



samen sterk voor werk

# Spring advanced

Deze cursus is eigendom van de VDAB©

## Inhoudsopgave

<b>1</b>	<b>INLEIDING.....</b>	<b>6</b>
1.1	Doelstelling.....	6
1.2	Vereiste voorkennis.....	6
1.3	Nodige software .....	6
1.4	Website .....	6
1.5	Database.....	6
1.6	Unit testen en integration testen.....	6
<b>2</b>	<b>PROJECT OPZETTEN.....</b>	<b>7</b>
2.1	application.properties .....	7
2.2	Controllers.....	7
2.3	Static content .....	8
2.4	Thymeleaf.....	8
2.5	Thymeleaf fragment.....	9
2.6	JPA project.....	10
2.7	Value objecten .....	10
2.8	Entities.....	10
<b>3</b>	<b>SPRING DATA.....</b>	<b>13</b>
3.1	Automatisch gegenereerde repository classes .....	13
3.1.1	Automatische String naar Entity converters .....	16
3.2	Spring data en Entity graphs .....	17
3.3	Pagineren .....	19
<b>4</b>	<b>TESTEN MET EEN IN-MEMORY DATABASE .....</b>	<b>22</b>
4.1	Unique constraint.....	23
<b>5</b>	<b>VALIDATION ANNOTATIONS MAKEN.....</b>	<b>24</b>
5.1	Samenstelling van andere validation annotations .....	24

5.2	Eigen bean validation in detail .....	25
5.3	Object properties ten opzichte van mekaar valideren.....	26
<b>6</b>	<b>SESSION SCOPE, VALIDATION GROUPS.....</b>	<b>28</b>
6.1	Session scope .....	28
6.2	Validation groups .....	28
<b>7</b>	<b>MEERTALIGE WEBSITES.....</b>	<b>32</b>
7.1	Taal en het land van de gebruiker .....	34
7.1.1	AcceptHeaderLocaleResolver .....	34
7.1.2	FixedLocaleResolver .....	34
7.1.3	SessionLocaleResolver .....	34
7.1.4	LocaleChangeInterceptor .....	34
7.1.5	CookieLocaleResolver .....	35
7.1.6	Taal en het land lezen in een controller bean.....	36
<b>8</b>	<b>REST .....</b>	<b>37</b>
8.1	Entities identificeren met URI's.....	37
8.1.1	URI die alle entities van hetzelfde type voorstelt .....	37
8.1.2	URI die één entity voorstelt .....	37
8.1.3	URI die een verzameling entities aangeduid met een groepsnaam voorstelt .....	37
8.1.4	URI die een verzameling entities die je zoekt op andere zoekcriteria voorstelt .....	37
8.1.5	Associaties tussen entities .....	37
8.2	HTTP methods .....	38
8.2.1	GET: entity of entities lezen .....	38
8.2.2	POST: entity toevoegen .....	38
8.2.3	PUT: entity wijzigen .....	38
8.2.4	DELETE: entity verwijderen.....	38
8.3	Formaat van de data .....	38
8.3.1	Content-type header .....	38
8.4	Response status code.....	39
8.5	HATEOAS .....	39
<b>9</b>	<b>NIET-BROWSER CLIENTS REST REQUESTS.....</b>	<b>41</b>
9.1	FiliaalController .....	41
9.2	FiliaalRestController .....	41
9.3	Java objecten omzetten van en naar XML en JSON .....	41
9.3.1	Libraries .....	41
9.3.2	LocalDate .....	41
9.3.3	Annotations .....	42

9.4	GET request naar één entity.....	42
9.4.1	FiliaalRestController.....	42
9.4.2	Postman.....	43
9.4.3	Status code 404 (Not Found) .....	43
9.5	DELETE request.....	44
9.6	POST request .....	45
9.6.1	@RequestBody .....	45
9.7	PUT request.....	46
9.8	HATEOAS .....	47
9.8.1	Configuratie per REST controller.....	47
9.8.2	EntityLinks.....	47
9.8.3	Location response header.....	47
9.8.4	Link elementen opnemen in de response body.....	48
9.8.5	Response met een verzameling entities .....	49
<b>10</b>	<b>INTEGRATION TEST VAN EEN REST SERVICE.....</b>	<b>51</b>
<b>11</b>	<b>REST CLIENT .....</b>	<b>53</b>
11.1	RestTemplate .....	53
11.1.1	DELETE request om een filiaal te verwijderen .....	53
11.1.2	POST request om een filiaal toe te voegen.....	53
11.1.3	PUT request om een filiaal te wijzigen.....	53
11.1.4	GET request om een filiaal te lezen .....	54
11.2	Voorbeeld.....	54
11.2.1	Structuur van de XML data en/of JSON data .....	54
11.2.2	URI van de service.....	55
11.2.3	Exceptions.....	55
11.2.4	RestClient.....	55
<b>12</b>	<b>JAVASCRIPT CODE ALS REST CLIENT (AJAX REQUESTS).....</b>	<b>58</b>
<b>13</b>	<b>MAIL STUREN .....</b>	<b>60</b>
13.1	Mail server configuratie .....	60
13.2	Bean die de mail stuurt .....	60
13.3	Services layer.....	61
13.4	Mail met opmaak en een hyperlink .....	61
<b>14</b>	<b>ASYNCHRONE VERWERKING .....</b>	<b>63</b>
14.1	@Async.....	63
14.2	@EnableAsync.....	63

14.3	Voorbeeld.....	63
<b>15</b>	<b>TERUGKERENDE TAKEN.....</b>	<b>64</b>
15.1	@Scheduled .....	64
15.2	@EnableScheduling.....	64
15.3	Voorbeeld.....	64
<b>16</b>	<b>JMS .....</b>	<b>66</b>
16.1	ActiveMQ.....	66
16.2	Boodschappen versturen .....	67
16.3	Boodschappen ontvangen.....	69
<b>17</b>	<b>CUSTOM ERROR PAGES.....</b>	<b>71</b>
17.1	404.....	71
17.2	500.....	71
<b>18</b>	<b>SPRING SECURITY.....</b>	<b>72</b>
18.1	Security woordenschat.....	72
18.2	Configuratie .....	72
18.2.1	pom.xml .....	72
18.2.2	Default security .....	72
18.2.3	Java config .....	73
18.3	Eigen 403 (forbidden) pagina .....	74
18.4	CSRF.....	74
18.5	Eigen inlogpagina .....	76
18.6	Login fouten .....	76
18.7	Expliciet inloggen en uitloggen .....	77
18.8	Security in Thymeleaf .....	77
18.8.1	pom.xml .....	77
18.8.2	authorize tag.....	77
18.8.3	Naam van de huidige gebruiker .....	78
18.9	Principals en authorization in een database .....	78
18.9.1	Password encoding .....	78
18.9.2	Default tabel structuren .....	78
18.9.3	Afwijkende tabel structuren .....	79
18.10	Services layer .....	80

18.10.1	@PreAuthorize activeren.....	80
18.10.2	@PreAuthorize .....	80
18.11	Niet-browser REST clients .....	80
18.11.1	Authenticatie .....	80
<b>19</b>	<b>ASPECT ORIENTED PROGRAMMING (AOP) .....</b>	<b>82</b>
19.1	Je hebt al AOP toegepast bij CSS.....	82
19.2	Voordelen van het centraliseren van terugkerende taken .....	82
19.3	Pointcut expressie .....	83
19.3.1	Syntax .....	83
19.3.2	Onderdelen.....	83
19.3.3	Voorbeelden .....	83
19.3.4	Pointcuts combineren.....	83
19.4	Types advice .....	84
19.4.1	Before .....	84
19.4.2	After returning.....	85
19.4.3	After throwing .....	86
19.4.4	After finally .....	87
19.4.5	Around .....	88
19.4.6	Gecentraliseerde pointcut expressions .....	89
19.4.7	Meerdere advices op hetzelfde pointcut.....	89
<b>20</b>	<b>STAPPENPLAN .....</b>	<b>90</b>
<b>21</b>	<b>COLOFON.....</b>	<b>92</b>

# 1 INLEIDING

## 1.1 Doelstelling

Je leert werken met Spring: een open source Java framework om enterprise applicaties te maken. Spring helpt je in alle applicatie lagen: repositories, services, web, security, ...

## 1.2 Vereiste voorkennis

- Java
- JDBC
- Maven
- Unit testing
- Spring fundamentals
- JPA met Hibernate

## 1.3 Nodige software

- een JDK (Java Developer Kit) met versie 8 of hoger.
- een relationele database. Je gebruikt in de cursus MySQL. ([www.mysql.com](http://www.mysql.com))
- Tomcat 8.
- Eclipse IDE for Java EE Developers (versie Photon).

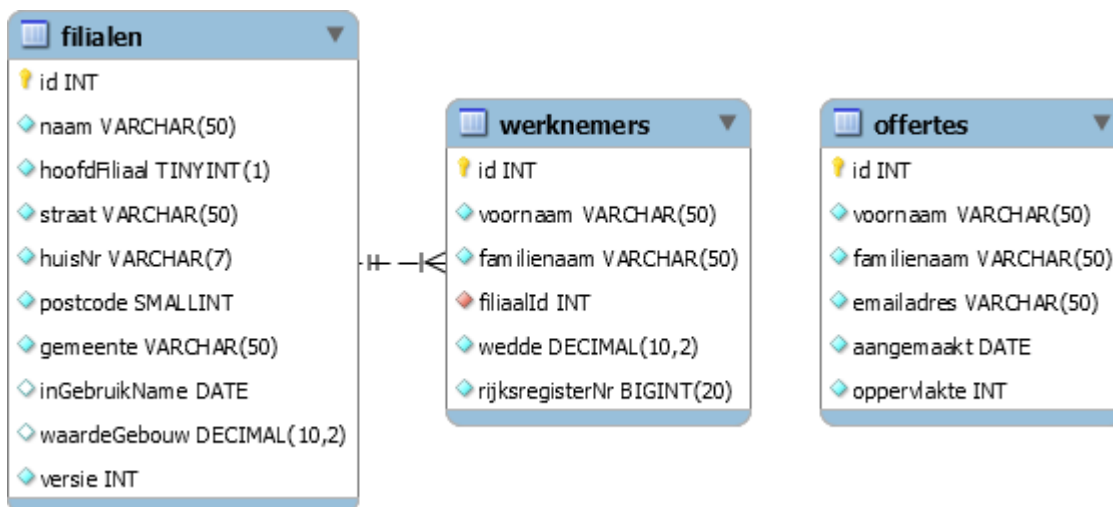
## 1.4 Website

Je maakt in deze cursus een website voor de firma Groene tenen. Deze firma legt gazons aan.

## 1.5 Database

De website werkt samen met de database groentenen.  
Je maakt deze database met het script groenetenen.sql.

De belangrijkste tables:



## 1.6 Unit testen en integration testen

Je maakt in deze cursus zelden unit testen en integration testen, om de omvang van de cursus te beperken. Je hebt hiervan veel voorbeelden gezien in de cursus “JPA met Hibernate”.  
Testen blijft in de praktijk belangrijk !

