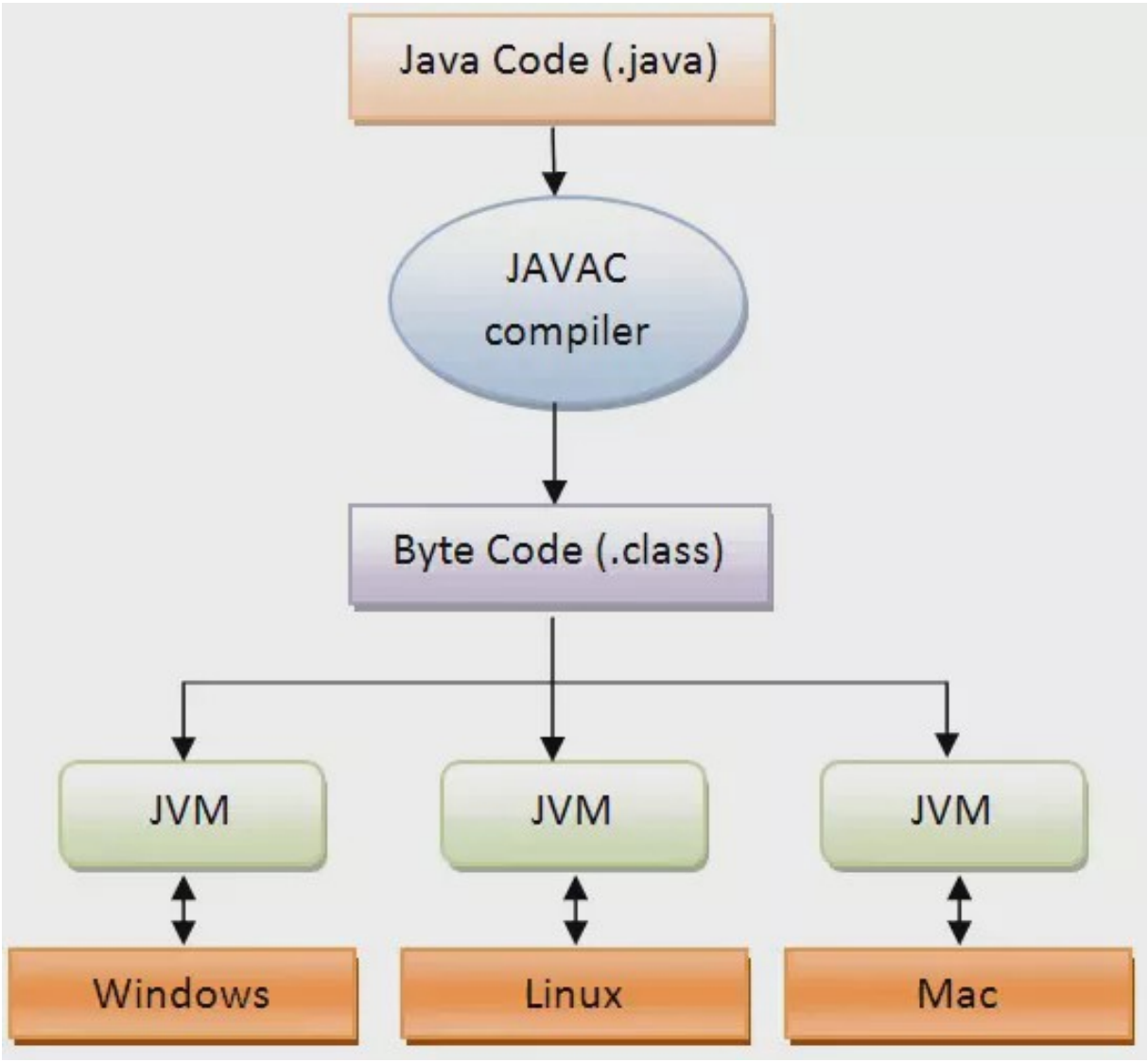


Figure 3.1: Compiler, interpreter en Java Virtual Machine.

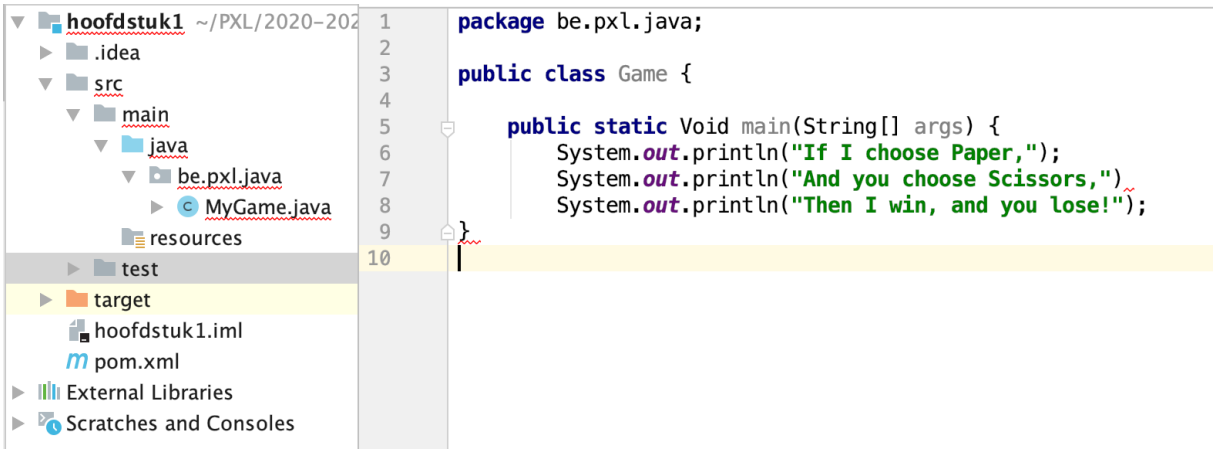


Figure 3.2: Compile-time errors

Oefening 3.1. Welke compile-time errors kan je ontdekken in het volgende code-fragment in figuur 3.2?

Omdat Java een object-geörienteerde programmeertaal is, wordt er ook een object gebruikt om aan te geven dat er iets fout ging bij uitvoeren van het Java programma. Zodra er zich een probleem voordoet tijdens het uitvoeren van een programma-instructie, wordt er een exception-object aangemaakt en "opgeworpen". Hierdoor stopt de normale uitvoer van het programma. Er wordt nog geprobeerd om het exception-object op een keurige manier af te handelen (indien die code aanwezig is), maar als dat niet lukt zal het programma beëindigd worden. Het exception-object bevat nuttige informatie voor de ontwikkelaar zoals de methode en lijn-nummer waar de exception werd aangemaakt en het type van de exception.

