Programming Advanced - Java

Introductie tot Spring Boot

Nele Custers

May 17, 2020

Hogeschool PXL

Contents

Inleiding 1

1.1 Wat is Spring Boot?

Spring Boot is een Java platform om kwaliteitsvolle, stand-alone Spring applicaties te ontwikkelen. Met een minimum aan configuratie kan je reeds van start gaan.

De voordelen van Spring Boot voor ontwikkelaars zijn:

- een eenvoudig framework om Spring applicaties te ontwikkelen
- verhoogt de productiviteit van de ontwikkelaars
- vermindert de ontwikkeltijd van toepassingen

1.2 Spring Boot Starters

Een Spring Boot applicatie zal (voor een groot deel) automatisch geconfigureerd kunnen worden aan de hand van de dependencies van het project.

Voor grote projecten is het beheren van de dependencies niet altijd gemakkelijk. Spring Boot lost dit probleem op door bepaalde dependencies te groeperen. Als je bijvoorbeeld JPA gebruikt om toegang te krijgen tot een databank, is het voldoende om de spring-boot-starter-data-jpa dependency in je project toe te voegen. Met deze ene dependency worden alle database-gerelateerde libraries toegevoegd.

Een tweede Spring Boot starter die we gaan gebruiken is spring-boot-starter-web. Deze voegt aan ons project alle libraries toe die we nodig hebben om webcomponenten te ontwikkelen. Zo zal er o.a. een embedded server in het Spring Boot project worden voorzien. Dit heeft als voordeel dat er op de omgeving waar het Spring Boot project wordt uitgevoerd, geen voorgeïnstalleerde server aanwezig moet zijn. De default embedded server voor Spring Boot is Tomcat. Het Spring MVC framework dat alles voorziet

voor het ontwikkelen van RESTful web services maakt ook deel uit van spring-boot-starter-web.

Alle Spring Boot starters volgen hetzelfde patroon voor de naamgeving: spring-boot-starter-*, waarbij * aanduidt welke type functionaliteit en wordt aangeboden.

1.3 Componenten en dependency injection

In onze applicatie voorzien we 3 lagen: presentatie-laag, service- of business-laag en persistence-laag. De typische componenten in de business- en persistence-laag gaan we markeren zodat Spring deze componenten kan beheren. Op die manier zijn we zelf niet verantwoordelijk om die componenten aan te maken, maar kunnen we ze gebruiken in andere klassen door ze hier te injecteren met de annotatie @Autowired.

Enkele van de annotaties die we gaan gebruiken zijn:

- @Component generieke annotatie voor componenten die door Spring worden beheerd
- @Service markeren van klassen uit de service-laag
- @Repository markeren van klassen uit de persistence-laag