## 2 PROJECT OPZETTEN

Je maakt het Maven project op <http://start.spring.io>:

1. Je tikt be.vdab bij Group.
2. Je tikt groeneten bij Artifact.
3. Je tikt bij Search for dependencies dependencies, gescheiden door enter:
  - a. Web
  - b. DevTools
  - c. JPA
  - d. MySQL
  - e. H2  
Je gebruikt een H2 database bij het testen van je repositories layer.
  - f. Thymeleaf  
Je gebruikt Thymeleaf als alternatief voor JSP's.  
Dit laat je ook toe de website als een JAR bestand te verpakken.
  - g. HATEOAS  
Je laat hiermee andere applicaties jouw applicatie aanspreken.
  - h. Mail  
Je verstuurt hiermee emails.
  - i. JMS (ActiveMQ)  
Applicaties kunnen hiermee berichten uitwisselen.
4. Je kiest Generate Project.
5. Je unzippt de map uit het ZIP bestand in je workspace directory.
6. Je kiest in Eclipse in het menu File de opdracht Import.
7. Je kiest in de categorie Maven voor Existing Maven Project en je kiest Next.
8. Je kiest Browse en je kiest de map die je uitpakte bij puntje 5.
9. Je kiest Finish.

Je wijzigt in pom.xml de scope van de h2 dependency naar test.

Je heb de H2 database enkel tijdens het testen nodig.

Je voegt de jsoup dependency toe aan pom.xml:

```
<dependency>
  <groupId>org.jsoup</groupId>
  <artifactId>jsoup</artifactId>
  <version>1.11.3</version>
  <scope>runtime</scope>
</dependency>
```

### 2.1 application.properties

Je voegt volgende regels toe, waarmee je de databaseverbinding definieert:

```
spring.datasource.url=jdbc:mysql://localhost/groenetenen?useSSL=false
spring.datasource.username=cursist
spring.datasource.password=cursist
spring.jpa.hibernate.naming.physical-strategy=\
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE
```

### 2.2 Controllers

Je maakt een package be.vdab.groenetenen.web. Je maakt daarin IndexController:

```
package be.vdab.groenetenen.web;
// enkele imports
@Controller
@RequestMapping("/")
class IndexController {
  private static final String VIEW = "index";
```



```

private String begroeting() {
    int uur = LocalDateTime.now().getHour();
    if (uur >= 6 && uur < 12) {
        return "goede morgen";
    }
    if (uur >= 12 && uur < 18) {
        return "goede middag";
    }
    return "goede avond";
}

@GetMapping
ModelAndView index() {
    return new ModelAndView(VIEW, "begroeting", begroeting());
}

```

❶

- (1) Je geeft de request door aan de Thymeleaf pagina `index.html`.  
Je moet daarbij de extensie `html` niet vermelden.

## 2.3 Static content

Je plaatst static content (afbeeldingen, CSS bestanden, JavaScript bestanden) in de folder `src/main/resources/static` of in een subfolder van deze folder.

Je maakt in de folder `src/main/resources/static` een folder `css`.

Je kopieert `groenetenen.css` in deze folder.

Je maakt in de folder `src/main/resources/static` een folder `images`.

Je kopieert `403.jpg`, `404.jpg`, `500.jpg`, `gazon.jpg` en `groenetenen.ico` in deze folder.

## 2.4 Thymeleaf

Bij Thymeleaf zijn je pagina's HTML pagina's die je plaatst in `src/main/resources/templates`.

Je maakt in deze folder `index.html`:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head>
    <meta charset='UTF-8'>
    <link rel='icon' th:href='@{/images/groenetenen.ico}' type='image/x-icon'>
    <title>Groene tenen</title>
    <meta name='viewport' content='width=device-width,initial-scale=1'>
    <link rel='stylesheet' th:href='@{/css/groenetenen.css}'>
</head>
<body>
    <h1>Groene tenen, <span th:text='${begroeting}'></span>.</h1>
    <blockquote>De aanleg van een gezellig gazon.</blockquote>
    <img alt='gazon' th:src='@{/images/gazon.jpg}' class='fullwidth'>
</body>
</html>

```

❶

❷

❸

- (1) Je associeert de namespace <http://www.thymeleaf.org> met de prefix `th`.  
Je schrijft met elementen uit deze namespace expressies in de pagina.
- (2) Je maakt een URL met `@{nodigeURL}`, zoals je in een JSP een URL maakt met `<c:url>`.
- (3) Thymeleaf gebruikt de data `begroeting` (doorgekregen van de controller) als tekst tussen `<span>` en `</span>`.

Je kan de website uitproberen.



Je commit de sources en je publiceert op GitHub.

## 2.5 Thymeleaf fragment

Het <head>...</head> onderdeel van elke pagina is hetzelfde, behalve de tekst tussen <title> en </title>. Elke pagina zal ook eenzelfde menu bevatten. Je gaat het <head>...</head> onderdeel en het menu niet in iedere pagina herhalen. Als er in het menu een menupuntje blijkt, zou je elke pagina moeten aanpassen. In de plaats daarvan beschrijf je het <head>...</head> onderdeel één keer als een Thymeleaf fragment, dat je kan opnemen in elke pagina. Je beschrijft op dezelfde manier het menu één keer als Thymeleaf fragment, dat je kan opnemen in elke pagina.

Je beschrijft een fragment als onderdeel van een HTML pagina.

Één pagina kan één of meerdere fragments bevatten.

Je maakt fragments.html:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:fragment='head(title)'>                                ❶
    <meta charset='UTF-8'>
    <link rel='icon' th:href='@{/images/groenetenen.ico}' type='image/x-icon'>
    <title th:text='${title}'></title>                            ❷
    <meta name='viewport' content='width=device-width,initial-scale=1'>
    <link rel='stylesheet' th:href='@{/css/groenetenen.css}'>
</head>
<body>
    <nav th:fragment='menu'>                                    ❸
        <ul>
            <li><a th:href='@{/}'>&#8962;</a></li>
            <li><a href='#'>Filialen</a>
                <ul>
                    <li><a th:href='@{/filialen/vantotpostcode}'>Van tot postcode</a></li>
                    <li><a th:href='@{/filialen/perid}'>Per id</a></li>
                </ul>
            </li>
            <li><a href='#'>Werknemers</a>
                <ul>
                    <li><a th:href='@{/werknemers}'>Lijst</a></li>
                </ul>
            </li>
            <li><a href='#'>Offertes</a>
                <ul>
                    <li><a th:href='@{/offertes/toevoegen}'>Toevoegen</a></li>
                </ul>
            </li>
        </ul>
    </nav>
</body>
</html>

```

- (1) Je definieert een fragment met de naam head en één parameter: title.  
Dit fragment begint hier een loopt tot de bijbehorende eindtag: </head>.
- (2) Je gebruikt de parameter title van het fragment.
- (3) Je definieert een fragment met de naam menu.

Je gebruikt deze fragmenten in index.html:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Groene tenen')"></head>    ❶
<body>
    <nav th:replace='fragments::menu'></nav>                        ❷
    <h1>Groene tenen, <span th:text='${begroeting}'></span>.</h1>
    <blockquote>De aanleg van een gezellig gazon.</blockquote>
    <img alt='gazon' th:src='@{/images/gazon.jpg}' class='fullwidth'>
</body></html>

```

- (1) Je vervangt het huidige element (<head>...</head>) door het fragment met de naam head in fragments.html. Je geeft Groene tenen mee als waarde voor de parameter title.
- (2) Je vervangt het huidige element (<nav>...</nav>) door het fragment met de naam menu in fragments.html.

Je kan de website uitproberen.

## 2.6 JPA project

Je converteert het project naar een JPA project (zie JPA cursus).

Je verwijdert het vinkje bij Include libraries with this application.

## 2.7 Value objecten

Je maakt een package be.vdab.groenetenen.valueobjects. Je maakt daarin Adres:

```
package be.vdab.groenetenen.valueobjects;
// enkele imports
@Embeddable
public class Adres implements Serializable {
    private static final long serialVersionUID = 1L;
    @NotBlank
    @SafeHtml
    private String straat;
    @NotBlank
    @SafeHtml
    private String huisNr;
    @NotNull
    @Range(min = 1000, max = 9999)
    private int postcode;
    @NotBlank
    @SafeHtml
    private String gemeente;
    // Je maakt zelf getters voor straat, huisNr, postcode en gemeente
    // Je maakt zelf een default en een geparametriseerde constructor
}
```

## 2.8 Entities

Je maakt een package be.vdab.groenetenen.valueobjects. Je maakt daarin Filiaal:

```
package be.vdab.groenetenen.entities;
// enkele imports ...
@Entity
@Table(name = "filialen")
public class Filiaal implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @NotBlank
    @SafeHtml
    private String naam;
    private boolean hoofdFiliaal;
    @NumberFormat(style = Style.NUMBER)
    @NotNull
    @PositiveOrZero
    @Digits(integer = 10, fraction = 2)
    private BigDecimal waardeGebouw;
    @DateTimeFormat(style = "S-")
    @NotNull
    private LocalDate inGebruikName;
```

```

@Valid
@Embedded
private Adres adres;
@Version
private long versie;
// Je maakt getters en setters voor naam, waardeGebouw, inGebruikName, adres,
// hoofdFiliaal en versie. Voor id maak je enkel een getter, geen setter
@OneToMany(mappedBy = "filiaal")
private Set<Werknemer> werknemers;
public Set<Werknemer> getWerknemers() {
    return Collections.unmodifiableSet(werknemers);
}
}

```

Je maakt ook Werknemer:

```

package be.vdab.groenetenen.entities;
// enkele imports ...
@Entity
@Table(name = "werknemers")
public class Werknemer implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @NotBlank
    @SafeHtml
    private String voornaam;
    @NotBlank
    @SafeHtml
    private String familienaam;
    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "filiaalId")
    @NotNull
    private Filiaal filiaal;
    @NotNull
    @PositiveOrZero
    @NumberFormat(style=Style.NUMBER)
    @Digits(integer = 10, fraction = 2)
    private BigDecimal wedde;
    private long rijksregisterNr;
    // je maakt getters voor id, voornaam, familienaam, filiaal, wedde
    // en rijksregisterNr.
    // Je maakt met Eclipse hashCode en equals op basis van rijksregisterNr
    // met het menu Source, Generate hashCode() and equals()
}

```

Je maakt ValidationMessages.properties in src/main/resources:

```

javax.validation.constraints.Null.message=moet leeg zijn
javax.validation.constraints.NotNull.message=mag niet leeg zijn
javax.validation.constraints.Min.message=minstens {value}
javax.validation.constraints.DecimalMin.message=minstens {value}
javax.validation.constraints.Email.message=ongeldig e-mail adres
javax.validation.constraints.Max.message=maximaal {value}
javax.validation.constraints.DecimalMax.message=maximaal {value}
javax.validation.constraints.Size.message=tussen {min} en {max}
javax.validation.constraints.Digits.message=\
max. {integer} voor en {fraction} na komma
javax.validation.constraints.Past.message=moet in verleden
javax.validation.constraints.PastOrPresent.message=\
moet vandaag zijn of in verleden
javax.validation.constraints.Future.message=moet in toekomst

```