3.2 First catch

```
public class DivisionByZero {

    public static void main(String[] args) {
        int a = (1 + 1) % 2;
        int b = 5;
        int c = b / a;
        System.out.println("Het resultaat is " + c);
    }
}
```

Als je het bovenstaande programma uitvoert zal het volgende in de console verschijnen:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
at be.pxl.ja.DivisionByZero.main(DivisionByZero.java:6)<5 internal calls>
```

De variabele a bevat inderdaad de waarde 0 en hierdoor hebben we dus te maken met een deling door 0. Zodra de deling wordt uitgevoerd loopt het dus fout en gooit de java runtime een ArithmeticException op. Omdat de ArithmeticException nergens wordt afgehandeld eindigt het programma. Je zal enkel nog een stacktrace zien verschijnen in de console. Een stacktrace is de naam en foutboodschap van de exception gevolgd door de weg die de exception heeft afgelegd (doorheen de methoden van je klassen) vanaf het moment dat ze werd opgegooid. We zien dus dat de exception is veroorzaakt op regel 6 in de klasse DivisionByZero.

```
public class DivisionByZero {

    public static void main(String[] args) {
        int a = (1 + 1) % 2;
        int b = 5;
        try {
            int c = b / a;
```

```
            System.out.println("Het resultaat is " + c);
        } catch (ArithmeticException e) {
            System.out.println("You should not divide a number by zero.");
        }
        System.out.println("First catch completed!");
    }
}
```

```
You should not divide a number by zero.
First catch completed!
```

We hebben de instructie met de deling nu in een try-blok geplaatst. Omdat de variabele c in het try-blok wordt aangemaakt, kan deze variabele ook enkel binnen het try-blok gebruikt worden. Indien een exception optreedt binnen een try-blok zal de programma-uitvoer de resterende code binnen het try-blok overslaan en verdergaan bij het eerste catch-blok dat direct volgt achter het try-blok. Je bent als programmeur verplicht om een try-blok steeds te laten volgen door één of meerdere catch-blokken. In het catch-blok wordt dan de code uitgevoerd om het probleem op te lossen of tenminste een duidelijke boodschap voor de gebruiker van het programma te voorzien. Als alle instructies uit het catch-blok zijn uitgevoerd zal het programma zijn normale uitvoer verderzetten.

3.3 Java exception hiërarchie

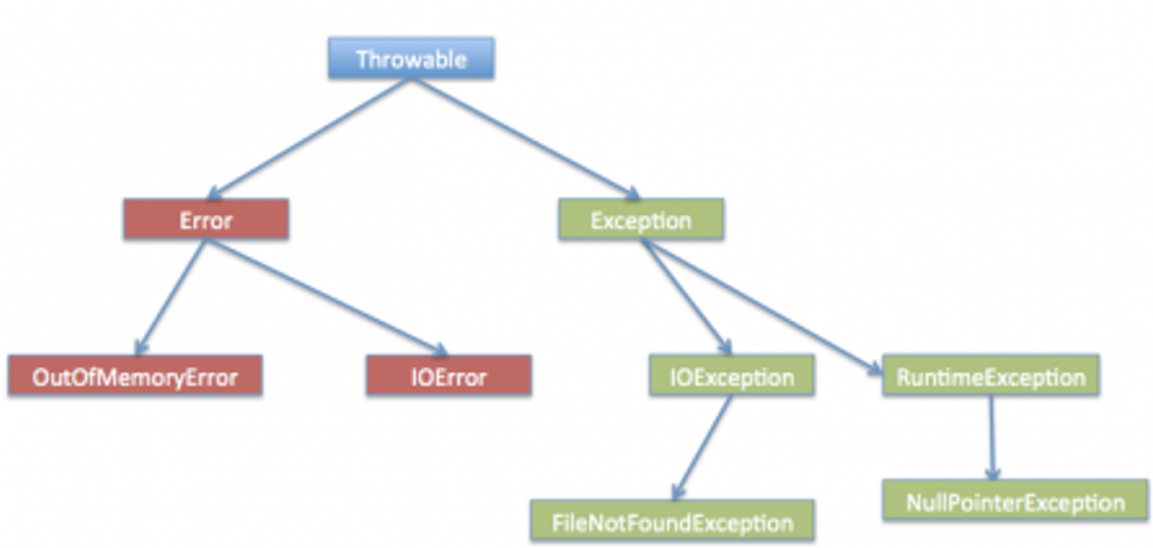


Figure 3.3: Exception hiërarchie

Zoals reeds vermeld wordt er een exception-object aangemaakt zodra zich een probleem voordoet in de code. Er is in Java een hiërarchie gebouwd van exception-klassen om verschillende soorten fouten in een programma te categorizeren. Throwable is de superklasse van alle exceptions en errors in Java. Er zijn dus 2 afgeleide klassen van Throwable: Error and Exception. Exceptions zijn nog verder onderverdeeld in **checked exceptions** en **runtime exceptions**.

3.3.1 Errors

Errors zijn problemen die zich voordoen tijdens het uitvoeren van het programma en die meestal niet gerelateerd zijn aan het programma zelf. Daarom is het onmogelijk om erop te anticiperen en te herstellen van deze fouten. Dit kan gaan van hardware falen, over JVM crashes en out of memory errors. We hebben een aparte hiërarchie van errors en we zullen nooit code toevoegen in ons programma om deze fouten af te handelen. We tonen hier enkel voorbeelden van errors.

StackOverflowError

Je hebt ongetwijfeld al eens per ongeluk een programma geschreven met een oneindige lus.