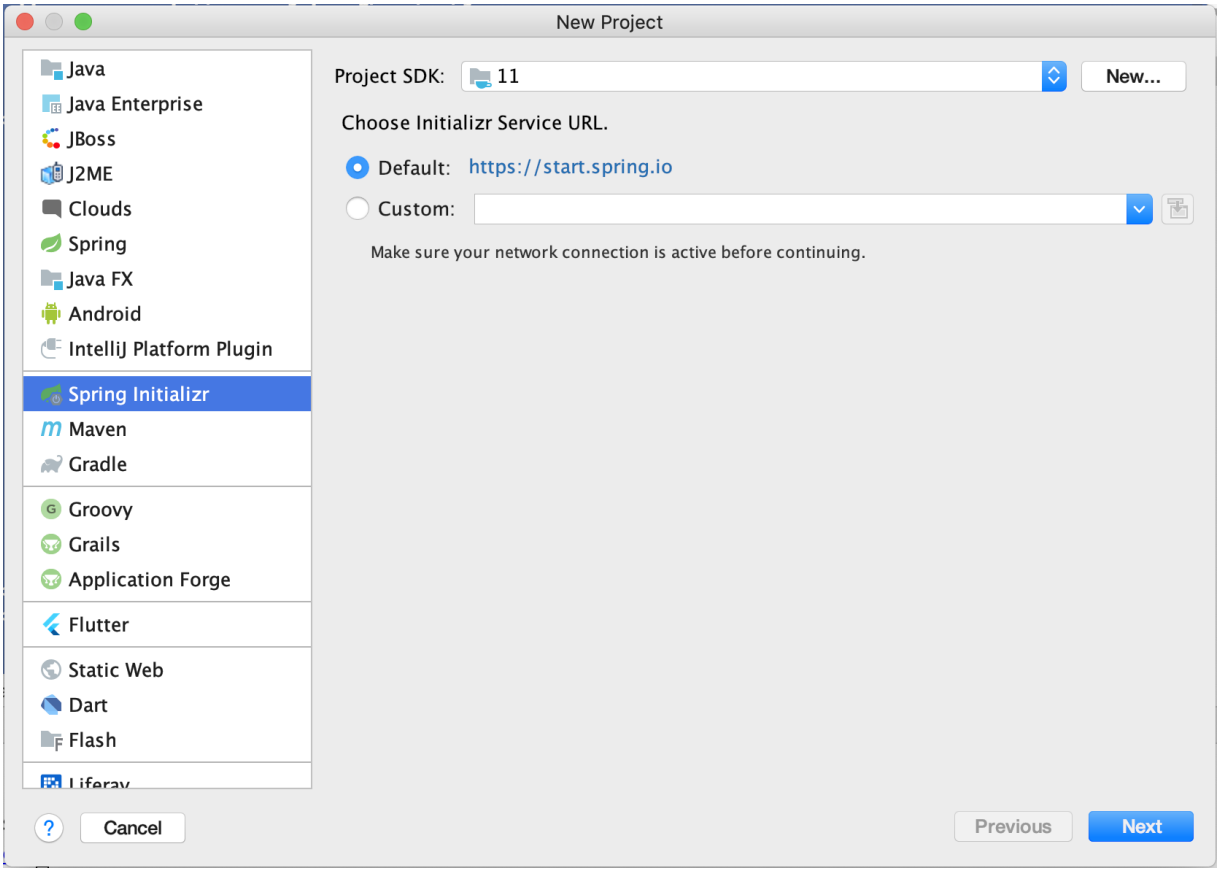
2 Demo project

2.1 Een project aanmaken

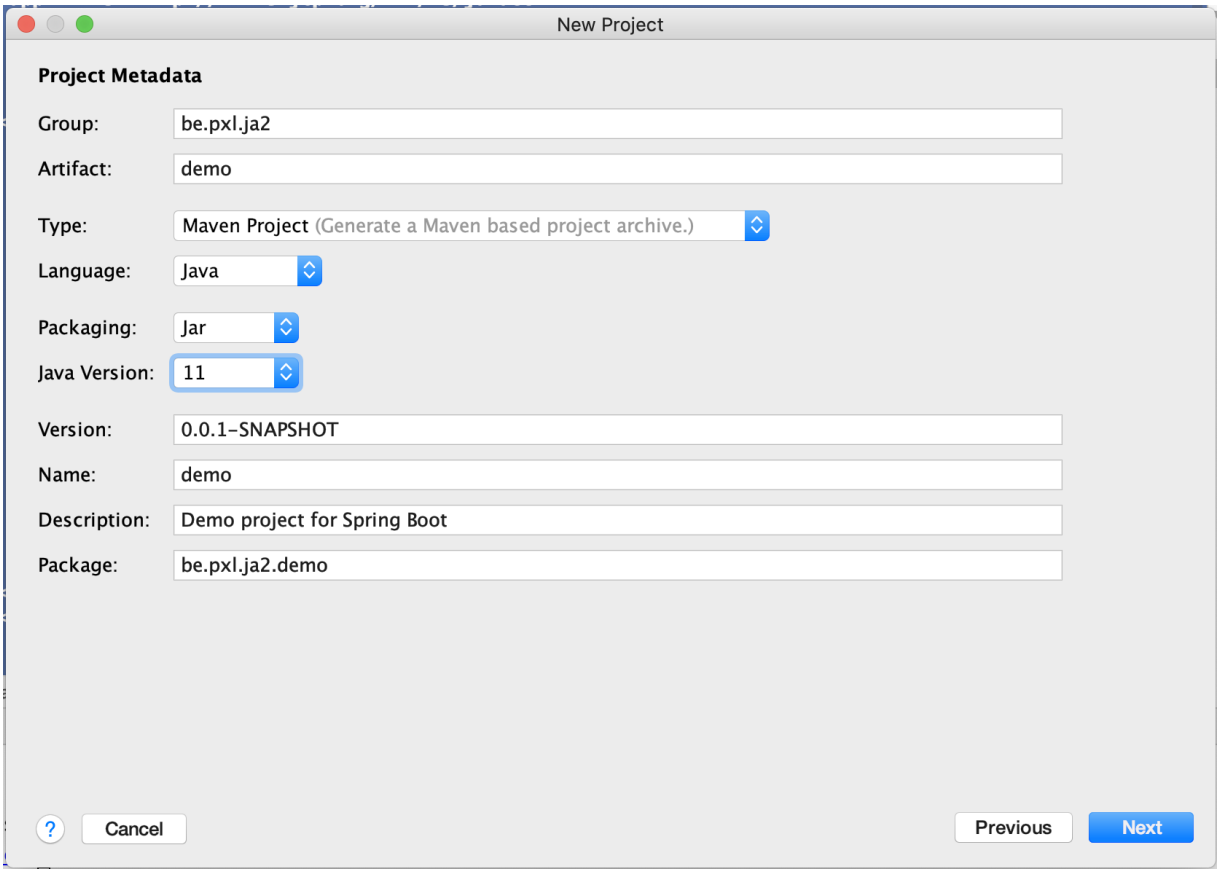
In dit hoofdstuk implementeren we een eerste Spring Boot applicatie.

We maken gebruik van de Spring Initializr om eenvoudig de startcode voor een nieuw Spring Boot project te creëren. Spring Initializr is een wizard die je doorheen de configruatie van een Spring Boot project loodst.

Wanneer je in IntelliJ IDEA een nieuwe project aanmaakt kan je hier Spring Initializr reeds selecteren.



In het eerste scherm dat verschijnt zal je de metadata voor het project invullen.



In dit eerste project gaan we een REST endpoint ter beschikking stellen waarmee de brouwers uit een opgegeven stad in JSON-formaat worden teruggeven.

Om te beginnen moeten we dus een REST endpoint ter beschikking kunnen stellen. Hiervoor hebben we Spring MVC, Tomcat en Jackson nodig. Heel wat dependencies dus. Maar ook hier is er één starter die al deze dependencies bundelt: spring-boot-starter-web.

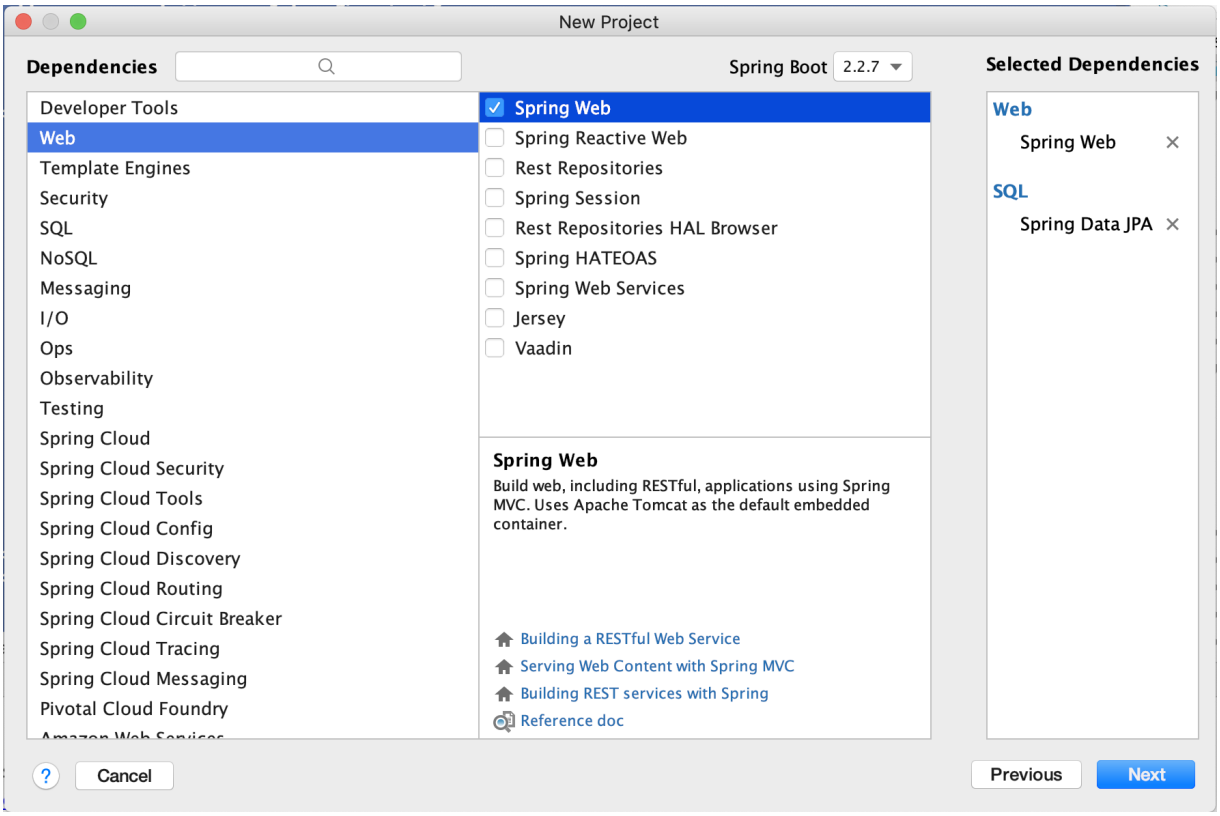
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring MVC is geen implementatie van JAX-RS. Spring MVC is het alternatief voor de JAX-RS standaard binnen Spring. Als je toch liever een JAX-RS implementatie gebruikt is dit ook perfect mogelijk.