```
javax.validation.constraints.FutureOrPresent.message=\  
moet vandaag zijn of in toekomst  
javax.validation.constraints.Pattern.message=moet voldoen aan patroon {regexp}  
javax.validation.constraints.NotBlank.message=\  
moet meer dan enkel spaties bevatten  
javax.validation.constraints.NotEmpty.message=mag niet leeg zijn  
javax.validation.constraints.Negative.message=moet negatief zijn  
javax.validation.constraints.NegativeOrZero.message=moet negatief of nul zijn  
javax.validation.constraints.Positive.message=moet positief zijn  
javax.validation.constraints.PositiveOrZero.message=\  
moet positief zijn of nul zijn  
org.hibernate.validator.constraints.CreditCardNumber.message=\  
ongeldig kredietkaartnummer  
org.hibernate.validator.constraints.EAN.message=ongeldig artikelnummer  
org.hibernate.validator.constraints.Length.message=\  
minstens {min} / maximaal {max} tekens  
org.hibernate.validator.constraints.Range.message=\  
moet liggen tussen {min} en {max}  
org.hibernate.validator.constraints.SafeHtml.message=mag geen script bevatten  
Je maakt messages.properties in src/main/resources:  
typeMismatch.java.lang.Integer=tik een getal
```



Je commit de sources. Je publiceert op GitHub.



Proefpakket: zie takenbundel

## 3 SPRING DATA

### 3.1 Automatisch gegenereerde repository classes

Je maakt bij Spring data enkel interfaces in je repositories layer. Spring data maakt de bijbehorende implementatieclasses bij de start van je applicatie. Je schrijft dus veel minder code.

Je erft je repository interface van de interface JpaRepository.

De belangrijkste method declaraties in deze interface:

- `count()` Het aantal entities tellen.
- `delete(entity)` Een entity verwijderen.
- `exists(entity)` Controleren of een entity bestaat in de database.
- `findAll()` Alle entities lezen, zonder een bepaalde volgorde.
- `findAll(Iterable it)` De entities lezen waarvan de primary key voorkomt in de Iterable (interface die List en Set implementeren).
- `findAll(Sort sort)` Alle entities lezen, met een bepaalde volgorde.
- `findById(primaryKey)` Een entity opzoeken op zijn primary key waarde.
- `save(entity)` Een entity toevoegen (create), als die nieuw is, of een entity wijzigen (update) als die al bestaat.

Je kan, bij het erven van JpaRepository, nog method declaraties toevoegen aan je interface.

Je maakt een package `be.vdab.groenetenen.repositories` en daarin `FiliaalRepository`:

```
package be.vdab.groenetenen.repositories;
// enkele imports ...
public interface FiliaalRepository extends JpaRepository<Filiaal, Long>{
    List<Filiaal> findByIdresPostcodeBetweenOrderByAdresPostcode(
        int van, int tot);
}
```

- (1) Je geeft, bij het erven van de generic interface JpaRepository, tussen < en > het type entity mee dat bij je interface hoort, daarna het variabele type dat bij de primary key hoort.
- (2) Spring maakt automatisch een JPQL query gebaseerd op een Java method als je:
  - De method naam begint met `findBy`.
  - Daarna een eigenschap van de huidige (Filiaal) entity: `adres.postcode`. Spring gebruikt dit in het where deel van de query (**where** `adres.postcode`).
  - Daarna een operator voor het where deel: `Between`.
  - Daarna een sorteeropdracht: `OrderBy`.
  - Daarna de eigenschap waarop je wil sorteren: `adres.postcode`.
  - Spring gebruikt de method parameters als query parameters in het where deel (**where** `adres.postcode between :van and :tot`).

De gegenereerde JPQL query is dus:

```
select f from Filiaal f where f.adres.postcode between :van and :tot
order by f.adres.postcode
```

Andere voorbeelden van methods en de bijbehorende queries:

Method	where onderdeel in JPQL query
<code>findByNaamIsNull</code>	<b>where</b> naam <b>is null</b>
<code>findByNaamIsNotNull</code>	<b>where</b> naam <b>is not null</b>
<code>findByNaamLike(String naam)</code>	<b>where</b> naam <b>like</b> :naam
<code>findByNaamStartingWith(String naam)</code>	<b>where</b> naam <b>like</b> :naam Spring voegt % toe na de inhoud van naam.
<code>findByNaamEndingWith(String naam)</code>	<b>where</b> naam <b>like</b> :naam Spring voeg % toe voor de inhoud van naam.
<code>findByNaamContaining(String naam)</code>	<b>where</b> naam <b>like</b> :naam Spring voeg % toe voor en na de inhoud van naam.
<code>findByNaamIn(Iterable&lt;String&gt; namen)</code>	<b>where</b> naam <b>in</b> (naam1, naam2, ...)

Je kan sommige queries niet uitdrukken als een vertaling van een method naam.  
Je schrijft die queries dan toch als named queries in orm.xml. Fictieve voorbeelden:

```
<named-query name='Filiaal.findMetHoogsteWaardeGebouw'>
  <query>
    select f from Filiaal f where f.waardeGebouw =
      (select max(f.waardeGebouw) from Filiaal f)
  </query>
</named-query>
<named-query name='Filiaal.findGemiddeldeWaardeGebouwInGemeente'>
  <query>
    select avg(f.waardeGebouw) from Filiaal f where f.adres.gemeente = :gemeente
  </query>
</named-query>
```

Je zou dan deze methods maken in FiliaalRepository:

```
List<Filiaal> findMetHoogsteWaardeGebouw(); ❶
BigDecimal findGemiddeldeWaardeGebouwInGemeente(
  @Param("gemeente") String gemeente); ❷
```

- (1) Spring data associeert de named query Filiaal.findMetHoogsteWaardeGebouw met deze method. Spring zoekt die named query als naam van de entity (Filiaal), een punt en dan de methodnaam(findMetHoogsteWaardeGebouw).  
Als je de method oproept, roept Spring data de named query op.
- (2) Je verwijst met @Param("gemeente") naar de named query parameter gemeente.  
Je geeft een gemeente mee wanneer je de method oproept.  
Spring data gebruikt die gemeente als waarde in de named query parameter gemeente.

Je maakt een package be.vdab.groenetenen.services en daarin een interface FiliaalService:

```
package be.vdab.groenetenen.services;
// enkele imports
public interface FiliaalService {
  List<Filiaal> findByPostcode(int van, int tot);
}
```

Je implementeert deze interface in de class DefaultFiliaalService:

```
package be.vdab.groenetenen.services;
// enkele imports
@Service
@Transactional(readOnly = true, isolation = Isolation.READ_COMMITTED)
class DefaultFiliaalService implements FiliaalService {
  private final FiliaalRepository filiaalRepository;
  DefaultFiliaalService(FiliaalRepository filiaalRepository) {
    this.filiaalRepository = filiaalRepository;
  }
  @Override
  public List<Filiaal> findByPostcode(int van, int tot) {
    return filiaalRepository
      .findByAdresPostcodeBetweenOrderByAdresPostcode(van, tot);
  }
}
```

Je maakt in de package be.vdab.groenetenen.web een class die de basis wordt van een form object:

```
package be.vdab.groenetenen.web;
// enkele imports
public class VanTotPostcodeForm {
  @NotNull
  @Range(min = 1000, max = 9999)
  private Integer van;
```

```

@NotNull
@Range(min = 1000, max = 9999)
private Integer tot;
// je maakt getters en setters voor van en tot
}

```

Je maakt in de package `be.vdab.groenetenen.web` een class `FiliaalController`:

```

package be.vdab.groenetenen.web;
// enkele imports
@Controller
@RequestMapping("filialen")
class FiliaalController {
    private static final String VAN_TOT_POSTCODE_VIEW =
        "filialen/vantotpostcode";
    private final FiliaalService filiaalService;
    FiliaalController(FiliaalService filiaalService) {
        this.filiaalService = filiaalService;
    }
    @GetMapping("vantotpostcode")
    ModelAndView vanTotPostcode() {
        return new ModelAndView(VAN_TOT_POSTCODE_VIEW)
            .addObject(new VanTotPostcodeForm());
    }
    @GetMapping(params = { "van", "tot" })
    ModelAndView vanTotPostcode(@Valid VanTotPostcodeForm form,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return new ModelAndView(VAN_TOT_POSTCODE_VIEW);
        }
        return new ModelAndView(VAN_TOT_POSTCODE_VIEW)
            .addObject("filialen", filiaalService.findByPostcode(
                form.getVan(), form.getTot()));
    }
}

```

- (1) Als je in een grote applicatie alle HTML bestanden in de map `templates` plaatst, bevat deze map honderden bestanden. Deze map verliest dan zijn overzichtelijkheid. Je verhindert dit door in deze map submappen te maken en de HTML bestanden in deze submappen te plaatsen. Je zal bij ons alle HTML bestanden die met filialen te maken hebben plaatsen in de submap `filialen`.

Je maakt in de map `templates` een map `filialen`. Je maakt daarin `vantotpostcode.html`:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Van tot postcode')"></head>
<body>
    <nav th:replace='fragments::menu'></nav>
    <h1>Van tot postcode</h1>
    <form th:action="@{/filialen}" th:object='${vanTotPostcodeForm}'
        method='get'>
        <label>Van:
            <span th:if="${#fields.hasErrors('van')}" th:errors='*{van}'></span>
            <input th:field='*{van}' type='number' required min='1000' max='9999'
                autofocus>
        </label>
        <label>Tot:
            <span th:if="${#fields.hasErrors('tot')}" th:errors='*{tot}'></span>
            <input th:field='*{tot}' type='number' required min='1000' max='9999'>
        </label>
        <input type='submit' value='Zoeken'>
    </form>

```



```

<div th:each='filiaal:${filialen}'>
    <h2 th:text='${filiaal.naam}'></h2>
    <div th:text='|${filiaal.adres.straat} ${filiaal.adres.huisNr}|'></div>
    <div th:text='|${filiaal.adres.postcode} ${filiaal.adres.gemeente}|'></div>
</div>
</body>
</html>

```

- (1) Je vermeldt bij th:action de URL /filialen. Dit is de URL waarnaar de form zijn data zal submitten. Je vermeldt bij th:object de naam van het form object.
- (2) Als er validatiefouten zijn bij het invullen van het attribuut van in het form object, toon je de foutboodschap die hoort bij dit attribuut.
- (3) Je associeert met \*{van} het attribuut van van het form object met dit invoervak.
- (4) Je itereert over de verzameling in \${filialen}.  
Je krijgt bij iedere iteratie het Filiaal object aangeboden in de variabele filiaal.  
Thymeleaf maakt per iteratie het huidig element (<div>...</div>) en al zijn child elementen.

Je kan de website uitproberen.

### 3.1.1 Automatische String naar Entity converters

Je krijgt soms van de browser een filiaal id binnen die automatisch moet geconverteerd worden naar het bijbehorende Filiaal object (door dit Filiaal object in de database op te zoeken).

Spring data maakt zo'n converter zodra je een interface FiliaalRepository erft van JpaRepository. Jij moet voor deze functionaliteit geen code toevoegen aan FiliaalRepository of aan FiliaalService.

Je maakt een voorbeeld die deze converter gebruikt:

Je voegt een regel toe voor de laatste </div> in vantotpostcode.html:

```

<div><a th:href="@{/filialen/{id}(id=${filiaal.id})}">Detail</a></div>

```

- (1) Je maakt een URL template /filialen/{id}. Je maakt een URL, gebaseerd op deze URL template, waarbij de path variabele id vervangen wordt door de id van het huidig filiaal.