```
public class DemoStackOverflow {

    private static void printNumber(int x) {
        System.out.println(x);
        printNumber(x + 2);
    }

    public static void main(String[] args) {
        printNumber(15);
    }
}
```

```
15
17
19
...
36597
36599
Exception in thread "main" java.lang.StackOverflowError
...
at be.pxl.ja.DemoStackOverflow.printNumber(DemoStackOverflow.java:5)
at be.pxl.ja.DemoStackOverflow.printNumber(DemoStackOverflow.java:5)
at be.pxl.ja.DemoStackOverflow.printNumber(DemoStackOverflow.java:5)
at be.pxl.ja.DemoStackOverflow.printNumber(DemoStackOverflow.java:5)
...
```

De call stack is de manier waarop tijdens de uitvoer van een programma o.a. wordt bijgehouden welke functies worden aangeroepen. Wanneer je programma-uitvoer in een oneindige lus terechtkomt, stapelen de gegevens in de call stack zich razendsnel op en loopt de call stack vol. Het programma zal uiteindelijk eindigen met een StackOverflow-Error.

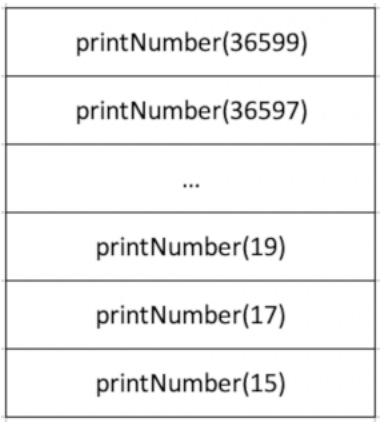


Figure 3.4: Java call stack

OutOfMemoryError

```
public class DemoOutOfMemory {

    private void generateOutOfMemory() {
        Long maxMemory = Runtime.getRuntime().maxMemory();
        System.out.println(maxMemory);

        int[] matrix = new int[(int) (maxMemory + 1)];
        for (int i = 0; i < matrix.length; ++i) {
            matrix[i] = i + 1;
        }
        System.out.println("Matrix filled" + matrix[(int)(Math.random() *
            ↪ 100)]);

    }

    public static void main(String[] args) {
        DemoOutOfMemory doom = new DemoOutOfMemory();
        doom.generateOutOfMemory();
    }
}
```

Om de OutOfMemoryError te illustreren maken we een array aan die meer geheugenplaatsen inneemt dan de ruimte die het java programma ter beschikking heeft.

Als je dit programma uitvoert, pas je best het beschikbare geheugen voor het programma aan. Dit doe je door een waarde voor de VM optie -Xmx mee te geven.

- -Xmssize: Geeft de initiële waarde voor de heap size.
- -Xmxsize: Geeft de maximale waarde voor heap size.

De heap size van een Java programma is de hoeveelheid geheugen dat een Java-programma mag gebruiken om objecten op te slaan.

```
67108864
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at be.pxl.ja.DemoOutOfMemory.generateOutOfMemory(DemoOutOfMemory.java:8)
at be.pxl.ja.DemoOutOfMemory.main(DemoOutOfMemory.java:16)<5 internal calls>
```

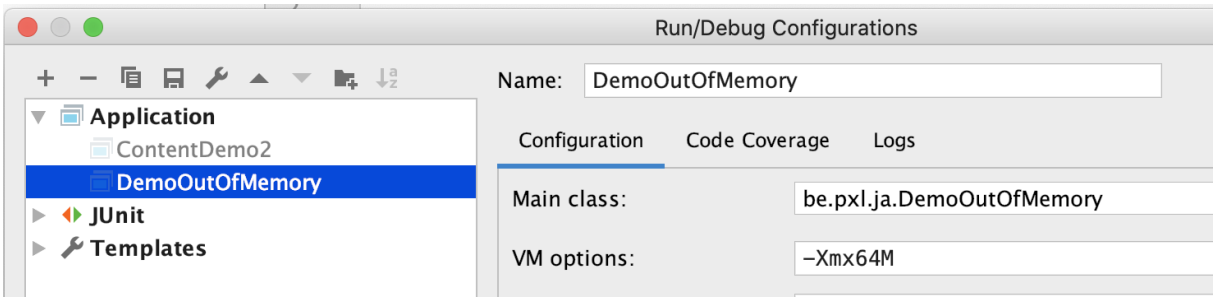


Figure 3.5: Maximum heap size aanpassen

3.3.2 Runtime exceptions

Runtime exceptions zijn exceptions die vaak veroorzaakt worden door logische fouten in het programma.Typische voorbeelden van runtime exceptions zijn ArrayIndexOutOfBoundsException, NullPointerException en IllegalArgumentException. Vaak moet je ze niet afhandelen, maar moet je ervoor zorgen dat je de bug in je code oplost. Ook hier zijn onze unit testen heel belangrijk. Door goede unit testen te schrijven ga je runtime exceptions in je code opmerken en kunnen oplossen.

```
public class Demo {

    public static void main(String[] args) {
        String tekst = "abc";
        System.out.println(tekst.repeat(-5));
    }
}
```

```
Exception in thread "main" java.lang.IllegalArgumentException: count is negative: -5
at java.base/java.lang.String.repeat(String.java:3586)
at be.pxl.ja.streamingservice.Demo.main(Demo.java:5)
```

Oefening 3.2.

```
public class StreamingService {

    private List<Account> accounts;

    public void addAccount(Account account) {
        accounts.add(account);
    }
}
```

Welke exception doet zich voor zodra je voor een object van de klasse StreamingService de methode addAccount() aanroept? Waarom treedt die exception op?

Wanneer je programma gebruikmaakt van gegevens die door de gebruiker worden ingevoerd, moet je er altijd rekening mee houden dat de gebruiker foutieve gegevens kan ingeven. Deze foutieve gegevens kunnen aanleiding geven tot runtime exceptions. Ook hier moet je steeds anticiperen op de mogelijke input die de gebruiker kan invoeren.

```
public class ElementInArray {

    public static void main(String[] args) {
        String[] elements = { "H", "He", "Li", "Be", "B", "C", "N", "O",
            ↪ "F", "Ne" };

        Scanner scanner = new Scanner(System.in);

        // OPLOSSING 1
        System.out.println("Kies een nummer: ");
        int chosen = scanner.nextInt();
        if (chosen < elements.length) {
            System.out.println(elements[chosen]);
        } else {
            System.out.println("U koos een verkeerd nummer.");
        }