Een applicatie die gebruikmaakt van Spring MVC heeft dus ook een web container nodig. Spring Boot zal dus automatisch een embedded Tomcat

toevoegen aan je project. Natuurlijk heb je steeds de mogelijkheid om een andere web container te kiezen.

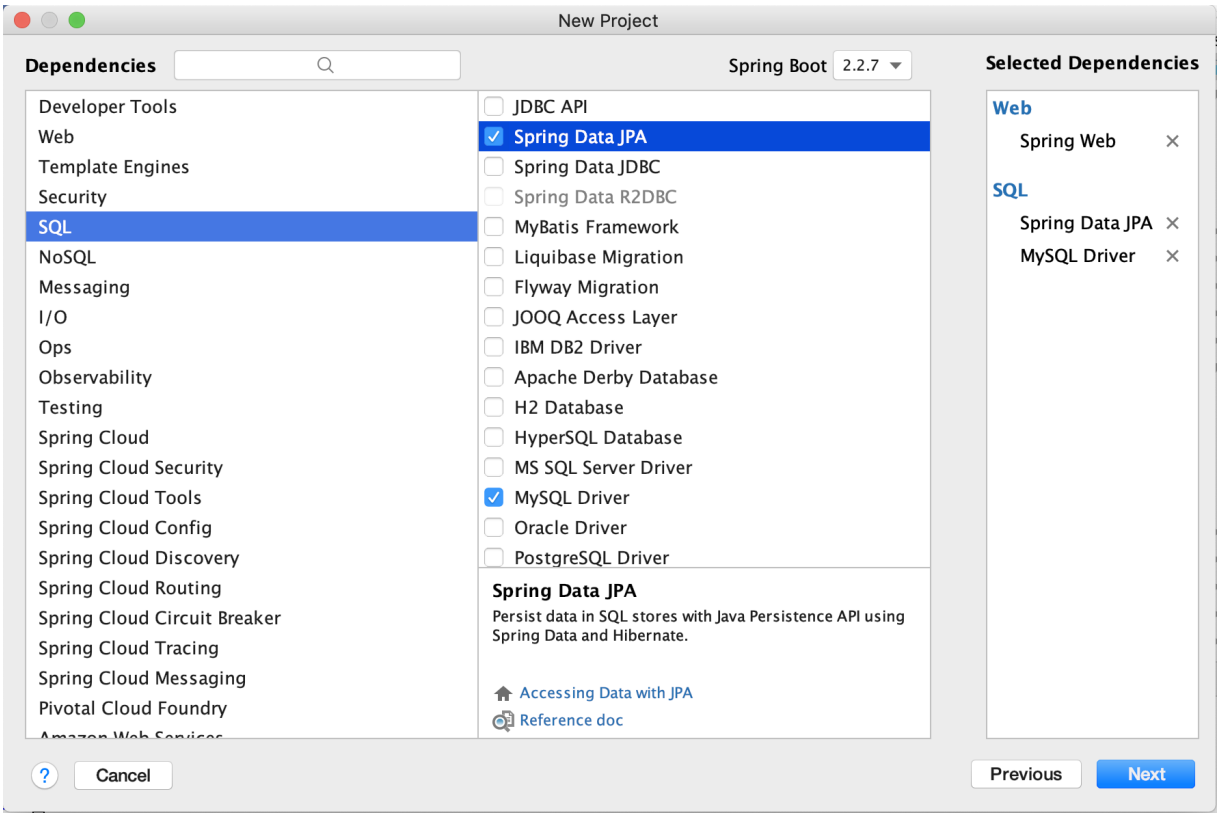
Tenslotte is Jackson een populaire library in java om java-objecten naar JSON te converteren en vice versa.



Naast de spring-boot-starter-web gaan we nog een tweede starter toevoegen. De spring-boot-starter-jpa. Deze hebben we nodig om verbinding te maken met de databank. Door gebruik te maken van de starter worden Hibernate, JPA en JDBC libraries toegevoegd aan het project en kan je zelfs zonder configuratie beginnen ontwikkelen. Indien je een in-memory databank zoals H2 toevoegt als dependency aan je project hoef je namelijk geen database URL en credentials te configureren. De verbinding met de H2 databank wordt dan door Spring Boot achter de schermen volledig in orde gebracht. In ons geval willen we verbinding maken met de reeds bestaande studentdb. We voorzien daarom straks de nodige configuratie in het bestand application.properties en voegen verder ook de gepaste driver dependency toe.

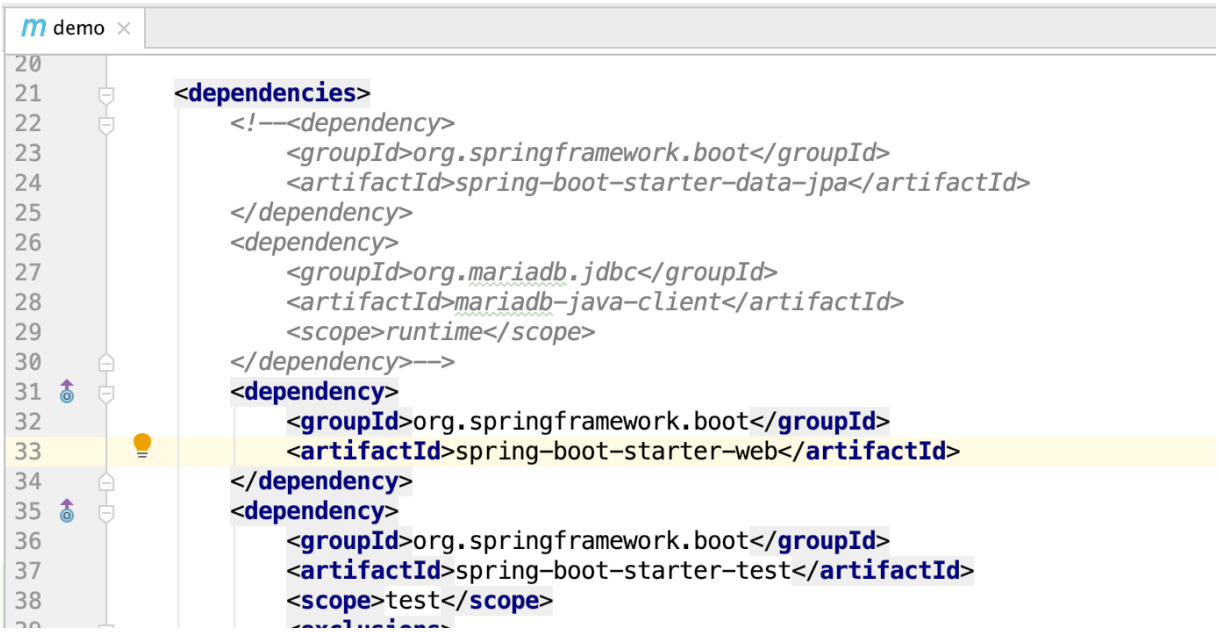
Spring Data JPA geeft je nog extra functionaliteit bovenop JPA. Het levert met name alle functionaliteit beschreven in de JPA specificatie zoals entities, associaties en query mogelijkheden. Maar naast deze basisfunctionaliteit