Je voegt een method toe aan FiliaalController. Als de gebruiker de hyperlink die je maakte bij (1) aanklikt, wordt de bijbehorende request door deze method verwerkt:

```

private static final String FILIAAL_VIEW = "filialen/filiaal";
private static final String REDIRECT_FILIAAL_NIET_GEVONDEN = "redirect:/";
@GetMapping("/{filiaal}")
ModelAndView read(@PathVariable Optional<Filiaal> filiaal,
    RedirectAttributes redirectAttributes) {
    if (filiaal.isPresent()) {
        return new ModelAndView(FILIAAL_VIEW).addObject(filiaal.get());
    }
    redirectAttributes.addAttribute("fout", "Filiaal niet gevonden");
    return new ModelAndView(REDIRECT_FILIAAL_NIET_GEVONDEN);
}

```

- (1) Je geeft aan dat de method read requests verwerkt met een URL vermeld bij @RequestMapping, gevolgd door een path variabele die je hier aanduidt met de naam filiaal.
- (2) Je vraagt Spring de parameter filiaal in te vullen met de inhoud van de path variabele filiaal. Spring gebruikt op dit moment de converter van Spring data die de path variabele omzet in een Optional<Filiaal> object.

Je maakt in templates/filialen filiaal.html:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace='fragments::head(title=${filiaal.naam})'></head>
<body>
    <nav th:replace='fragments::menu'></nav>
    <h1 th:text='${filiaal.naam}'></h1>

```

```

<dl>
  <dt>Straat</dt><dd th:text='${filiaal.adres.straat}'></dd>
  <dt>Huisnr.</dt><dd th:text='${filiaal.adres.huisNr}'></dd>
  <dt>Postcode</dt><dd th:text='${filiaal.adres.postcode}'></dd>
  <dt>Gemeente</dt><dd th:text='${filiaal.adres.gemeente}'></dd>
  <dt>Type</dt>
  <dd th:text="${filiaal.hoofdFiliaal ? 'Hoofdfiliaal' : 'Bijfiliaal'}"> ❶
</dd>
  <dt>Waarde gebouw</dt><dd th:text='${filiaal.waardeGebouw}'></dd> ❷
  <dt>In gebruikname</dt><dd th:text='${filiaal.inGebruikName}'></dd> ❸
</dl>
</body>
</html>

```

- (1) Als het attribuut hoofdFiliaal de waarde true bevat, stuur je de tekst Hoofdfiliaal naar de browser. Anders stuur je de tekst Bijfiliaal naar de browser.
- (2) Het attribuut waardeGebouw is in de class Filiaal voorzien van @NumberFormat. Je gebruikt dubbele accolades om de bijbehorende opmaak hier toe te passen.
- (3) Het attribuut inGebruikName is in de class Filiaal voorzien van @DateTimeFormat. Je gebruikt dubbele accolades om de bijbehorende opmaak hier toe te passen.

Je voegt in index.html volgende regel toe na <body>:

```
<div th:if='${param.fout != null}' th:text='${param.fout}' class='fout'></div>
```

Je kan de website uitproberen.

### 3.2 Spring data en Entity graphs

Je maakt eerst een pagina met alle werknemers. Je toont per werknemer ook zijn filiaal.

Je maakt de pagina daarna performanter door het toepassen van een entity graph.

Je maakt in be.vdab.groenetenen.repositories een interface WerknemerRepository:

```

package be.vdab.groenetenen.repositories;
// enkele imports ...
public interface WerknemerRepository extends JpaRepository<Werknemer, Long> {}

```

Je maakt in be.vdab.groenetenen.services een interface WerknemerService:

```

package be.vdab.groenetenen.services;
// enkele imports
public interface WerknemerService {
    List<Werknemer> findAll();
}

```

Je implementeert deze interface:

```

package be.vdab.groenetenen.services;
// enkele imports ...
@Service
@Transactional(readOnly=true, isolation=Isolation.READ_COMMITTED)
class DefaultWerknemerService implements WerknemerService {
    private final WerknemerRepository werknemerRepository;
    DefaultWerknemerService(WerknemerRepository werknemerRepository) {
        this.werknemerRepository = werknemerRepository;
    }
    @Override
    public List<Werknemer> findAll() {
        return werknemerRepository.findAll(Sort.by("familienaam", "voornaam")); ❶
    }
}

```

- (1) De by method aanvaardt een variabel aantal eigenschappen waarop Spring sorteert.

Je maakt WerknemerController:

```
package be.vdab.groenetenen.web;
// enkele imports
@Controller
@RequestMapping("werknemers")
class WerknemerController {
    private static final String WERKNEMERS_VIEW = "werknemers/werknemers";
    private final WerknemerService werknemerService;
    WerknemerController(WerknemerService werknemerService) {
        this.werknemerService = werknemerService;
    }
    @GetMapping
    ModelAndView lijst() {
        return new ModelAndView(WERKNEMERS_VIEW,
            "werknemers", werknemerService.findAll());
    }
}
```

Je maakt in de map templates een map werknemers. Je maakt daarin werknemers.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Werknemers')"></head>
<body>
    <nav th:replace='fragments::menu'></nav>
    <h1>Werknemers</h1>
    <table>
        <thead>
            <tr><th>Voornaam</th><th>Familienaam</th><th>Filiaal</th></tr>
        </thead>
        <tbody>
            <tr th:each='werknemer:${werknemers}'>
                <td th:text='${werknemer.voornaam}'></td>
                <td th:text='${werknemer.familienaam}'></td>
                <td th:text='${werknemer.filiaal.naam}'></td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

Je probeert de website uit.

De pagina werkt, maar is niet performant. Als je na het afbeelden van de pagina kijkt in het Eclipse venster Console, zie je het N+1 probleem (zie cursus “JPA met Hibernate”):

JPA stuurt één select statement naar de database om de werknemers te lezen, en stuurt daarna meerdere select statements die elk één filiaal lezen.

Je zal dit oplossen met een entity graph (zie cursus “JPA met Hibernate”), zodat JPA met één select statement de werknemers én de bijbehorende filialen leest.

Je maakt een constante in de class Werknemer:

```
public static final String MET_FILIAAL="Werknemer.metFiliaal";
```

Je voegt juist voor de class Werknemer de definitie van een named entity graph toe:

```
@NamedEntityGraph(name = Werknemer.MET_FILIAAL,
    attributeNodes = @NamedAttributeNode("filiaal") )
```

Je kan nu met @EntityGraph aangeven dat Spring data, bij het uitvoeren van een query, rekening moet houden met deze named entity graph.

Als je interface `WerknemerRepository` volgende method declaratie zou bevatten

```
Werknemer findByRijksregisterNr(long rijksRegisterNr)
```

schrijf je voor die method declaratie `@EntityGraph(Werknemer.MET_FILIAAL)`

Spring Data houdt dan rekening met de named entity graph `Werknemer.metFiliaal`

bij het uitvoeren van de query die hoort bij de method `findByRijksregisterNr`.

In ons voorbeeld moet Spring Data rekening houden met de named entity graph

bij het uitvoeren van de method `findAll(Sort sort)`. Je erft die method van `JpaRepository`.

Je voegt daartoe code toe aan de `WerknemerRepository`:

```
@Override
@EntityGraph(Werknemer.MET_FILIAAL)
List<Werknemer> findAll(Sort sort);
```

❶  
❷  
❸

(1) Je overschrijft een method uit je base interface (`JpaRepository`).

(2) Dit overschrijven is het toepassen van de named entity graph `Werknemer.metFiliaal`

(3) Dit is de method declaratie zoals beschreven in de base interface.

Je kan dit uitproberen.

Je ziet in het Eclipse venster Console dat je het N+1 probleem opgelost hebt.

### 3.3 Pagineren

Als de gebruiker een zoekopdracht uitvoert die veel records teruggeeft,

is het de gewoonte die records niet in één keer te tonen, maar per pagina.

Spring data helpt om dit vlot uit te werken.

De interface `JpaRepository` bevat de method `Page<T> findAll(Pageable pageable)`

die één pagina records teruggeeft. Het generische type `T` wordt in je eigen repository interface een entity type (bvb. `Filiaal`, `Werknemer`)

Je geeft in de parameter `Pageable` informatie mee over de gewenste pagina.

Je moet zelf geen `Pageable` object maken: je kan een `Pageable` parameter toevoegen

aan een `@GetMapping` method van een Controller. Spring vult de eigenschappen

van deze `Pageable` parameter aan de hand van volgende request parameters:

- `page` de hoeveelste pagina je wenst (telt vanaf 0)
- `size` het aantal records op één pagina
- `sort` de entity eigenschap waarop Spring Data de records moet sorteren

Een request met de request parameters: `?page=1&size=10&sort=naam`

vraagt dus de 2° pagina, 10 records per pagina en te sorteren op de entity eigenschap naam.

Je krijgt de gevraagde pagina terug als de returnwaarde van `findAll`: een `Page` object.

Zo'n object heeft interessante getters. Je ziet in de laatste kolom hoe je die informatie

ook kan opvragen in Thymeleaf, als `page` het `Page` object bevat

- |                              |  |                                     |
|------------------------------|--|-------------------------------------|
| • <code>getTotalPages</code> | totaal aantal pagina's                 | <code>\${page.totalPages}</code>    |
| • <code>getNumber</code>     | volgnummer van de huidige pagina       | <code>\${page.number}</code>        |
| • <code>getContent</code>    | List met entities in de huidige pagina | <code>\${page.content}</code>       |
| • <code>isFirst</code>       | is dit de eerste pagina (true/false)   | <code>\${page.first}</code>         |
| • <code>isLast</code>        | is dit de laatste pagina (true/false)  | <code>\${page.last}</code>          |
| • <code>hasNext</code>       | is er een volgende pagina (true/false) | <code>\${page.hasNext()}</code>     |
| • <code>hasPrevious</code>   | is er een vorige pagina (true/false)   | <code>\${page.hasPrevious()}</code> |

Je past dit nu toe in de lijst met werknemers.

Je maakt een method declaratie in `WerknemerRepository`, zodat Spring data bij het uitvoeren van deze versie van de method `findAll` ook rekening houdt met de named entity graph:

```
@Override
@EntityGraph(Werknemer.MET_FILIAAL)
Page<Werknemer> findAll(Pageable pageable);
```

Je wijzigt in WerknemerService de declaratie van de method findAll:

```
Page<Werknemer> findAll(Pageable pageable);
```

Je wijzigt in DefaultWerknemerService de method findAll:

```
@Override
public Page<Werknemer> findAll(Pageable pageable) {
    return werknemerRepository.findAll(pageable);
}
```

Je wijzigt in WerknemerController de method lijst:

```
@GetMapping
ModelAndView lijst(Pageable pageable) {
    return new ModelAndView(WERKNEMERS_VIEW,
        "page", werknemerService.findAll(pageable));
}
```

Je wijzigt in werknemers.jsp \${werknemers} naar \${page.content}

Als je de werknemer lijst vraagt, zie je een eerste pagina met werknemers. Je hebt bij die request geen request parameters meegegeven. Spring Data initialiseert het Pageable object dan zodat de eerste pagina opgevraagd wordt, met 20 records per pagina en geen sortering.

Als je in de adresbalk ?size=10 toevoegt, zie je een pagina met 10 records.

Als je in de adresbalk &page=1 toevoegt, zie je de tweede pagina.

Als je in de adresbalk &sort=filiaal.naam toevoegt, zijn de records gesorteerd op filiaal.

Je breidt met deze kennis werknemers.html uit.

De gebruiker kan sorteren. Je toont daartoe de woorden Voornaam, Familienaam en Filiaal in de tabelhoofding als hyperlink. Als de gebruiker zo'n hyperlink aanklikt, sorteert hij op de bijbehorende eigenschap. Je wijzigt de regels met <th>...</th>:

```
<th><a th:href="@{/werknemers?sort=voornaam}">Voornaam</a></th>
<th><a th:href="@{/werknemers?sort=familienaam}">Familienaam</a></th>
<th><a th:href="@{/werknemers?sort=filiaal.naam}">Filiaal</a></th>
```

Je kan de website uitproberen.

Je breidt de pagina verder uit. De gebruiker kan een gewenste pagina aanduiden met hyperlinks onder de tabel. Je toont het nummer van de huidige pagina (hier pagina 1) als platte tekst:



Als mens is het volgnummer van de eerste pagina 1. Voor Spring data is dit volgnummer 0.

Je voegt regels na </table>:

```
<p class='pagineren'>
    <span th:each='pageNr:${#numbers.sequence(1,page.totalPages)}'>
        <span th:if='${pageNr -1 == page.number}' th:text='${pageNr}'></span> ❶
        <a th:if='${pageNr -1 != page.number}' th:text='${pageNr}' ❷
            th:href="@{/werknemers(page=${pageNr-1}, sort=${param.sort})}"></a> ❸
    </span> ❹
</p>
```

- (1) De expressie `${#numbers.sequence(1,page.totalPages)}` genereert een reeks getallen van 1 tot en met het aantal pagina's. Je itereert over deze reeks met een variabele `pageNr`.
- (2) Als `pageNr` gelijk is aan het volgnummer van de huidig getoonde pagina, toon je `pageNr` als platte tekst.
- (3) Als `pageNr` verschilt van het volgnummer van de huidig getoonde pagina, toon je `pageNr` als hyperlink.
- (4) `${param.sort}` verwijst naar de request parameter `sort` in de huidige URL.

Je kan de pagina uitproberen.

JPA houdt bij het pagineren de performantie hoog door nooit *alle* werknemers uit de database te lezen. Als je de eerste pagina vraagt, stuurt JPA volgend SQL statement naar de database:

```
select ... from werknemers ... limit 20
```

Het onderdeel `limit 20` leest enkel de eerste 20 records uit de database.

Als je de derde pagina vraagt, stuurt JPA volgend SQL statement naar de database:

```
select ... from werknemers ... limit 39, 20
```

Het onderdeel `limit 39, 20` leest enkel de 20 records vanaf het 40<sup>e</sup> record uit de database (de nummering van records begint vanaf 0).



Je commit de sources. Je publiceert op GitHub.



Opmerking: methods in je repository interface kunnen ook pagineren. Het volstaat dat de method een Page object teruggeeft en een Pageable parameter bevat. Voorbeeld:  
`Page<Werknemer> findByNaam(String naam, Pageable pageable);`



Opmerking: Spring data ondersteunt ook het aanspreken van niet-relatieve databases, zoals MongoDB, Redis, Cassandra, Neo4J, ...



Brouwers: zie takenbundel

## 4 TESTEN MET EEN IN-MEMORY DATABASE

Een in-memory database houdt zijn data bij in het interne geheugen.

- Dit is niet praktisch voor een productiedatabase:  
bij elektriciteitspanne verliest een in-memory database zijn data.
- Een in-memory database is wel handig bij testen:  
de testen lopen snel (data in RAM) en na de testen mag de test data verloren gaan.

Er bestaan meerdere merken in-memory databases: H2, HSQLDB, Derby, ....

Je gebruikt de populairste: H2.

Je maakt in `src/test/java` een package `be.vdab.groenetenen.repositories`.

Je maakt daarin een class `FiliaalRepositoryTest`:

```
// enkele imports
@RunWith(SpringRunner.class)
@DataJpaTest
public class FiliaalRepositoryTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    private static final String FILIALEN = "filialen";
    @Autowired
    private FiliaalRepository repository;
    @Test
    public void create() {
        Adres adres = new Adres("straat", "huisNr", 1000, "gemeente");
        Filiaal filiaal = new Filiaal();
        filiaal.setNaam("test");
        filiaal.setAdres(adres);
        filiaal.setWaardeGebouw(BigDecimal.ZERO);
        filiaal.setInGebruikName(LocalDate.now());
        int aantalFilialen = super.countRowsInTable(FILIALEN);
        repository.save(filiaal);
        assertEquals(aantalFilialen + 1, super.countRowsInTable(FILIALEN));
        assertEquals(0, filiaal.getId());
        assertEquals(1, super.countRowsInTableWhere(FILIALEN,
            "id=" + filiaal.getId()));
    }
}
```

(1) `@DataJpaTest` doet heel wat werk:

- Hij maakt met H2 een lege, in-memory database.
- Hij maakt lege tabellen in die database. Hij baseert de tabel namen, kolom namen, kolom types, primary keys en foreign keys op de entity classes en value object classes en de JPA annotations in die classes.
- Hij maakt een `DataSource` bean die naar deze database verwijst.  
Zo worden alle database handelingen in de test uitgevoerd op de in-memory database.
- Hij maakt een `EntityManager` bean.
- Hij maakt beans van de classes, gegenereerd door Spring Data, die gebaseerd zijn op de interfaces uit je repository layer.

Je kan deze test uitvoeren.

Je ziet in het venster Console ook de SQL opdrachten waarmee de tables worden aangemaakt.

#### 4.1 Unique constraint

In de MySQL database groenetenen bevat de table werknemers een unieke index op de kolom rijksregisternr. Deze controleert dat de waarden in die kolom uniek zijn.

Als je wil dat Spring, bij het aanmaken van de table werknemers in de H2 database, ook uniciteit controleert op de kolom rijksregisternr, tik je in de class Werknemer voor de private variabele rijksregisterNr

```
@Column(unique = true)
```

Als je de test opnieuw uitvoert, zie je in het venster Console nu ook een SQL opdracht die de kolom rijksregisternr instelt als een kolom met unieke waarden:

```
alter table werknemers  
  add constraint UK_6d6qhy7b2q6vcuwyihroqchid unique (rijksregisterNr)
```



Je commit de sources. Je publiceert op GitHub.



## 5 VALIDATION ANNOTATIONS MAKEN

Je hebt in je kleine website al 3 keer (met `@Range(min = 1000, max = 9999)`) gedefinieerd dat een postcode moet liggen tussen 1000 en 9999. Dit is niet interessant: als de voorwaarden voor een correcte postcode in de werkelijkheid wijzigen, moet je op 3 plaatsen aanpassingen doen.

Je leert dit probleem hier oplossen.

Je maakt zelf een validation annotation `@Postcode`.

Die controleert intern of een waarde tussen 1000 en 9999 ligt.

Je vervangt op drie plaatsen `@Range(min = 1000, max = 9999)` door `@Postcode`.

Als de voorwaarden voor een correcte postcode in de werkelijkheid wijzigen, moet je enkel één keer de interne werking van `@Postcode` wijzigen.

Je kan een validation annotation op twee manieren maken:

- Als een samenstelling van andere validation annotations.
- De validatie in detail uitprogrammeren

### 5.1 Samenstelling van andere validation annotations

Je maakt `@Postcode` als een samenstelling van één bestaande bean validation annotation: `@Range(min = 1000, max = 9999)`

Je maakt een package `be.vdab.groenetenenen.constraints` en daarin de annotation `@Postcode`:

1. Je maakt de package `be.vdab.groenetenenen.constraints`.
2. Je klikt met de rechtermuisknop op de package `be.vdab.groenetenenen.constraints`.
3. Je kiest New, Other.
4. Je kiest binnen de categorie Java voor Annotation en je kiest Next.
5. Je tikt Postcode bij Name
6. Je plaatst een vinkje bij Add `@Retention` en je kiest Runtime (uitleg hier onder).
7. Je plaatst een vinkje bij Add `@Target` en je plaatst vinkjes bij Field, Method, Annotation type (uitleg hier onder).
8. Je kiest Finish.

Je wijzigt de source:

```
package be.vdab.groenetenenen.constraints;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
// enkele andere imports
@Target({METHOD, FIELD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Range(min = 1000, max = 9999)
public @interface Postcode {
    @OverrideAttribute(constraint = Range.class, name = "message")
    String message() default "{be.vdab.groenetenenen.constraints.Postcode.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

- (1) Je definieert de source onderdelen waarvoor je de annotation kan tikken.
  - a. METHOD voor een method  
(Je kan de validation annotation schrijven voor een getter)
  - b. FIELD voor de declaratie van een instance variabele  
(Je kan de validation annotation schrijven voor voor een instance variabele)
  - c. ANNOTATION\_TYPE voor de definitie van een andere annotation  
(Je kan de annotation gebruiken als basis van nog een andere eigen annotation.

- (2) Je definieert hoe lang Java de annotation behoudt. `RUNTIME` betekent dat de annotation bij het uitvoeren van het programma nog ter beschikking is. Als je `SOURCE` zou kiezen in plaats van `RUNTIME`, neemt de compiler de annotation niet op in de bytecode.
- (3) Je tikt bij een annotation `@Constraint` als die annotation een *validation* annotation is. De parameter `validatedBy` is verplicht te vermelden. Je geeft een lege array mee als je de eerste manier gebruikt om een validation annotation te maken.
- (4) Je vermeldt de bestaande bean validation annotation waarop je de eigen bean validation baseert. Je mag ook meer dan één bestaande bean validation annotation vermelden.
- (5) Je definieert een annotation met het keyword `@interface`.
- (6) Je geeft aan dat je de message parameter van `@Range` wil overschrijven met een eigen waarde. Je doet dit op de volgende regel.
- (7) Je definieert een annotation parameter. Je doet dit met de syntax van een method declaratie. `String message()` betekent dat je aan `@Postcode` een parameter `message` kan meegeven. Die parameter bevat de key van de boodschap die bij de validation annotation hoort. `message` is een optionele parameter. Als je hem niet meegeeft, krijgt hij een default waarde. Je geeft deze waarde mee met het keyword `default`.
- (8) Elke validation annotation moet ook een optionele parameter `groups` hebben. Je ziet verder in deze cursus wat validation groups zijn.
- (9) Elke validation annotation moet ook een optionele parameter `payload` hebben. Het gebruik van deze parameter valt buiten deze cursus.

Je voegt de fouttekst die hoort bij `@Postcode` toe aan `ValidationMessages.properties`:  
`be.vdab.groenetenenen.constraints.Postcode.message=een postcode ligt tussen 1000 en 9999`

Je vervangt in `Adres` en `VanTotPostcodeForm` `@Range(...)` door `@Postcode`.

Je kan de website uitproberen.

## 5.2 Eigen bean validation in detail

Sommige bean validations bevatten complexe validaties. Het is soms onmogelijk deze uit te drukken als een samenstelling van andere validation annotations.