        // OPLOSSING 2
        System.out.println("Kies een nummer: ");
        chosen = scanner.nextInt();
        try {
            System.out.println(elements[chosen]);
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("U koos een verkeerd nummer.");
        }
    }
}
```

In bovenstaand codevoorbeeld gaat onze voorkeur uit naar oplossing 1 waarbij de ingevoerde waarde wordt gecontroleerd vooraleer het element uit de array wordt benaderd. Deze code is makkelijker leesbaar en onderhoudbaar.

Oefening 3.3. Maak een programma dat de geboortedatum vraagt als input. Vervolgens berekent het programma hoeveel dagen het nog duurt vooraleer je jarig bent. Gebruik de methode parse() uit de klasse LocalDateTime om van de input van de gebruiker een LocalDateTime-object te maken. Welke exception kan optreden? Blijf input vragen totdat de gebruiker een correcte datum heeft ingegeven.

3.3.3 Checked exceptions

Wanneer je in java een methode aanroept, kan het voorkomen dat je direct een compileerfout voorgeschoteld krijgt. Het kan namelijk zijn dat java reeds anticipeert op mogelijke problemen en je dwingt om rekening te houden met het scenario dat er iets mis kan gaan.

De compileerfout raakt pas opgelost wanneer je het afhandelen van de exception netjes programmeert.

Kijk eens naar onderstaand voorbeeld uit de streaming service. We gebruiken hier de klasse MessageDigest uit JDK. Message digests zijn functies waarmee we voor input-data van willekeurige lengte een hash-waarde met een vaste lengte kunnen berekenen. Als je de hash-waarde kent, kan je hieruit de input-data niet afleiden. We gebruiken dit om een paswoord bij te houden in een Account-object. We willen absoluut vermijden dat paswoorden in een leesbaar formaat in onze objecten worden bijgehouden.

Om een message digest te berekenen in Java moet je eerst de static methode getInstance() aanroepen met als parameter het door jouw gekozen algoritme. In dit voorbeeld wordt er gekozen voor MD5. Je ziet dat de lijn code, ondanks het feit dat alles correct is geschreven, toch rood wordt onderlijnd. Dat komt omdat de methode getInstance() een exception kan opgooien die we verplicht moeten afhandelen.

Oefening 3.4. Open de documentatie van de klasse MessageDigest en bekijk de uitleg voor de static methode getInstance(). Kan je hier zien welke exceptions er kunnen voorkomen?

```
public class PasswordUtil {

    private static final String SPECIAL_CHARACTERS = "~!@#$%^&*()_-";
    private static final String ALGORITHM = "MD5";

    public static String encodePassword(String password) {
        MessageDigest messageDigest = MessageDigest.getInstance(ALGORITHM);
        messageDigest.update(pass
        return new BigInteger( sig
    }
```

Unhandled exception: java.security.NoSuchAlgorithmException

Figure 3.6: Een checked exception: NoSuchAlgorithmException

Deze compileerfout wordt veroorzaakt omdat de exception-klasse NoSuchAlgorithmException een checked exception is. Dit betekent dat het aanroepen van de methode die de exception gooit een compileerfout zal geven, omdat er geen code is toegevoegd om correct met de exception om te gaan. Er zijn 2 mogelijke oplossingen om deze compileerfout aan te pakken. Ofwel vang de exception op en handel je ze af in de methode encodePassword(), ofwel voeg je in de signatuur van de methode encodePassword() toe dat je de methode toelaat om een NoSuchAlgorithmException op te werpen. We geven nu eerst een voorbeeld van hoe je de exception kan opvangen en afhandelen.

```
import be.pxl.ja.streamingservice.util.PasswordUtil;

public class Account {
    private String email;
    private String password;

    public Account(String email, String password) {
        this.email = email;
```

```
        setPassword(password);
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public boolean verifyPassword(String password) {
        return PasswordUtil.isValid(password, this.password);
    }

    public void setPassword(String password) {
        this.password = PasswordUtil.encodePassword(password);
    }
}
```

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtil {

    private static final String ALGORITHM = "MD5";

    public static String encodePassword(String password) {
        MessageDigest messageDigest = null;
        try {
            messageDigest = MessageDigest.getInstance(ALGORITHM);
        } catch (NoSuchAlgorithmException e) {
            return null;
        }
        messageDigest.update(password.getBytes(), 0, password.length());
        return new BigInteger(1, messageDigest.digest()).toString(16);
    }

    public static boolean isValid(String providedPassword, String
        ↪ securedPassword) {
        return encodePassword(providedPassword).equals(securedPassword);
    }
}
```

```
import be.pxl.ja.streamingservice.model.Account;
```

```java
public class CheckedExceptionDemo {

    public static void main(String[] args) {
        Account newAccount = new Account("daffy@duckstad.be", "daffy123!");
        System.out.println(newAccount.verifyPassword("daffy123"));
        System.out.println(newAccount.verifyPassword("daffy123!"));
    }
}
```

Het algoritme "MD5" is een geldige waarde voor de parameter algorithm en je krijgt een probleemloos verloop van je programma. Maar een programmeur die zich vergist en de constante ALGORITHM in de klasse PasswordUtil de waarde "MD4" geeft zal een probleem veroorzaken.

```
Exception in thread "main" java.lang.NullPointerException
at be.pxl.ja.streamingservice.util.PasswordUtil.isValid(PasswordUtil.java:24)
at be.pxl.ja.streamingservice.model.Account.verifyPassword(Account.java:37)
at be.pxl.ja.streamingservice.CheckedExceptionDemo.main(CheckedExceptionDemo.java:9)
```

```
Exception in thread "main" java.lang.NullPointerException
at be.pxl.ja.streamingservice.util.PasswordUtil.isValid(PasswordUtil.java:25)
at be.pxl.ja.streamingservice.model.Account.verifyPassword(Account.java:40)
at be.pxl.ja.CheckedExceptionDemo.main(CheckedExceptionDemo.java:11)
```

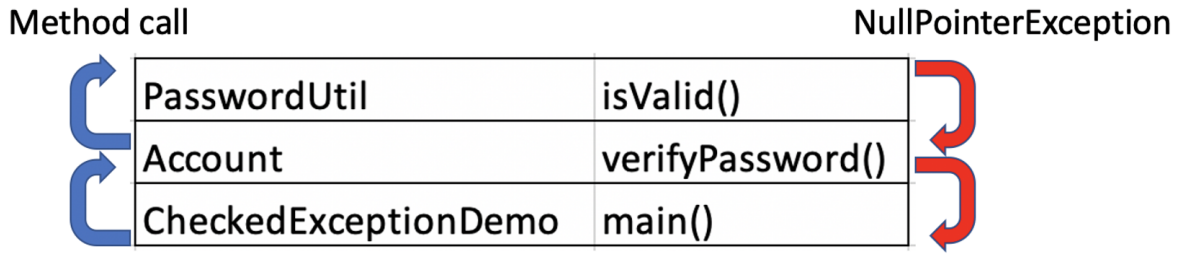


Figure 3.7: Method call stack

De NoSuchAlgorithmException wordt afgehandeld door null te geven als returnwaarde van de methode encodePassword. Hierdoor wordt het probleem pas opgemerkt op het ogenblik dat we de methode isValid van de klasse PasswordUtil gebruiken. De informatie dat het foute algoritme werd meegegeven is momenteel volledig verloren gegaan. Daarom wordt altijd aangeraden om exceptions die zich voordoen in je programma bij te houden in logbestanden. Het gebruik van logbestanden valt buiten de scope van deze cursus. Als alternatief voor het loggen van exceptions zullen we in deze cursus de stacktrace van de exception tonen in de console.

```java
import java.math.BigInteger;
```

```java
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtil {

    private static final String ALGORITHM = "MD4";

    public static String encodePassword(String password) {
        MessageDigest messageDigest = null;
        try {
            messageDigest = MessageDigest.getInstance(ALGORITHM);
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
        messageDigest.update(password.getBytes(), 0, password.length());
        return new BigInteger(1, messageDigest.digest()).toString(16);
    }

    public static boolean isValid(String providedPassword, String
        ↪ securedPassword) {
        return encodePassword(providedPassword).equals(securedPassword);
    }
}
```