biedt Spring Data JPA data access objecten zonder code en het genereren van queries aan de hand van de naam van een methode in het data access object.



Doorloop de wizard nu verder om uiteindelijk je Spring Boot project te openen in IntelliJ IDEA.

Voorlopig gaan we spring-boot-starter-jpa in pom.xml nog even in commentaar zetten. Indien we dit niet doen krijgen we een foutmelding wanneer we het Spring Boot project starten omdat we de configuratie van de databank in application.properties nog niet hebben vervolledigd.

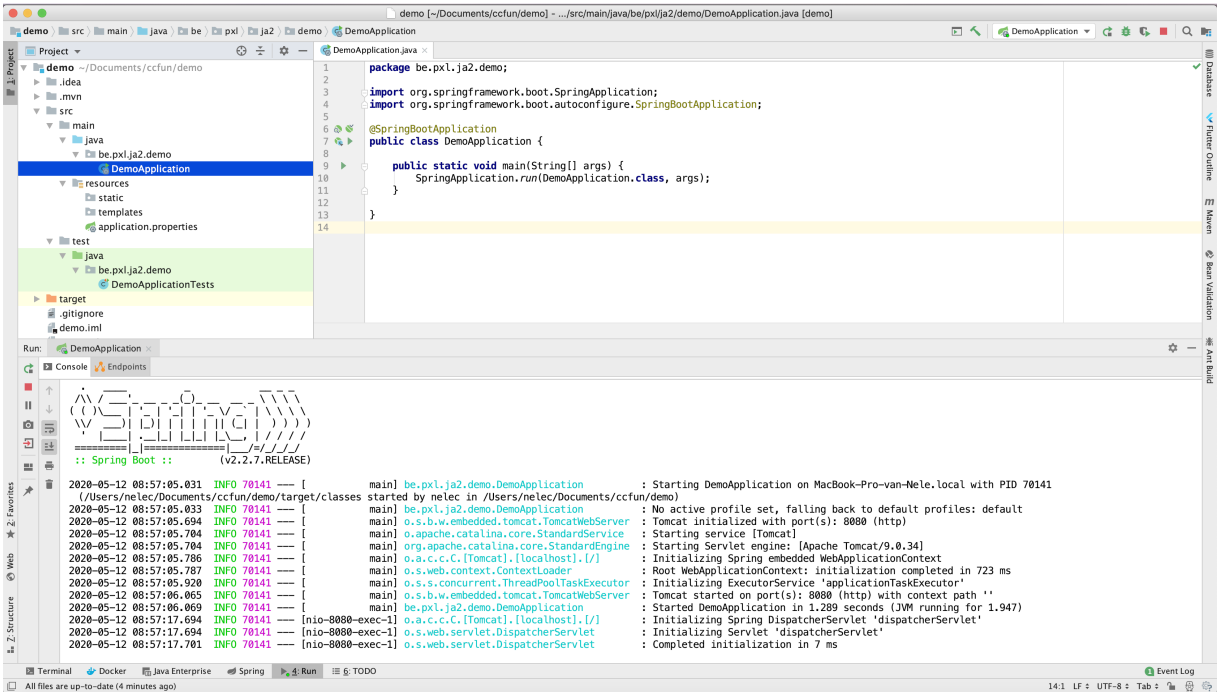


2.2 Spring Boot project uitvoeren

Het startpunt van de Spring Boot applicatie is de klasse met de main-methode die geannoteerd is met @SpringBootApplication. Deze klasse vind je terug in de folder /src/main/java.

```
1  package be.pxl.ja2.demo;
2
3  import org.springframework.boot.SpringApplication;
4  import
     org.springframework.boot.autoconfigure.SpringBootApplication;
5
6  @SpringBootApplication
7  public class DemoApplication {
8
9      public static void main(String[] args) {
10         SpringApplication.run(DemoApplication.class, args);
11     }
12
13 }
```

Door het uitvoeren van deze main-klasse start je het Spring Boot project.



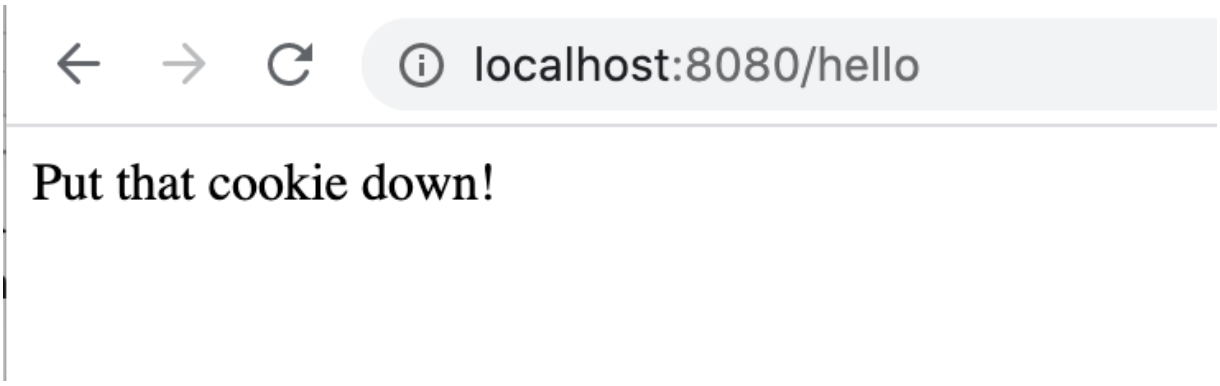
Het enige dat je nu kan zien van de Spring Boot applicatie is de whitelabel error page die je krijgt als je een GET doet op http://localhost:8080.