Je leert hier hoe je zo'n bean validation schrijft, waarbij je de validatie zelf in Java code schrijft.

Je wijzigt `Postcode`:

```
package be.vdab.groenetenenen.constraints;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target({FIELD, METHOD, ANNOTATION_TYPE })
@Retention(RUNTIME)
@Constraint(validatedBy = PostcodeValidator.class)
public @interface Postcode {
    String message() default "{be.vdab.groenetenenen.constraints.Postcode.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

- (1) Je definieert in de huidige source enkel de annotation `@Postcode`. Je definieert in een aparte class (`PostcodeValidator`) de validatie code (ligt de waarde tussen 1000 en 9999 ?). Je koppelt deze class aan de annotation via `@Constraint`.

Je maakt in dezelfde package de class PostcodeValidator:

```
package be.vdab.groenetenen.constraints;
// enkele imports uit javax.validation
public class PostcodeValidator
    implements ConstraintValidator<Postcode, Integer> {
    private static final int MIN_POSTCODE = 1000;
    private static final int MAX_POSTCODE = 9999;
    @Override
    public void initialize(Postcode postcode) {
    }
    @Override
    public boolean isValid(Integer postcode,
        ConstraintValidatorContext context) {
        return postcode == null
            || (postcode >= MIN_POSTCODE && postcode <= MAX_POSTCODE);
    }
}
```

- (1) Een class, met de validatiecode van een bean validation annotation, implementeert ConstraintValidator. Je geeft tussen < en > eerst de bean validation annotation mee. Je definieert daarna het type variabele waarbij de annotation mag voorkomen. @Postcode mag voorkomen bij een Integer (of int) variabele.
- (2) Het gebruik van de method initialize valt buiten deze cursus.
- (3) Bean validation roept de method isValid op bij het valideren van een variabele voorzien van @Postcode. Je geeft true terug als de variabele een correcte waarde bevat.
- (4) Zoals de ingebakken annotations (@Min, @Max, ...) produceer je geen foutmelding als de te valideren waarde gelijk is aan null: de validatie hiervan gebeurt door @NotNull.

Je kan de website terug uitproberen.

### 5.3 Object properties ten opzichte van mekaar valideren

Je valideert tot nu elke object property apart. Je leert nu hoe je object properties ten opzichte van mekaar valideert. Je valideert als voorbeeld dat in de class VanTotPostcodeForm de property tot groter of gelijk moet zijn aan de property van.

Je maakt hiervoor een bean validation annotation

@VanTotPostcodeFormVanKleinerDanOfGelijkAanTot:

1. Je klikt met de rechtermuisknop op de package be.vdab.groenetenen.constraints.
2. Je kiest New, Other.
3. Je kiest binnen de categorie Java voor Annotation en je kiest Next.
4. Je tikt VanTotPostcodeFormVanKleinerDanOfGelijkAanTot bij Name
5. Je plaatst een vinkje bij Add @Retention en je kiest Runtime (uitleg hier onder).
6. Je plaatst een vinkje bij Add @Target en je plaatst vinkjes bij Type en Annotation type.
7. Je kiest Finish.

Je wijzigt de source:

```
package be.vdab.groenetenen.constraints;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
// enkele andere imports
@Retention(RUNTIME)
@Target({TYPE, ANNOTATION_TYPE})
@Constraint(
    validatedBy = VanTotPostcodeFormVanKleinerDanOfGelijkAanTotValidator.class)
public @interface VanTotPostcodeFormVanKleinerDanOfGelijkAanTot {
    String message() default
        "{be.vdab.groenetenen.constraints.VanKleinerDanOfGelijkAanTot.message}";
    Class<?>[] groups() default {};
```

```
Class<? extends Payload>[] payload() default {};
```

- (1) Je definieert de source onderdelen waarvoor je de annotation kan tikken.
  - a. TYPE voor een class
  - b. ANNOTATION\_TYPE voor de definitie van een andere annotation  
(Je kan de annotation gebruiken als basis van nog een andere eigen annotation).

Je maakt in dezelfde package de class

VanTotPostcodeFormVanKleinerDanOfGelijkAanTotValidator:

```
package be.vdab.groenetenen.constraints;
// enkele imports
public class VanTotPostcodeFormVanKleinerDanOfGelijkAanTotValidator
    implements ConstraintValidator<
        VanTotPostcodeFormVanKleinerDanOfGelijkAanTot, VanTotPostcodeForm> {    ❶
    @Override
    public void initialize(VanTotPostcodeFormVanKleinerDanOfGelijkAanTot arg0) {
    }
    @Override
    public boolean isValid(VanTotPostcodeForm form,                                ❷
        ConstraintValidatorContext context) {
        if (form.getVan() == null || form.getTot() == null) {                    ❸
            return true;
        }
        return form.getVan() <= form.getTot();                                  ❹
    }
}
```

- (1) @VanTotPostcodeFormVanKleinerDanOfGelijkAanTot mag voorkomen bij de class VanTotPostcodeForm.
- (2) Bean validation roept de method isValid op bij het valideren van een VanTotPostcodeForm object. Je geeft true terug bij een correcte validatie.
- (3) Je produceert geen foutmelding als de te valideren waarden gelijk zijn aan null, zoals de ingebakken annotations (@Min, @Max, ...) dat ook doen.
- (4) De validatie is correct als van kleiner is of gelijk aan tot.

Je tikt @VanTotPostcodeFormVanKleinerDanOfGelijkAanTot voor de class VanTotPostcodeForm.

Je voegt de fouttekst die hoort bij @PostcodeReeksVanKleinerDanOfGelijkAanTot toe aan ValidationMessages.properties:

```
be.vdab.groenetenen.constraints.VanKleinerDanOfGelijkAanTot.message=\
van moet kleiner of gelijk zijn aan tot
```

Je voegt volgende regels toe aan vantotpostcode.html, voor </form>:

```
<div th:if='${#fields.hasGlobalErrors()}'>    ❶
    <div class='fout' th:each='err : ${#fields.globalErrors()}'    ❷
        th:text='${err}'></div>
    </div>
```

- (1) De expressie \${#fields.hasGlobalErrors()} geeft true terug als het command object fouten bevat die niet aan één veld verbonden zijn.
- (2) Je itereert met een variabele err over deze fouten.

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



Ondernemingsnummer: zie takenbundel

## 6 SESSION SCOPE, VALIDATION GROUPS

### 6.1 Session scope

Spring onthoudt een form object standaard niet in een session variabele. Zo blijft je website performant bij veel gelijktijdige gebruikers.

Je hebt het form object soms over meerdere requests nodig.

Spring kan dan het form object als een session variabele onthouden.

Het form object behoudt zo zijn inhoud over de requests heen.

Je werkt de use case 'Offerte toevoegen' uit.

- Je vraagt de voornaam, familienaam en email adres in een 1° pagina.
- Je vraagt de oppervlakte (van het gazon) in een 2° pagina.

### 6.2 Validation groups

Je valideert tot nu alle properties in één keer. Dit kan in dit voorbeeld niet.

Op de eerste pagina heeft de gebruiker enkel de voornaam, familienaam en email adres ingetikt.

Je kan bij het submitten van die eerste pagina enkel die eigenschappen valideren.

Je verdeelt daartoe de validation annotations in validation groups:

- een group Stap1 met de annotations van voornaam, familienaam en emailAdres.
- een group Stap2 met de annotations van oppervlakte.

Je maakt in `be.vdab.groenetenen.entities` een class `Offerte`:

```
package be.vdab.groenetenen.entities;
// enkele imports ...
@Entity
@Table(name="offertes")
public class Offerte implements Serializable {
    public interface Stap1 {}
    public interface Stap2 {}
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    @NotBlank(groups = Stap1.class)
    @SafeHtml(groups = Stap1.class)
    private String voornaam;
    @NotBlank(groups = Stap1.class)
    @SafeHtml(groups = Stap1.class)
    private String familienaam;
    @NotNull(groups = Stap1.class)
    @Email(groups = Stap1.class)
    private String emailAdres;
    @NotNull(groups = Stap2.class)
    @Positive(groups = Stap2.class)
    @NumberFormat
    private Integer oppervlakte;
    @DateTimeFormat(style = "S-")
    private LocalDate aangemaakt = LocalDate.now();
    // Je maakt getters voor id, voornaam, familienaam, emailAdres, oppervlakte en
    // aangemaakt.
    // Je maakt setters voor voornaam, familienaam, emailAdres, oppervlakte.
}
```

- (1) Je definieert een validation group als een lege interface. De naam van de interface wordt de naam van de validation group.
- (2) Je bepaalt met groups tot welke validation group de annotation hoort.

Je maakt OfferteRepository:

```
package be.vdab.groenetenen.repositories;
// enkele imports ...
public interface OfferteRepository extends JpaRepository<Offerte, Long>{
}
```

Je maakt OfferteService:

```
package be.vdab.groenetenen.services;
// enkele imports
public interface OfferteService {
    void create(Offerte offerte);
    Optional<Offerte> read(long id);
}
```

Je maakt DefaultOfferteService:

```
package be.vdab.groenetenen.services;
// enkele imports
@Service
@Transactional(readOnly = true, isolation = Isolation.READ_COMMITTED)
class DefaultOfferteService implements OfferteService {
    private final OfferteRepository offerteRepository;
    DefaultOfferteService(OfferteRepository offerteRepository) {
        this.offerteRepository = offerteRepository;
    }
    @Override
    @Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
    public void create(Offerte offerte) {
        offerteRepository.save(offerte);
    }
    @Override
    public Optional<Offerte> read(long id) {
        return offerteRepository.findById(id);
    }
}
```

Je maakt OfferteController. Als de gebruiker een request doet naar /offertes/toevoegen, toont de method stap1 de pagina met de 1° stap:

```
package be.vdab.groenetenen.web;
// enkele imports
@Controller
@RequestMapping("offertes")
@SessionAttributes("offerte")
class OfferteController {
    private final OfferteService offerteService;
    public OfferteController(OfferteService offerteService) {
        this.offerteService = offerteService;
    }
    private static final String STAP_1_VIEW="offertes/stap1";
    @GetMapping("toevoegen")
    ModelAndView stap1() {
        return new ModelAndView(STAP_1_VIEW).addObject(new Offerte());
    }
}
```

- (1) Je tikt @SessionAttributes bij de controller class. Je geeft de naam van het command object mee dat Spring als HttpSession variabele moet bijhouden.

Je maakt in de map templates een map offertes. Je maakt daarin stap1.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Offerte toevoegen: stap 1')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>Offerte toevoegen: stap 1</h1>
<form th:action="@{/offertes/toevoegen}" th:object='${offerte}' method='post'>
<label>Voornaam:
<span th:if="${#fields.hasErrors('voornaam')}}" th:errors='*{voornaam}'></span>
<input th:field='*{voornaam}' required autofocus>
</label>
<label>Familienaam:
<span th:if="${#fields.hasErrors('familienaam')}}"
  th:errors='*{familienaam}'></span>
<input th:field='*{familienaam}' required >
</label>
<label>Email adres:
<span th:if="${#fields.hasErrors('emailAdres')}}"
  th:errors='*{emailAdres}'></span>
<input th:field='*{emailAdres}' type='email' required >
</label>
<input type='submit' value='Stap 2' name='stap2'>
</form>
</body>
</html>
```

- (1) Je geeft de submit knop ook een name. Wanneer de form gesubmit wordt, zal de browser naast de parameters voornaam, familienaam en emailAdres ook een parameter stap2 naar de webserver sturen. Deze parameter bevat de tekst van de knop: Stap 2.