Door de stacktrace te tonen in de console raken we geen cruciale informatie kwijt.

```
java.security.NoSuchAlgorithmException: MD4 MessageDigest not available
at java.base/sun.security.jca.GetInstance.getInstance(GetInstance.java:159)
at java.base/java.security.Security.getImpl(Security.java:700)
at java.base/java.security.MessageDigest.getInstance(MessageDigest.java:177)
at be.pxl.ja.streamingservice.util.PasswordUtil.encodePassword(PasswordUtil.java:15)
at be.pxl.ja.streamingservice.model.Account.setPassword(Account.java:45)
at be.pxl.ja.streamingservice.model.Account.<init>(Account.java:16)
at be.pxl.ja.streamingservice.CheckedExceptionDemo.main(CheckedExceptionDemo.java:8)
java.security.NoSuchAlgorithmException: md4 MessageDigest not available
at java.base/sun.security.jca.GetInstance.getInstance(GetInstance.java:159)
at java.base/java.security.Security.getImpl(Security.java:700)
at java.base/java.security.MessageDigest.getInstance(MessageDigest.java:177)
at be.pxl.ja.streamingservice.util.PasswordUtil.encodePassword(PasswordUtil.java:15)
at be.pxl.ja.streamingservice.util.PasswordUtil.isValid(PasswordUtil.java:25)
at be.pxl.ja.streamingservice.model.Account.verifyPassword(Account.java:37)
at be.pxl.ja.streamingservice.CheckedExceptionDemo.main(CheckedExceptionDemo.java:9)
Exception in thread "main" java.lang.NullPointerException
at be.pxl.ja.streamingservice.util.PasswordUtil.isValid(PasswordUtil.java:25)
at be.pxl.ja.streamingservice.model.Account.verifyPassword(Account.java:37)
at be.pxl.ja.streamingservice.CheckedExceptionDemo.main(CheckedExceptionDemo.java:9)
```

Opnieuw zie je het belang van unit testen. Een foute waarde voor het gekozen algoritme

ga je al heel snel opmerken en herstellen als je unit testen schrijft voor de methoden encodePassword() en isValid().

In dit voorbeeld is het eigenlijk aangewezen om de exception niet af te handelen. Een alternatief is dat we de methode encodePassword() toelaten om de exception, als die zich voordoet, gewoon verder door te geven (gooien). Hierdoor komt de exception dus terecht op de plaatsen waar je de methode encodePassword() gaat aanroepen en moet je op die plaatsen afhandeling voorzien. Je kan er dus voor kiezen om de checked exception NoSuchAlgorithmException helemaal mee te sleuren doorheen je applicatie tot aan de main()-methode.

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtil {

    private static final String ALGORITHM = "MD5";

    public static String encodePassword(String password) throws
        ↪ NoSuchAlgorithmException {
        MessageDigest messageDigest = MessageDigest.getInstance(ALGORITHM);
        messageDigest.update(password.getBytes(), 0, password.length());
        return new BigInteger(1, messageDigest.digest()).toString(16);
    }

    public static boolean isValid(String providedPassword, String
        ↪ securedPassword) throws NoSuchAlgorithmException {
        return encodePassword(providedPassword).equals(securedPassword);
    }
}
```

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

public class Account {
    private String email;
    private String password;

    public Account(String email, String password) throws
        ↪ NoSuchAlgorithmException {
        this.email = email;
        setPassword(password);
    }

    public String getEmail() {
        return email;
```

```
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public boolean verifyPassword(String password) throws
        ↪ NoSuchAlgorithmException {
        return PasswordUtil.isValid(password, this.password);
    }

    public void setPassword(String password) throws
        ↪ NoSuchAlgorithmException {
        this.password = PasswordUtil.encodePassword(password);
    }
}
```

```
import be.pxl.ja.streamingservice.model.Account;
import java.security.NoSuchAlgorithmException;

public class CheckedExceptionDemo {

    public static void main(String[] args) throws NoSuchAlgorithmException {
        Account newAccount = new Account("daffy@duckstad.be", "daffy123!");
        System.out.println(newAccount.verifyPassword("daffy123"));
        System.out.println(newAccount.verifyPassword("daffy123!"));
    }
}
```

Je ziet dat nu bij de signatuur van verschillende methoden "throws NoSuchAlgorithmException" verschijnt. Regelmatig verschijnen er artikels met titels als "Checked exceptions: Java's biggest mistake" en "Checked Exceptions are Evil" om het gebruik van checked exceptions te ontmoedigen.

Een laatste en nette oplossing is om de NoSuchAlgorithmException the **wrappen** in een runtime exception bijv. een IllegalArgumentException.

```
import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class PasswordUtil {

    private static final String ALGORITHM = "MD4";

    public static String encodePassword(String password) {
        MessageDigest messageDigest = null;
        try {
```

```
            messageDigest = MessageDigest.getInstance(ALGORITHM);
        } catch (NoSuchAlgorithmException e) {
            throw new IllegalArgumentException(e);
        }
        messageDigest.update(password.getBytes(), 0, password.length());
        return new BigInteger(1, messageDigest.digest()).toString(16);
    }

    public static boolean isValid(String providedPassword, String
        ↪ securedPassword) {
        return encodePassword(providedPassword).equals(securedPassword);
    }
}
```