2.3 GreetingController

Klassen gemarkeerd met @RestController, worden door Spring MVC gebruikt om web requests af te handelen. Wanneer Spring MVC onze GreetingController heeft aangemaakt, zal de init() methode worden uitgevoerd om de mogelijke boodschappen te initialiseren. @GetMapping mapt de url /hello op de methode doGreeting(). Wanneer we de URL aanroepen via bijvoorbeeld een browers, zal de methode een begroeting in tekstformaat geven.

```
1 package be.pxl.ja2.demo.rest;
2
3 import org.springframework.web.bind.annotation.GetMapping;
4 import
     org.springframework.web.bind.annotation.RestController;
5
6 import javax.annotation.PostConstruct;
7 import java.util.ArrayList;
8 import java.util.List;
```

```
9  import java.util.Random;
10
11
12 @RestController
13 public class GreetingController {
14
15     private List<String> messages = new ArrayList<>();
16     private static final Random RANDOM = new Random();
17
18     @PostConstruct
19     public void init() {
20         messages.add("Peek-a-boo!");
21         messages.add("Howdy-doody!");
22         messages.add("My name's Ralph, and I'm a bad guy.");
23         messages.add("I come in peace!");
24         messages.add("Put that cookie down!");
25
26     }
27
28     @GetMapping("/hello")
29     public String doGreeting() {
30         return messages.get(RANDOM.nextInt(messages.size()));
31     }
32
33 }
```



2.4 Gegevens opvragen

We maken voor deze demotoepassing gebruik van de bestaande studentdb. We gaan met name data ophalen uit en wegschrijven naar de Brewers tabel.

Zorg ervoor dat je de spring-starter-data-jpa dependency terug enabled in je pom.xml.

Om toegang te krijgen tot onze databank moeten we eerst de URL en credentials van de databank in het bestand application.properties plaatsen. Dit bestand vind je terug in de folder src/main/resources.

```
spring.datasource.url=jdbc:mariadb://localhost:3307/studentdb
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.hibernate.naming.physical-strategy=

   org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
```

By default gebruikt Spring Boot tabelnamen in lower-case. De tabel Brewers in de studentdb start met een hoofdletter. Daarom moeten we de naming strategy aanpassen. Als je dit niet doet krijg je bij het uitvoeren van queries op de Brewers tabel de volgende foutmelding:

```
java.sql.SQLException: Table 'studentdb.brewers' doesn't
   exist
```

Om data uit de Brewers tabel op te halen hebben we dus eerst en vooral een entity-klasse nodig.

```
package be.pxl.ja2.demo.domain;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

```

```
10 @Entity
11 @Table(name = "Brewers")
12 @NamedQuery(name = "findByCity", query = "select b from
      Brewer as b where b.city = :city")
13 public class Brewer {
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private Integer id;
18     private String name;
19     private String address;
20     private String zipCode;
21     private String city;
22     private int turnover;
23
24     public Integer getId() {
25         return id;
26     }
27
28     public String getName() {
29         return name;
30     }
31
32     public void setName(String name) {
33         this.name = name;
34     }
35
36     public String getAddress() {
37         return address;
38     }
39
40     public void setAddress(String address) {
41         this.address = address;
42     }
43
44     public String getZipCode() {
45         return zipCode;
46     }
```

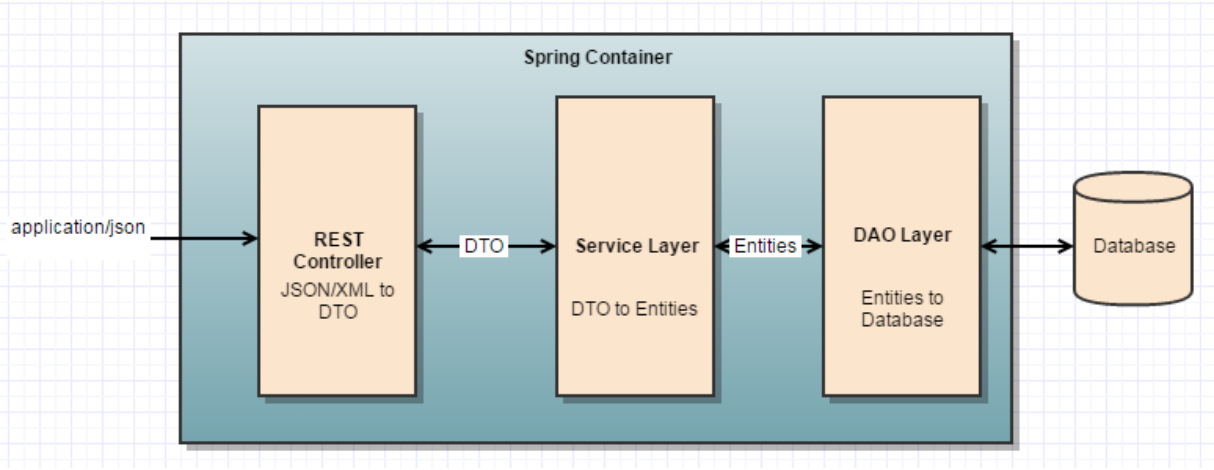
```
47
48      public void setZipCode(String zipCode) {
49          this.zipCode = zipCode;
50      }
51
52      public String getCity() {
53          return city;
54      }
55
56      public void setCity(String city) {
57          this.city = city;
58      }
59
60      public int getTurnover() {
61          return turnover;
62      }
63
64      public void setTurnover(int turnover) {
65          this.turnover = turnover;
66      }
67
68      @Override
69      public String toString() {
70          return "Brewer [id=" + id + ", name=" + name + "]";
71      }
72
73  }
```

Door gebruik te maken van Spring Data kunnen we onze persistence-laag nog vereenvoudigen. We voorzien een DAO interface die een uitbreiding is van de interface JPARepository. JPARepository is een generieke interface en je zal 2 klassen moeten opgeven: de entity-klasse zelf en het datatype van de primairy key (primaire sleutel). Hierdoor zal Spring Data de CRUD (create-read-update-delete) functionaliteiten voor je entity-objecten kunnen aanbieden. Verder is Spring Data in staat om aan de hand van de naam van een methode de bijhorende query te genereren. Zo zie je in onze implementatie 2 automatic custom queries: findBrewersByCity en findBrewerByNameAndCity. Het is ook nog mogelijk om eigen (custom) queries toe te voegen waarbij je zelf

de query-implementatie voorziet.

```
1  package be.pxl.ja2.demo.dao;
2
3  import be.pxl.ja2.demo.domain.Brewer;
4  import org.springframework.data.jpa.repository.JpaRepository;
5  import org.springframework.stereotype.Repository;
6  import java.util.List;
7
8  @Repository
9  public interface BrewerDao extends JpaRepository<Brewer,
       Integer> {
10
11     List<Brewer> findBrewersByCity(String city);
12     Brewer findBrewerByNameAndCity(String name, String city);
13 }
```