Bij het uitvoeren van de methode encodePassword() met een foutief algoritme krijg je dan een runtime exception.

```
Exception in thread "main" java.lang.IllegalArgumentException: java.security.NoSuchAlgorit
at be.pxl.ja.streamingservice.util.PasswordUtil.encodePassword(PasswordUtil.java:17)
at be.pxl.ja.streamingservice.model.Account.setPassword(Account.java:48)
at be.pxl.ja.streamingservice.model.Account.<init>(Account.java:18)
at be.pxl.ja.CheckedExceptionDemo.main(CheckedExceptionDemo.java:10)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorIm
at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAc
at java.base/java.lang.reflect.Method.invoke(Method.java:564)
at com.intellij.rt.execution.application.AppMainV2.main(AppMainV2.java:131)
Caused by: java.security.NoSuchAlgorithmException: MD4 MessageDigest not available
at java.base/sun.security.jca.GetInstance.getInstance(GetInstance.java:159)
at java.base/java.security.Security.getImpl(Security.java:700)
at java.base/java.security.MessageDigest.getInstance(MessageDigest.java:177)
at be.pxl.ja.streamingservice.util.PasswordUtil.encodePassword(PasswordUtil.java:15)
... 8 more
```

3.4 Multi-catch blok en finally

```
import java.util.Scanner;

public class MultiCatchBlockDemo {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Kies een positie: ");
        int positie = scanner.nextInt();
        System.out.println("Kies een deler: ");
        int deler = scanner.nextInt();
        try {
            int getallen[] = new int[10];
```

```
            getallen[positie] = 30 / deler;
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Je moet een positie kiezen tussen 0 en 9.");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Je koos positie " + positie);
        }
        System.out.println("Start je het programma nog een keer.");
    }
}
```

Een try-blok kan gevolgd worden door één of meerder catch-blokken. Wanneer een exception optreedt zal bij het eerste catch-blok gestart worden. Indien onze exception een instantie is van de opgevangen exception (instanceof) dan zal dat catch-blok uitgevoerd worden en worden de volgende catch-blokken niet meer bekeken. Indien de exception geen instantie is van de opgevangen exception dan wordt er verder gekeken naar de volgende catch-blokken tot een overeenkomstig catch-blok worden gevonden. Indien er geen catch-blok wordt gevonden zal de runtime-exception doorstromen naar de aanroepende methode.

De volgorde van de catch-blokken is van belang. ArrayIndexOutOfBoundsException is een subklasse van Exception. Als we eerst een catch-blok aanmaken voor de superklasse en pas daarna een catch-blok voor de subklasse zou het tweede catch-blok "onbereikbaar" zijn. Het eerste catch-blok gaat de exception reeds kunnen afhandelen. Code die onbereikbaar is (unreachable code) wordt opgemerkt door de compiler wat resulteert in een compileerfout van je code.

Oefening 3.5. Wissel beide catch-blokken eens van plaats in bovenstaande code.

Het finally-blok tenslotte is een codeblok dat altijd wordt uitgevoerd: of er nu een exception optreedt of niet. Zelfs als er een exception optreedt en die niet kan worden afgehandeld door een catch-blok, zal toch het finally-blok uitgevoerd worden.

Oefening 3.6. Test de werking van het finally-blok eens uit met bovenstaand programma MultiCatchBlockDemo. Verwijder het tweede catch-blok eens en veroorzaak een deling door 0. Wat gebeurt er?

Indien je dezelfde code hebt om verschillende exceptions af te handelen, mag je exceptions combineren in een catch-block. Zo zal het catch-blok in onderstaand voorbeeld zowel ArrayIndexOutOfBoundsExceptions als ArithmeticExceptions afhandelen.

```
public class MultipleCatches {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Kies een positie: ");
        int positie = scanner.nextInt();
```

```
        System.out.println("Kies een deler: ");
        int deler = scanner.nextInt();
        try {
            int getallen[] = new int[10];
            getallen[positie] = 30 / deler;
        } catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {
            System.out.println(e.getMessage());
        }
        System.out.println("Start je het programma nog een keer.");
    }
}
```

3.5 Zelf exceptions opgooien

We willen gaan controleren dat de gebruikers van onze streaming service enkel geldige kredietkaartnummers invullen. Daarom maken we een aparte klasse CreditCardNumber. We gaan ervoor zorgen dat objecten van de klasse CreditCardNumber nooit ongeldige gegevens bevatten. Wanneer je een object van de klasse CreditCardNumber probeert aan te maken met ongeldige gegevens zal er een IllegalArgumentException gegooid worden.

We laten 2 types van kredietkaarten toe: VISA en MASTERCARD. Kaartnummers van VISA-kaarten starten altijd met het nummer 5, kaartnummers van MASTERCARD-kaarten starten altijd met 4. Verder wordt ook gecontroleerd dat de kaartnummers bestaan uit 16 cijfers. Bestudeer de klasse CreditCardNumber.

```
public class CreditCardNumber {
    private static final int LENGTH = 16;
    private static final int CVC_LENGTH = 3;

    private CreditCardType creditCardType;
    private String number;
    private String cvc;

    public CreditCardNumber(String number, String cvc) {
        if (!isNumeric(number) || number.length() != LENGTH) {
            throw new IllegalArgumentException("A card number must have " +
                ↪ LENGTH + " digits.");
        }
        creditCardType = getCreditCardType(number);
        if (creditCardType == null) {
            throw new IllegalArgumentException("This is not a valid credit
                ↪ card.");
        }
    }

    private boolean isNumeric(String text) {
        if (text == null || text.length() == 0) {
```

```
            return false;
        }
        try {
            Long.parseLong(text);
            return true;
        } catch (NumberFormatException e) {
            return false;
        }
    }

    private CreditCardType getCreditCardType(String number) {
        for (CreditCardType cardType : CreditCardType.values()) {
            if (cardType.getFirstNumber() ==
                ↪ Integer.parseInt(number.substring(0, 1))) {
                return cardType;
            }
        }
        return null;
    }
}
```

Natuurlijk gaan we de constructor van onze nieuwe klasse CreditCardNumber ook grondig testen. Wanneer we dus foutieve waarden meegeven aan de constructor gaan we moeten verifiëren dat de IllegalArgumentException wordt opgegooid.

Hier is alvast een eenvoudig voorbeeld om te tonen hoe de methode assertThrows van de klasse Assertions in junit werkt.

```
@Test
void testExpectedException() {
  Assertions.assertThrows(NumberFormatException.class, () -> {
    Integer.parseInt("One");
  });
}
```

De assertThrows() verwacht dat een NumberFormatException zal worden opgegooid. Omdat de string "One" een ongeldige waarde is zal de NumberFormatException ook effectief gegooid worden en zal de test dus slagen.

Nu zie je een aantal testen om onze constructor van de klasse CreditCardNumber te testen. Bestudeer de testen grondig.

```
import org.junit.jupiter.api.Test;

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

public class CreditCardNumberTest {
```

```
    @Test
    public void validVisaCard() {
        CreditCardNumber creditCardNumber = new
            ↪ CreditCardNumber("4321876532147654", "123");

        assertEquals(CreditCardType.VISA, creditCardNumber.getType());
        assertEquals("123", creditCardNumber.getCvc());
        assertEquals("4321876532147654", creditCardNumber.getNumber());
    }

    @Test
    public void validVisaCardWithBlanks() {
        CreditCardNumber creditCardNumber = new CreditCardNumber(" 43218
            ↪ 76532 1476 54 ", " 1 2 3 ");

        assertEquals(CreditCardType.VISA, creditCardNumber.getType());
        assertEquals("123", creditCardNumber.getCvc());
        assertEquals("4321876532147654", creditCardNumber.getNumber());
    }

    @Test
    public void validMasterCard() {
        CreditCardNumber creditCardNumber = new
            ↪ CreditCardNumber("5321876532147654", "123");

        assertEquals(CreditCardType.MASTERCARD, creditCardNumber.getType());
        assertEquals("123", creditCardNumber.getCvc());
        assertEquals("5321876532147654", creditCardNumber.getNumber());
    }

    @Test
    public void validMasterCardWithBlanks() {
        CreditCardNumber creditCardNumber = new CreditCardNumber(" 53218
            ↪ 76532 1476 54 ", " 1 2 3 ");

        assertEquals(CreditCardType.MASTERCARD, creditCardNumber.getType());
        assertEquals("123", creditCardNumber.getCvc());
        assertEquals("5321876532147654", creditCardNumber.getNumber());
    }

    @Test
    public void throwsInvalidArgumentExceptionWhenNumberTooShort() {
        assertThrows(IllegalArgumentException.class, () -> {
            new CreditCardNumber(" 53218 76532 1476 ", " 1 2 3 ");
        });
    }

    @Test
```

```
    public void throwsInvalidArgumentExceptionWhenNumberTooLong() {
        assertThrows(IllegalArgumentException.class, () -> {
            new CreditCardNumber(" 53218 76532 1476 4445 ", " 1 2 3 ");
        });
    }

    @Test
    public void throwsInvalidArgumentExceptionWhenInvalidCardType() {
        assertThrows(IllegalArgumentException.class, () -> {
            new CreditCardNumber("7321876532147654", "123");
        });
    }
}
```

Oefening 3.7. Voeg in de constructor van de klasse CreditCardNumber een extra validatie toe voor de CVC (card validation code). De CVC is een getal bestaande uit 3 cijfers. Pas je unit testen aan. Waarschijnlijk moet je extra unit testen toevoegen om de constructor van de klasse CreditCardNumber te testen.

3.6 Zelf exception-klassen schrijven

In de klasse CreditCardNumber hebben we gebruikgemaakt van een bestaande exception uit de JDK. We kunnen ook onze eigen Exception-klassen voorzien.

De klasse PaymentInfo gaan we nu grondig aanpassen door o.a. gebruik te maken van de klasse CreditCardNumber. Daarnaast willen we ook de vervaldatum van de kredietkaart gaan controleren. We willen namelijk dat de kredietkaart nog minstens 1 maand geldig is op het ogenblik dat de betaalgegevens worden ingegeven. Indien de vervaldatum binnen de maand valt, gaan we een InvalidDateException opgooien.

Deze InvalidDateException-klasse bestaat niet in de JDK, dus gaan we hem zelf voorzien. Wanneer je een **checked exception** wil maken gebruik je de klasse Exception als superklasse van je nieuwe exception. Wanneer je een **unchecked exception** wil maken gebruik je de klasse RuntimeException als superklasse.

Hier is alvast onze nieuwe exception klasse InvalidDateException. Als je zelf een exception klasse aanmaakt kan je zelf beslissen welke parameters en extra eigenschappen je voorziet in de constructor. Regelmatig wordt ook de afspraak gehanteerd dat er pas een nieuwe exception klasse wordt toegevoegd, indien één van de reeds bestaande exception klassen niet eenvoudig hergebruikt kan worden. In ons geval, gaan we de ongeldige datum willen tonen. Om het samenstellen van de foutboodschap te kunnen hergebruiken is het dus zinvol om een InvalidDateException te maken met o.a. de ongeldige datum (LocalDate) als parameter.

```
public class InvalidDateException extends RuntimeException {

    public static final DateTimeFormatter FORMATTER =
        ↪ DateTimeFormatter.ofPattern("dd/MM/yyyy");
```

```
    public InvalidDateException(LocalDate incorrectDate, String type,
        ↪ String description) {
        super(FORMATTER.format(incorrectDate) + " is not a valid " + type +
            ↪ ". " + description);
    }
}
```

```
import java.time.LocalDate;

public class PaymentInfo {

    private String firstName;
    private String lastName;
    private CreditCardNumber cardNumer;
    private LocalDate expirationDate;

    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 void setCardNumer(CreditCardNumber cardNumer) {
        this.cardNumer = cardNumer;
    }

    public LocalDate getExpirationDate() {
        return expirationDate;
    }

    public void setExpirationDate(LocalDate expirationDate) {
        if (LocalDate.now().plusMonths(1).isAfter(expirationDate)) {
            throw new InvalidDateException(expirationDate, "expirationDate",
                ↪ "Must be valid for at least 1 month.");
        }
        this.expirationDate = expirationDate;
    }
```

```
}
```

We gaan deze methode setExpirationDate nu ook grondig testen.