Vervolgens voorzien we onze service-laag die onze DAO interface gaat gebruiken. De service-laag zal nooit entity-objecten retourneren. Entity-objecten worden door onze service-laag steeds geconverteerd naar dto's (data transfer objecten). Merk op dat niet alle velden uit het entity-object in het data transfer object moeten worden toegevoegd. Onze backend zal enkel de nodige informatie beschikbaar stellen van zijn eindgebruiker.




```
1  package be.pxl.ja2.demo.dto;
2
3  public class BrewerDTO {
4      private Integer id;
5      private String name;
```

```
6       private String city;
7
8       public Integer getId() {
9           return id;
10      }
11
12      public void setId(Integer id) {
13          this.id = id;
14      }
15
16      public String getName() {
17          return name;
18      }
19
20      public void setName(String name) {
21          this.name = name;
22      }
23
24      public String getCity() {
25          return city;
26      }
27
28      public void setCity(String city) {
29          this.city = city;
30      }
31  }
```

```
1   package be.pxl.ja2.demo.service;
2
3   import be.pxl.ja2.demo.dao.BrewerDao;
4   import be.pxl.ja2.demo.domain.Brewer;
5   import be.pxl.ja2.demo.dto.BrewerDTO;
6   import be.pxl.ja2.demo.exceptions.NonUniqueBrewerException;
7   import be.pxl.ja2.demo.rest.resources.BrewerCreateResource;
8   import
        org.springframework.beans.factory.annotation.Autowired;
9   import org.springframework.stereotype.Service;
10
```

```
11 import java.util.List;
12 import java.util.stream.Collectors;
13
14 @Service
15 public class BrewerService {
16     @Autowired
17     private BrewerDao brewerDao;
18
19     public List<BrewerDTO> findBrewersByCity(String city) {
20         List<Brewer> brewers =
   brewerDao.findBrewersByCity(city);
21         return mapBrewers(brewers);
22     }
23
24     private static List<BrewerDTO> mapBrewers(List<Brewer>
   brewers) {
25         return
   brewers.stream().map(BrewerService::mapBrewer).collect(Collectors.toL
26     }
27
28     private static BrewerDTO mapBrewer(Brewer brewer) {
29         BrewerDTO result = new BrewerDTO();
30         result.setId(brewer.getId());
31         result.setName(brewer.getName());
32         result.setCity(brewer.getCity());
33         return result;
34     }
35 }
```

Tenslotte voorzien we een BrewersController die de data ter beschikking stelt aan eindgebruikers (frontend, mobiele app,...). We voorzien een Rest endpoint met een path variabele in de url.

```
1 package be.pxl.ja2.demo.rest;
2
3 import be.pxl.ja2.demo.dto.BrewerDTO;
4 import be.pxl.ja2.demo.exceptions.NonUniqueBrewerException;
5 import be.pxl.ja2.demo.rest.resources.BrewerCreateResource;
```

```
6  import be.pxl.ja2.demo.service.BrewerService;
7  import
     org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.http.HttpStatus;
9  import org.springframework.web.bind.annotation.GetMapping;
10 import org.springframework.web.bind.annotation.PathVariable;
11 import org.springframework.web.bind.annotation.PostMapping;
12 import org.springframework.web.bind.annotation.RequestBody;
13 import
     org.springframework.web.bind.annotation.RequestMapping;
14 import
     org.springframework.web.bind.annotation.RestController;
15 import
     org.springframework.web.server.ResponseStatusException;
16
17 import java.util.List;
18
19 @RestController
20 @RequestMapping(path = "brewers")
21 public class BrewersController {
22
23     @Autowired
24     private BrewerService brewerService;
25
26     @GetMapping("{city}")
27     public List<BrewerDTO>
     findBrewersByCity(@PathVariable("city") String city) {
28         return brewerService.findBrewersByCity(city);
29     }
30 }
```

2.5 Gegevens opslaan

We willen nu ook een Rest endpoint voorzien om een nieuwe brouwer toe te voegen in de databank. Een brouwer mag enkel toegevoegd word als de combinatie naam en gemeente uniek is.

Eerst breiden we onze BrewersController verder uit. De gegevens van de nieuwe brouwer worden in json-formaat in de request body doorgestuurd. We maken een BrewerCreateResource en gebruiken de @RequestBody annotatie om het binnenkomende json request te converteren naar een BrewerCreateResource-object.

```
1 package be.pxl.ja2.demo.rest.resources;
2
3 public class BrewerCreateResource {
4     private String name;
5     private String address;
6     private String zipCode;
7     private String city;
```

```
8       private int turnover;
9
10      public String getName() {
11          return name;
12      }
13
14      public void setName(String name) {
15          this.name = name;
16      }
17
18      public String getAddress() {
19          return address;
20      }
21
22      public void setAddress(String address) {
23          this.address = address;
24      }
25
26      public String getZipCode() {
27          return zipCode;
28      }
29
30      public void setZipCode(String zipCode) {
31          this.zipCode = zipCode;
32      }
33
34      public String getCity() {
35          return city;
36      }
37
38      public void setCity(String city) {
39          this.city = city;
40      }
41
42      public int getTurnover() {
43          return turnover;
44      }
45
```

```
46      public void setTurnover(int turnover) {
47          this.turnover = turnover;
48      }
49  }
```

```
1   package be.pxl.ja2.demo.rest;
2
3   import be.pxl.ja2.demo.dto.BrewerDTO;
4   import be.pxl.ja2.demo.exceptions.NonUniqueBrewerException;
5   import be.pxl.ja2.demo.rest.resources.BrewerCreateResource;
6   import be.pxl.ja2.demo.service.BrewerService;
7   import
      org.springframework.beans.factory.annotation.Autowired;
8   import org.springframework.http.HttpStatus;
9   import org.springframework.web.bind.annotation.PathVariable;
10  import org.springframework.web.bind.annotation.PostMapping;
11  import org.springframework.web.bind.annotation.RequestBody;
12  import
      org.springframework.web.bind.annotation.RequestMapping;
13  import
      org.springframework.web.bind.annotation.RestController;
14  import
      org.springframework.web.server.ResponseStatusException;
15
16  import java.util.List;
17
18  @RestController
19  @RequestMapping(path = "brewers")
20  public class BrewersController {
21
22      @Autowired
23      private BrewerService brewerService;
24
25      @PostMapping
26      public BrewerDTO createBrewer(@RequestBody
      BrewerCreateResource brewerCreateResource) {
27          try {
28              return
```

```
    brewerService.createBrewer(brewerCreateResource);
29          } catch (NonUniqueBrewerException e) {
30              throw new
    ResponseStatusException(HttpStatus.BAD_REQUEST,
    e.getMessage());
31          }
32
33      }
34  }
```