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.time.LocalDate;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class PaymentInfoSetExpirationDateTest {

    private PaymentInfo paymentInfo;

    @BeforeEach
    public void init() {
        paymentInfo = new PaymentInfo();
    }

    @Test
    public void throwsInvalidDateExceptionWhenExpirationDayWithinOneMonth()
        ↪ {
        LocalDate withinOneMonth =
            ↪ LocalDate.now().plusMonths(1).minusDays(1);
        assertThrows(InvalidDateException.class, () ->
            ↪ paymentInfo.setExpirationDate(withinOneMonth));
    }

    @Test
    public void expirationDayWithinExactlyOneMonthIsAllowed() {
        LocalDate exactlyOneMonth = LocalDate.now().plusMonths(1);
        paymentInfo.setExpirationDate(exactlyOneMonth);

        assertNotNull(paymentInfo.getExpirationDate());
        assertEquals(exactlyOneMonth, paymentInfo.getExpirationDate());
    }

    @Test
    public void expirationDayOverOneMonthIsAllowed() {
        LocalDate overOneMonth = LocalDate.now().plusMonths(1).plusDays(1);
        paymentInfo.setExpirationDate(overOneMonth);

        assertNotNull(paymentInfo.getExpirationDate());
        assertEquals(overOneMonth, paymentInfo.getExpirationDate());
    }
```

```
}
```

Oefening 3.8. Schrijf ook een kort programma waarin je de InvalidDateException veroorzaakt. Vang de exception op en toon de foutboodschap (message)?

Chapter 4

Spring Validation

Learning goals
De junior-collega
1. kan gebruikmaken van Spring Validation
2. kan de correcte validatieregels gebruiken

Wanneer Spring Boot een HTTP-verzoek ontvangt, is het belangrijk om de gegevens die met dat verzoek zijn meegezonden te valideren. Spring Boot heeft voorgedefinieerde validatoren waardoor het controleren van veelvoorkomende regels eenvoudig kan gebeuren. Zo kan Spring Boot bijvoorbeeld heel eenvoudig controleren of een e-mailadres het juiste formaat heeft of een getal binnen een bepaald bereik valt,. In Spring Boot is het uiteraard ook mogelijk om aangepaste of eigen validatieregels te programmeren.

Om Spring Validation te gebruiken voeg je een dependency toe aan het project.

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

Here is a list of the most common validation annotations.

@NotNull	om aan te duiden dat een referentie niet null mag zijn.
@NotEmpty	om aan te duiden dat een list elementen moet bevatten.
@NotBlank	om aan te duiden dat een string niet leeg mag zijn.
@Min and @Max	om aan te duiden dat een numerieke waarde enkel geldig is als de waarde groter of kleiner is dan een gegeven waarde.
@Size	om de lengte van een string te valideren.
@Email	om te controleren of een string een geldig e-mailadres bevat.

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

import com.fasterxml.jackson.annotation.JsonProperty;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
```

```
public class Song {
    @NotBlank
    private String title;
    @NotBlank
    private String artist;
    @JsonProperty("duration_seconds")
    @Min(value = 1, message = "Must be greater than zero")
    private int durationSeconds;
    @NotNull
    private Genre genre;

    ...
}
```

Wanneer Spring Boot een argument vindt dat is geannoteerd met @Valid, wordt het automatisch gevalideerd.

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

Wanneer het argument niet voldoet aan de regels dan gooit Spring Boot een exception van de klasse MethodArgumentNotValidException.

Sinds versie 2.3 moet je de eigenschap server.error.include-message de waarde always geven om de foutboodschappen te kunnen zien in het response. Spring Boot verbergt namelijk het foutmelding in de respons om het lekken van gevoelige informatie te voorkomen.

De foutmelding blijft ondanks in de instelling eerder cryptisch. Stel dat ik de REST API gebruik om een lied toe te voegen met de volgende gegevens:

```
{
  "title": "",
  "artist": "Bruno Mars",
  "duration_seconds": -2,
  "genre": "OTHER"
}
```

De krijg ik het volgende antwoord terug:

```
{
"timestamp": "2023-09-23T13:48:50.490+00:00",
"status": 400,
"error": "Bad Request",
"message": "Validation failed for object='song'. Error count: 2",
"path": "/playlist/songs"
}
```

Meer gegevens krijg ik te zien als ik de eigenschap server.error.include-binding-errors=always toevoeg in application.properties. Voor bovenstaande RequestBody krijg ik dan de volgende respons. Er wordt op dat ogenblik al veel interne informatie gelekt.

```
{
"timestamp": "2023-09-23T13:52:38.028+00:00",
"status": 400,
"error": "Bad Request",
"message": "Validation failed for object='song'. Error count: 2",
"errors": [
{
"codes": [
"Min.song.durationSeconds",
"Min.durationSeconds",
"Min.int",
"Min"
],
"arguments": [
{
"codes": [
"song.durationSeconds",
"durationSeconds"
],
"arguments": null,
"defaultMessage": "durationSeconds",
"code": "durationSeconds"
},
10
],
"defaultMessage": "Invalid duration_seconds: must equal or exceed 10.",
"objectName": "song",
```

```
"field": "durationSeconds",
"rejectedValue": -2,
"bindingFailure": false,
"code": "Min"
},
{
"codes": [
"NotBlank.song.title",
"NotBlank.title",
"NotBlank.java.lang.String",
"NotBlank"
],
"arguments": [
{
"codes": [
"song.title",
"title"
],
"arguments": null,
"defaultMessage": "title",
"code": "title"
}
],
"defaultMessage": "must not be blank",
"objectName": "song",
"field": "title",
"rejectedValue": "",
"bindingFailure": false,
"code": "NotBlank"
}
],
"path": "/playlist/songs"
}
```


De best practice om de MethodArgumentNotValidException af te handelen is door gebruik te maken van een exception handler. Hierbij gebruiken we de annotaties @RestControllerAdvice en @ExceptionHandler.

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

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import java.util.HashMap;
import java.util.List;
```

```
import java.util.Map;
import java.util.stream.Collectors;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<Map<String, List<String>>>
        ↪ handleValidationErrors(MethodArgumentNotValidException ex) {
        List<String> errors = ex.getBindingResult().getFieldErrors()
                .stream().map(FieldError::getDefaultMessage).collect(Collectors.toList());
        return new ResponseEntity<>(getErrorsMap(errors),
            ↪ HttpStatus.BAD_REQUEST);
    }

    private Map<String, List<String>> getErrorsMap(List<String> errors) {
        Map<String, List<String>> errorResponse = new HashMap<>();
        errorResponse.put("errors", errors);
        return errorResponse;
    }

}
```

We passen de boodschap bij onze validaties ook best aan. Gebruik een duidelijke foutboodschap om de gebruiker duidelijk te maken welke gegevens hij moet doorgeven. Als je een meertalige applicatie hebt, kan je best een key (zoals title.not.blank) doorgeven aan de frontend zodat de eigenlijk foutboodschap in de juiste taal getoond kan worden.

```
    @NotBlank(message = "title.not.blank")
    private String title;
    @Min(value = 10, message = "Invalid duration_seconds: must be equal to
        ↪ or exceed 10.")
    private int durationSeconds;
```

Er wordt in het geval van een MethodArgumentNotValidException een HTTP status 400 met de volgende response body gegeven:

```
{
"errors": [
"Invalid duration_seconds: must be equal to or exceed 10.",
"title.not.blank"
]
}
```