Indien er reeds een brouwer met de opgegeven naam en gemeente bestaat dan zal onze service-laag een NonUniqueBrewerException opgooien. In dat geval willen we een gepaste http status code met een duidelijke foutboodschap aan de gebruiker van ons Rest endpoint geven. Dit kunnen we op een elegante manier oplossen met de ResponseStatusException. De ResponseStatusException wordt afgehandeld door Spring die ervoor zorgt dat een response gestuurd wordt met volgend formaat:

```
1 {
2   "timestamp": "2020-05-13T09:40:02.969+0000",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "There already exists a brewer with name [New
     Brewer] in city [Nieuwerkerke]",
6   "path": "/brewers"
7 }
```

In onze BrewerService klasse implementeren we nu logica voor het aanmaken van de brouwer. Eerst wordt gecontroleerd of er reeds een brouwer bestaat met de opgegeven naam en gemeente. Indien de brouwer uniek is gebruiken we de save methode uit de BrewerDao om het nieuw aangemaakt entity-object op te slaan in de databank.

```
1 package be.pxl.ja2.demo.service;
2
3 import be.pxl.ja2.demo.dao.BrewerDao;
4 import be.pxl.ja2.demo.domain.Brewer;
5 import be.pxl.ja2.demo.dto.BrewerDTO;
6 import be.pxl.ja2.demo.exceptions.NonUniqueBrewerException;
```

```
7  import be.pxl.ja2.demo.rest.resources.BrewerCreateResource;
8  import
     org.springframework.beans.factory.annotation.Autowired;
9  import org.springframework.stereotype.Service;
10
11 import java.util.List;
12 import java.util.stream.Collectors;
13
14 @Service
15 public class BrewerService {
16     @Autowired
17     private BrewerDao brewerDao;
18
19     public List<BrewerDTO> findBrewersByCity(String city) {
20         List<Brewer> brewers =
     brewerDao.findBrewersByCity(city);
21         return mapBrewers(brewers);
22     }
23
24
25     private static List<BrewerDTO> mapBrewers(List<Brewer>
     brewers) {
26         return
     brewers.stream().map(BrewerService::mapBrewer).collect(Collectors.toL
27     }
28
29     private static BrewerDTO mapBrewer(Brewer brewer) {
30         BrewerDTO result = new BrewerDTO();
31         result.setId(brewer.getId());
32         result.setName(brewer.getName());
33         result.setCity(brewer.getCity());
34         return result;
35     }
36
37
38     private static Brewer mapBrewer(BrewerCreateResource
     brewerCreateResource) {
39         Brewer brewer = new Brewer();
```

```
40          brewer.setName(brewerCreateResource.getName());
41          brewer.setAddress(brewerCreateResource.getAddress());
42          brewer.setCity(brewerCreateResource.getCity());
43          brewer.setZipCode(brewerCreateResource.getZipCode());
44
    brewer.setTurnover(brewerCreateResource.getTurnover());
45          return brewer;
46      }
47
48      public BrewerDTO createBrewer(BrewerCreateResource
    brewerCreateResource) throws NonUniqueBrewerException {
49          Brewer existingBrewer =
    brewerDao.findBrewerByNameAndCity(brewerCreateResource.getName(),
    brewerCreateResource.getCity());
50          if (existingBrewer != null) {
51              throw new NonUniqueBrewerException("There
    already exists a brewer with name [" +
    brewerCreateResource.getName() + "] in city [" +
    brewerCreateResource.getCity() + "]");
52          }
53          Brewer newBrewer =
    brewerDao.save(mapBrewer(brewerCreateResource));
54          return mapBrewer(newBrewer);
55      }
56  }
```

2.6 Unit testing BrewerDao

De starter voor unit testing wordt automatisch toegevoegd in een nieuw Spring Boot project. Deze starter zorgt dat je junit en mockito ter beschikking hebt. Als je custom queries toevoegt in je DAO interface, kan je deze testen door gebruik te maken van een in-memory database (bijv. hsqldb of h2).

Eerst voeg je de dependency voor de in-memory database driver toe in je pom.xml.

```
1       <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
            <exclusions>
                <exclusion>
                    <groupId>org.junit.vintage</groupId>
                    <artifactId>junit-vintage-engine</artifactId>
                </exclusion>
            </exclusions>
        </dependency>

        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <scope>test</scope>
        </dependency>
```

Je hoeft verder geen configuratie toe te voegen als je gebruikmaakt van de annotatie @DataJpaTest. In onderstaand voorbeeld vind je 2 testen terug. De eerste test de automatic custom query findBrewersByCity, de tweede test is voor findBrewersByNameAndCity.

```
package be.pxl.ja2.demo.dao;

import be.pxl.ja2.demo.builder.BrewerBuilder;
import be.pxl.ja2.demo.domain.Brewer;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import
   org.springframework.beans.factory.annotation.Autowired;
import
   org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import
   org.springframework.test.context.junit.jupiter.SpringExtension;

import java.util.List;

```

```
13 import static org.junit.jupiter.api.Assertions.assertEquals;
14 import static org.junit.jupiter.api.Assertions.assertNotNull;
15
16 @ExtendWith(SpringExtension.class)
17 @DataJpaTest
18 class BrewerDaoTest {
19
20     @Autowired
21     private BrewerDao brewerDao;
22
23     @Test
24     void brewersCanBeFoundByCity() {
25
   brewerDao.save(BrewerBuilder.aBrewer().withName("Brewer
   A").withCity("Bree").build());
26
   brewerDao.save(BrewerBuilder.aBrewer().withName("Brewer
   B").withCity("Genk").build());
27
   brewerDao.save(BrewerBuilder.aBrewer().withName("Brewer
   C").withCity("Bree").build());
28
29         List<Brewer> result =
   brewerDao.findBrewersByCity("Bree");
30
31         assertNotNull(result);
32         assertEquals(2, result.size());
33         long countBree = result.stream().filter(b ->
   b.getCity().equals("Bree")).count();
34         assertEquals(2, countBree);
35      }
36
37      @Test
38      void brewerCanBeFoundByNameAndCity() {
39
   brewerDao.save(BrewerBuilder.aBrewer().withName("Brewer
   X").withCity("Bree").build());
40
```

```
    brewerDao.save(BrewerBuilder.aBrewer().withName("Brewer
    Y").withCity("Genk").build());
41
42          Brewer result =
    brewerDao.findBrewerByNameAndCity("Brewer X", "Bree");
43
44          assertNotNull(result);
45          assertEquals("Bree", result.getCity());
46          assertEquals("Brewer X", result.getName());
47      }
48  }
```

In het setup gedeelte van een unit test moeten we vaak meerdere objecten initialiseren. Het builder patroon is een goede manier om deze objecten aan te maken. Door gebruik te maken van het builder patroon hou je de testen kleiner en het aanmaken van de objecten eenvoudiger. Je kan in het builder patroon ook default waarden toekennen aan de eigenschappen van een object, waardoor het initialiseren van objecten nog eenvoudiger wordt.

```
1  package be.pxl.ja2.demo.builder;
2
3  import be.pxl.ja2.demo.domain.Brewer;
4
5  public final class BrewerBuilder {
6
7      private Integer id;
8      private String name;
9      private String address;
10     private String zipCode;
11     private String city;
12     private int turnover;
13
14     private BrewerBuilder() {}
15
16     public static BrewerBuilder aBrewer() { return new
    BrewerBuilder(); }
17
```

```
18      public BrewerBuilder withId(Integer id) {
19          this.id = id;
20          return this;
21      }
22
23      public BrewerBuilder withName(String name) {
24          this.name = name;
25          return this;
26      }
27
28      public BrewerBuilder withAddress(String address) {
29          this.address = address;
30          return this;
31      }
32
33      public BrewerBuilder withZipCode(String zipCode) {
34          this.zipCode = zipCode;
35          return this;
36      }
37
38      public BrewerBuilder withCity(String city) {
39          this.city = city;
40          return this;
41      }
42
43      public BrewerBuilder withTurnover(int turnover) {
44          this.turnover = turnover;
45          return this;
46      }
47
48      public Brewer build() {
49          Brewer brewer = new Brewer();
50          brewer.setId(id);
51          brewer.setName(name);
52          brewer.setAddress(address);
53          brewer.setZipCode(zipCode);
54          brewer.setCity(city);
55          brewer.setTurnover(turnover);
```

```
56          return brewer;
57      }
58  }
```

2.7 Unit testing BrewerService

Tenslotte voegen we nog unit testen toe voor de createBrewer-methode in de klasse BrewerService. In het geval dat we onze service-laag gaan testen, gebruiken we een mock-object voor de BrewerDao.

Er zijn 2 scenario's. Ofwel bestaat er reeds een brouwer met de opgegeven naam en gemeente, dan krijg je dus een NonUniqueBrewerException. Ofwel is het een nieuwe brouwer, en dan gaan we controleren dat er een entity-object wordt aangemaakt dat via de BrewerDao opgeslagen wordt in de databank. De test brewerCorrectlySaved gaat controleren dat het entity-object correct wordt doorgegeven aan de save-methode van de BrewerDao. Omdat we het entity-object willen onderscheppen, om zo te controleren of alle velden correct zijn ingevuld, gebruiken we een ArgumentCaptor. De test returnsBrewerDTO controleert of de createBrewer-methode het correcte resultaat teruggeeft.

```
1  package be.pxl.ja2.demo.service;
2
3  import be.pxl.ja2.demo.dao.BrewerDao;
4  import be.pxl.ja2.demo.domain.Brewer;
5  import be.pxl.ja2.demo.dto.BrewerDTO;
6  import be.pxl.ja2.demo.exceptions.NonUniqueBrewerException;
7  import be.pxl.ja2.demo.rest.resources.BrewerCreateResource;
8  import org.junit.jupiter.api.BeforeEach;
9  import org.junit.jupiter.api.Test;
10 import org.junit.jupiter.api.extension.ExtendWith;
11 import org.mockito.ArgumentCaptor;
12 import org.mockito.Captor;
13 import org.mockito.InjectMocks;
14 import org.mockito.Mock;
15 import org.mockito.junit.jupiter.MockitoExtension;
16
```

```
17 import static org.junit.jupiter.api.Assertions.assertEquals;
18 import static org.junit.jupiter.api.Assertions.assertThrows;
19 import static org.mockito.ArgumentMatchers.any;
20 import static org.mockito.Mockito.verify;
21 import static org.mockito.Mockito.when;
22
23 @ExtendWith(MockitoExtension.class)
24 public class BrewerServiceCreateBrewerTest {
25     private static final Integer ID = 101;
26     private static final String NAME = "White pony";
27     private static final String CITY = "Aalst";
28     private static final String ADDRESS = "Kerkstraat 5";
29     private static final int TURNOVER = 5000;
30     private static final String ZIP_CODE = "9300";
31     @Mock
32     private BrewerDao brewerDao;
33     @Mock
34     private Brewer brewer;
35     @InjectMocks
36     private BrewerService brewerService;
37     @Captor
38     private ArgumentCaptor<Brewer> brewerArgumentCaptor;
39
40     private BrewerCreateResource brewerCreateResource;
41     private Brewer createdBrewer;
42
43     @BeforeEach
44     public void init() {
45         brewerCreateResource = new BrewerCreateResource();
46         brewerCreateResource.setName(NAME);
47         brewerCreateResource.setAddress(ADDRESS);
48         brewerCreateResource.setCity(CITY);
49         brewerCreateResource.setTurnover(TURNOVER);
50         brewerCreateResource.setZipCode(ZIP_CODE);
51
52         createdBrewer = new Brewer();
53         createdBrewer.setId(ID);
54         createdBrewer.setName(NAME);
```

```
55          createdBrewer.setCity(CITY);
56      }
57
58      @Test
59      public void
    nonUniqueBrewerExceptionIsThrownWhenThereIsAlreadyABrewerWithTheGiver
    {
60          when(brewerDao.findBrewerByNameAndCity(NAME,
    CITY)).thenReturn(brewer);
61
62          assertThrows(NonUniqueBrewerException.class, () ->
    brewerService.createBrewer(brewerCreateResource));
63      }
64
65      @Test
66      public void returnsBrewerDTO() throws
    NonUniqueBrewerException {
67          when(brewerDao.findBrewerByNameAndCity(NAME,
    CITY)).thenReturn(null);
68
    when(brewerDao.save(any())).thenReturn(createdBrewer);
69
70          BrewerDTO brewerDTO =
    brewerService.createBrewer(brewerCreateResource);
71
72          assertEquals(NAME, brewerDTO.getName());
73          assertEquals(CITY, brewerDTO.getCity());
74          assertEquals(ID, brewerDTO.getId());
75      }
76
77      @Test
78      public void brewerCorrectlySaved() throws
    NonUniqueBrewerException {
79          when(brewerDao.findBrewerByNameAndCity(NAME,
    CITY)).thenReturn(null);
80
    when(brewerDao.save(any())).thenReturn(createdBrewer);
81
```

```
82          brewerService.createBrewer(brewerCreateResource);
83
84
    verify(brewerDao).save(brewerArgumentCaptor.capture());
85          Brewer createdBrewer =
    brewerArgumentCaptor.getValue();
86          assertEquals(NAME, createdBrewer.getName());
87          assertEquals(CITY, createdBrewer.getCity());
88          assertEquals(ZIP_CODE, createdBrewer.getZipCode());
89          assertEquals(ADDRESS, createdBrewer.getAddress());
90          assertEquals(TURNOVER, createdBrewer.getTurnover());
91      }
92
93  }
```