Je maakt een method in OfferteController.

Deze verwerkt de request als de form van stap 1 gesubmit wordt:

```
private static final String STAP_2_VIEW = "offertes/stap2";
@PostMapping(value = "toevoegen", params = "stap2")
String stap1NaarStap2(@Validated(Offerte.Stap1.class) Offerte offerte,
  BindingResult bindingResult) {
  return bindingResult.hasErrors() ? STAP_1_VIEW : STAP_2_VIEW;
}
```

- (1) De method stap1NaarStap2 verwerkt een POST request naar /offertes/toevoegen, op voorwaarde dat die een request parameter stap2 bevat. Die request is een submit van stap 1.
- (2) Je valideert de Offerte attributen die de gebruiker bij de eerste stap intikte met @Validated. Je geeft als parameter de interface mee, waarmee je de validaties van de eerste stap groepeerde in een validation group.
- (3) Als stap 1 validatiefouten bevatte, toon je terug stap 1. Anders toon je stap 2.

Je maakt stap2.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Offerte toevoegen: stap 2')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>Offerte toevoegen: stap 2</h1>
<form th:action="@{/offertes/toevoegen}" th:object='${offerte}' method='post'>
<label>Oppervlakte:
<span th:if="${#fields.hasErrors('oppervlakte')}}"
  th:errors='*{oppervlakte}'></span>
<input th:field='*{oppervlakte}' type='number' required min='1' autofocus>
</label>
<input type='submit' value='Stap 1' name='stap1' formnovalidate>
```



```
<input type='submit' value='Opslaan' name='opslaan'>
</form>
</body>
</html>
```

- (1) Je geeft met `formnovalidate` aan dat als je de form submit met deze knop, de browser de client-sided validaties uitgedrukt met de attributen `required`, `min`, ... niet moet uitvoeren.

Je maakt een method in `OfferteController`.

Deze method wordt uitgevoerd als de gebruiker in de tweede stap op de knop `Stap 1` klikt:

```
@PostMapping(value = "toevoegen", params = "stap1")           ❶
String stap2NaarStap1(Offerte offerte) {                    ❷
    return STAP_1_VIEW;
}
```

- (1) De method `stap2NaarStap1` verwerkt een POST request naar `/offertes/toevoegen`, op voorwaarde dat die een request parameter `stap1` bevat.  
Die request is een een klik op de knop `Stap 1`.
- (2) Als de gebruiker op deze knop klikt, hoef je niets te valideren.

Je maakt een method in `OfferteController`.

Deze method wordt uitgevoerd als de gebruiker in de tweede stap op de knop `Opslaan` klikt:

```
private static final String REDIRECT_URL_NA_TOEVOEGEN = "redirect:/";
@PostMapping(value = "toevoegen", params = "opslaan")
String create(@Validated(Offerte.Stap2.class) Offerte offerte,
    BindingResult bindingResult, SessionStatus sessionStatus) {           ❶
    if (bindingResult.hasErrors()) {
        return STAP_2_VIEW;
    }
    offerteService.create(offerte);
    sessionStatus.setComplete();                                           ❷
    return REDIRECT_URL_NA_TOEVOEGEN;
}
```

- (1) Om de `HttpSession` variabele te verwijderen geef je aan een `@GetMapping` of `@PostMapping` method een `SessionStatus` parameter mee.
- (2) Je verwijdert de `HttpSession` variabele met de `SessionStatus` method `setComplete`.

Je voegt een regel toe aan `application.properties` om session fixation te voorkomen (zie cursus Spring fundamentals):

```
server.session.tracking-modes=cookie
```

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



Aanvraag: zie takenbundel



## 7 MEERTALIGE WEBSITES

Je hebt de teksten van de website tot nu in één taal hard gecodeerd in de HTML bestanden.

Je leert hier een meertalige website maken, waarbij je de teksten in meerdere talen aanbiedt.

Je haalt bij een meertalige website de teksten uit `messages.properties` en variaties van dit bestand. Zo'n `properties` bestand met teksten heet ook een resource bundle.

Één resource bundle bevat teksten in één taal. Één regel is één tekst.

De regel bestaat uit een unieke sleutel, gevolgd door `=`, gevolgd door de tekst zelf.

Je geeft de taal, die bij de resource bundle hoort, aan in de bestandsnaam:

Je eindigt de bestandsnaam met een `_`, gevolgd door de taalcode (`messages_nl.properties`)

Men spreekt sommige talen in meerdere landen.

Je kan de tekst per land vertalen in een resource bundle specifiek voor dat land.

Je geeft dit land ook aan in de bestandsnaam (`messages_nl_BE.properties`)

Spring bepaalt de taal van de gebruiker standaard op de request header `Accept-Language`.

Die bevat een taalcode (bvb. `nl`) of een combinatie van een taalcode en een landcode (bvb. `nl-BE`).

Als je een tekst ophaalt, doorzoekt Spring de resource bundles in deze volgorde:

- De resource bundle met de gebruikerstaal- én landcode (`messages_nl_BE.properties`).
- Als de tekst daarin niet voorkomt, de resource bundle met enkel de taalcode (`messages_nl.properties`).
- Als de tekst daarin niet voorkomt, de resource bundle zonder taalcode (`messages.properties`).



Je kan meerdere taal-land combinaties instellen in je browser. Spring gebruikt in dit zoekproces *alle* taal-land combinaties in de request header `Accept-Language`.

Als deze header bijvoorbeeld `nl én fr` bevat, zoekt JSP een tekst in `messages_nl.properties`, daarna in `messages_fr.properties` en daarna in `messages.properties`.

Je voegt volgende regels toe aan `messages.properties`.

Nederlands wordt de standaardtaal van de website.

```
aanlegGazon=De aanleg van een gezellig gazon
groeneTenen=Groene tenen
goedeMorgen=goede morgen
goedeMiddag=goede middag
goedeAvond=goede avond
gazon=gazon
```

Je maakt ook een Franstalige versie: `messages_fr.properties`.

De keys van de teksten moeten dezelfde zijn als in `messages.properties`:

```
typeMismatch.java.lang.Integer=appuyez un nombre
aanlegGazon=La construction d'une pelouse confortable
groeneTenen=Orteils verts
goedeMorgen=bonjour
goedeMiddag=bon après-midi
goedeAvond=bonsoir
gazon=pelouse
```



Je kan ook van `ValidationMessages.properties` meerdere versies maken (bijvoorbeeld `ValidationMessages_fr.properties`).

Je wijzigt in `IndexController` de method begroeting:

```
private String begroeting() {
    int uur = LocalDateTime.now().getHour();
    if (uur >= 6 && uur < 12) {
        return "goedeMorgen";
    }
}
```

```

    if (uur >= 12 && uur < 18) {
        return "goedeMiddag";
    }
    return "goedeAvond";
}

```

- (1) Je geeft geen hard gecodeerde Nederlandstalig tekst meer terug, maar een key uit de resource bundles.

Je wijzigt in `index.html` de `<head>`, `<h1>`, `<blockquote>` en `<img>` elementen:

```

...
<head th:replace='fragments::head(title=#{groeneTenen})'></head> ❶
...
<h1><span th:text=#{groeneTenen}></span>,
<span th:text=#{${begroeting}}></span></h1> ❷
<blockquote th:text=#{aanLegGazon}></blockquote>
<img th:alt=#{gazon}' th:src=@{/images/gazon.jpg}' class='fullwidth'>
...


```

- (1) Je haalt de tekst op die hoort bij de key `groeneTenen` uit een resource bundle. Je doet dit met de syntax `#{keyVanDeOpTeHalenTekst}`.
- (2) Je leest de inhoud van de variabele `begroeting` met `${begroeting}`. Deze inhoud is bijvoorbeeld `goedeMorgen`. Je gebruikt deze inhoud als key uit een resource bundle, om de bijbehorende tekst op te halen.

Je kan de website uitproberen.

Je ziet de welkompagina (behalve het menu) in de standaardtaal van de website: Nederlands.

Je stelt Frans in als je voorkeurtaal in Firefox:

1. Je kiest rechts boven in Firefox .
2. Je kiest Options.
3. Je kiest Choose bij Choose your preferred language for displaying pages.
4. Je voegt French toe met Select a language to add en Add.
5. Je kunt French als voorkeurtaal instellen met Move Up en Move Down.

Je kan de website uitproberen. Je ziet de welkompagina (behalve het menu) in het Frans.

Een tekst in `messages.properties` kan parameters bevatten.

Je tikt een parameter als `{volgNrVanDeParameter}`. De volgnummers beginnen per tekst vanaf 0.

Voorbeeld: `filiaalVan=Filiaal {0} van {1} filialen` ❶

- (1) De tekst bevat twee parameters, aangegeven met `{0}` en `{1}`.

Bij het oproepen van de tekst in je HTML bestand, geef je waarden mee voor die parameters:

`#{filiaalVan(${filiaalVolgNr},${aantalFilialen})}` ❶

- (1) Je geeft als eerste parameterwaarde de data mee die de controller meegaf onder de naam `filiaalVolgNr`. Je geeft als tweede parameterwaarde de data mee die de controller meegaf onder de naam `aantalFilialen`.

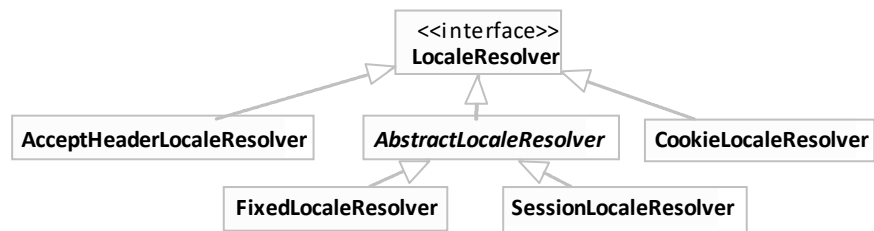
Je voegt volgende regel toe aan `application.properties`:

`spring.messages.fallback-to-system-locale=false` ❶

- (1) Default staat deze property op true.
- Als Spring een tekst niet vindt in de resource bundle met taalcode en landcode van de gebruiker en ook niet in de resource bundel met taalcode van de gebruiker, zoekt Spring de tekst in de resource bundle met de taalcode van het besturingssysteem. Dit is vervelend: je website gedraagt zich anders volgens de taal van het besturingssysteem waarop hij draait. Je plaatst de property daarom op false.
- Als Spring nu een tekst niet vindt in de resource bundle met taalcode en landcode van de gebruiker en ook niet in de resource bundle met taalcode van de gebruiker, zoekt Spring de tekst in de resource bundle zonder taalcode.

## 7.1 Taal en het land van de gebruiker

Meerdere Spring classes bevatten een strategie om de taal en het land van de gebruiker te bepalen:



### 7.1.1 AcceptHeaderLocaleResolver

Dit is de default strategie.

Een browser request bevat een header Accept-Language met de gebruikerstaal en landcode. AcceptHeaderLocaleResolver bepaalt met die header de taal-en landcode van de gebruiker.

### 7.1.2 FixedLocaleResolver

De taal (en eventueel het land) is hierbij voor elke gebruiker dezelfde.

Je kiest deze strategie door een bean van deze class te maken.

Je maakt in de package `be.vdab.groenetenen.web` een class `WebConfig`:

```
package be.vdab.groenetenen.web;
```

```
@Configuration
```

```
class WebConfig {
```

```
    @Bean
```

```
    FixedLocaleResolver localeResolver() {
```

```
        return new FixedLocaleResolver(new Locale("fr", "BE"));
```

```
    }
```

```
}
```

❶

❷

❸

(1) Je tikt `@Configuration` voor een class waarbinnen je met `@Bean` (zie ❷) Spring beans maakt.

(2) Je gebruikt voor het eerst `@Bean` om een Spring bean te maken.

Je schrijft `@Bean` voor een method. Spring maakt, bij het starten van de website van de returnwaarde van die method (hier een `FixedLocaleResolver` object) één Spring bean. Spring houdt die bean bij zolang de website draait.

(3) De class `Locale` stelt een combinatie van een land en een taal voor.

Je kan de website uitproberen. Ongeacht je browserinstelling zie je de welkompagina in het Frans.

### 7.1.3 SessionLocaleResolver

Jij vraagt hierbij de taal (en eventueel het land) van de gebruiker. Je geeft de keuze door aan de `SessionLocaleResolver` bean. Die onthoudt de keuze in een `HttpSession` variabele. De gebruiker behoudt zo zijn taal- en landkeuze zolang hij op de website blijft.

Zolang de gebruiker geen taal of land kiest, gebruikt Spring de taal en landkeuze van de browser.

Je wijzigt de method `localeResolver`:

```
@Bean
```

```
SessionLocaleResolver localeResolver() {
```

```
    return new SessionLocaleResolver();
```

```
}
```

### 7.1.4 LocaleChangeInterceptor

De gebruiker zal in elke pagina, via hyperlinks Nederlands of Français, zijn taal kunnen kiezen.

Je voegt regels toe aan `fragments.html`, voor de laatste `</ul>`:

```
<li><a th:href="@{?Locale=nl_BE}">Nederlands</a></li>
```

❶

```
<li><a th:href="@{?Locale=fr_BE}">Français</a></li>
```

(1) Als de gebruiker deze hyperlink aanklikt, zal hij een request doen naar dezelfde pagina waar hij zich bevond, maar met een request parameter `locale=nl_BE` toegevoegd aan de URL.

#### 7.1.4.1 Interceptor

Als de gebruiker Nederlands of Français kiest, stuurt de browser een request naar een controller. Het zou omslachtig zijn die request in elke controller te verwerken en de taalkeuze door te geven aan de `LocaleResolver` bean. Een betere oplossing is de class `LocaleChangeInterceptor`.

Vooraleer `LocaleChangeInterceptor` te gebruiken, leer je wat een interceptor is. Dit is een class die Spring oproept telkens een browser request binnenkomt en voor die request naar de Controller gestuurd wordt. Een interceptor implementeert de interface `HandlerInterceptor`.

#### 7.1.4.2 De methods in de interface `HandlerInterceptor`

<<interface>> <b>HandlerInterceptor</b>	
+preHandle( <code>HttpServletRequest request</code> , <code>HttpServletResponse response</code> , <code>Object handler</code> ): <code>boolean</code>	
+postHandle( <code>HttpServletRequest request</code> , <code>HttpServletResponse response</code> , <code>Object handler</code> , <code>ModelAndView modelAndView</code> )	
+afterCompletion( <code>HttpServletRequest request</code> , <code>HttpServletResponse response</code> , <code>Object handler</code> , <code>Exception ex</code> )	

- **preHandle**  
Als een request binnenkomt, roept Spring `preHandle` op, daarna de controller method die de request verwerkt. De parameter `handler` is de controller die de request verwerkt. De parameters `request` en `response` geven low-level toegang tot de request en de response. Als de method `false` teruggeeft, roept Spring andere interceptors en de controller niet op.
- **postHandle**  
Spring roept `postHandle` op nadat de controller method de request verwerkte, maar voor de bijbehorende Thymeleaf pagina de request verwerkt. De parameter `modelAndView` bevat de `ModelAndView` die de controller method teruggaf.
- **afterCompletion**  
Spring roept `afterCompletion` op als de request helemaal verwerkt is, ook door de Thymeleaf pagina. Als een exception optrad tijdens het verwerken van de request, bevat de parameter `ex` deze exception. Als geen exception optrad, bevat `ex` de waarde `null`.

#### 7.1.4.3 `LocaleChangeInterceptor`

`LocaleChangeInterceptor` is een interceptor van het Spring framework.

Als de request een parameter `locale` bevat, roept de interceptor de method `setLocale` van de `SessionLocaleResolver` bean op en geeft die request parameter mee.

Je onthoudt zo de taal en het land dat de gebruiker gekozen heeft als HTTP session variabele.

Je moet elke interceptor registreren. Je doet dit in `WebConfig`.

Deze class moet daartoe de interface `WebMvcConfigurer` implementeren:

```
public class WebConfig implements WebMvcConfigurer
```

Je doet de registratie zelf in de method `addInterceptors`,

die gedeclareerd is in de interface `WebMvcConfigurer`:

```
@Override
```

```
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(new LocaleChangeInterceptor());  
}
```

Je kan de website uitproberen.

#### 7.1.5 `CookieLocaleResolver`

Jij vraagt hierbij de taal (en eventueel het land) van de gebruiker.

Je geeft de keuze door aan de `CookieLocaleResolver` bean.

Die onthoudt de keuze in een cookie.

Zolang je de taalkeuze niet vraagt, gebruikt Spring de taal en landkeuze van de browserinstellingen.

Je wijzigt de method `localeResolver`:

`@Bean`

```
CookieLocaleResolver localeResolver() {  
    CookieLocaleResolver resolver = new CookieLocaleResolver();  
    resolver.setCookieMaxAge(604_800);  
    return resolver;  
}
```

❶

- (1) De property `cookieMaxAge` bepaalt de levensduur (in seconden) van de cookie.  
(604800 = 7 dagen).

Je hoeft niets anders te wijzigen. Je kan de website uitproberen.

#### 7.1.6 Taal en het land lezen in een controller bean

Als je de Locale van de gebruiker wil lezen in een `@GetMapping` of `@PostMapping` method, voeg je aan die method een `Locale` parameter toe.

Spring vult die parameter met de Locale van de gebruiker.

De `FiliaalController` method `vanTotPostcode` zou bijvoorbeeld volgende signatuur krijgen:

`ModelAndView vanTotPostcode(Locale locale)`



Je commit de sources. Je publiceert op GitHub.



Meertalig: zie takenbundel

## 8 REST

REST is een afkorting van Representational State Transfer.

Bij REST roept een applicatie via het HTTP protocol de diensten op van een andere applicatie.

- De applicatie die de diensten oproept heet de (REST) client.
- De applicatie die de diensten aanbiedt heet de (REST) service.



Een browser is een client en de website is een service. Ook andere applicaties roepen mekaars diensten op: een reisbureau website roept de diensten op van een vliegtuigmaatschappij service.

REST is gebaseerd op volgende principes, die je straks meer in detail leert kennen:

- De service identificeert de entities die hij aanbiedt aan de client met URI's.
- De client doet GET, POST, PUT en DELETE requests naar die URI's.
- Services en clients wisselen data uit in meerdere formaten: HTML, XML, JSON (het standaard dataformaat van JavaScript).
- De service geeft met de response status code aan of hij de request correct verwerkte.
- De response die de service aanbiedt, bevat naast data ook hyperlinks.
- De service is stateless. Dit betekent dat de service geen data bijhoudt als HTTP session variabelen. Dit is moeilijk haalbaar in een service waarvan de clients browsers zijn met bvb. een winkelmandje, maar is wel haalbaar in een service waarvan de clients geen browsers zijn. De clients onthouden dan zelf hun state (bvb. een winkelmandje).

### 8.1 Entities identificeren met URI's

Elke entity én elke verzameling entities krijgt een URI.

In theorie is deze URI vrij te kiezen. /a1bd89 kan filiaal 1 voorstellen en /9zdb3 filiaal 2

De meeste websites gebruiken echter clean URI's.

De syntax van een clean URI verschilt volgens het soort data die de URI voorstelt.

#### 8.1.1 URI die alle entities van hetzelfde type voorstelt

URI Syntax: /meervoudsvormEntities. /filialen stelt bijvoorbeeld alle filialen voor.

#### 8.1.2 URI die één entity voorstelt

URI syntax: /meervoudsvormEntities/identificatieVanDieEntity  
/filialen/3 stelt bijvoorbeeld het filiaal 3 voor.

#### 8.1.3 URI die een verzameling entities aangeduid met een groepsnaam voorstelt

Een subverzameling entities kan een groepsnaam hebben: hoofdfilialen, bijfilialen.

URI syntax: /meervoudsvormEntities/groepsnaam  
/filialen/hoofdfilialen stelt bijvoorbeeld de hoofdfilialen voor.

#### 8.1.4 URI die een verzameling entities die je zoekt op andere zoekcriteria voorstelt

Je zoekt soms een verzameling entities met andere zoekcriteria dan hier boven. Je gebruikt dan request parameters. /werknemers?vanafwedde=2000 zijn werknemers met een wedde >=2000.

Je kan met request parameters ook de sorteervolgorde aangeven

/filialen?sort=naam stelt bijvoorbeeld alle filialen voor gesorteerd op naam.

#### 8.1.5 Associaties tussen entities

Je kan met een URI een associatie tussen entities uitdrukken.

URI Syntax: /meervoudsvormEntities/identificatie/naamVanDeAssociatie  
/filialen/3/werknemers stelt bijvoorbeeld de werknemers van filiaal 3 voor.

## 8.2 HTTP methods

De client doet een request naar de URI die een entity of een verzameling entities voorstelt. De client duidt met de HTTP method (GET, POST, PUT of DELETE) de handeling aan die hij wil uitvoeren op de entity of verzameling entities. HTML gebruikt enkel GET en POST, maar andere clients kunnen ook PUT en DELETE gebruiken.

### 8.2.1 GET: entity of entities lezen

De client doet een GET request naar de URI die de entity of verzameling entities voorstelt. Hij krijgt een response terug met de data van die entity of entities.

Voorbeelden:

- De client doet een GET request naar /filialen om alle filialen te lezen.
- De client doet een GET request naar /filialen/3 om filiaal 3 te lezen.

### 8.2.2 POST: entity toevoegen

De client doet een POST request naar de URI die de verzameling gelijkaardige entities voorstelt. De request body bevat de data van de toe te voegen entity.

Voorbeeld: de client doet een POST request naar /filialen om een filiaal toe te voegen.

### 8.2.3 PUT: entity wijzigen

De client doet een PUT request naar de URI die de te wijzigen entity voorstelt.

De request body bevat de te wijzigen data van de entity.

Voorbeeld: de client doet een PUT request naar /filialen/3 om filiaal 3 te wijzigen.

### 8.2.4 DELETE: entity verwijderen

De client doet een DELETE request naar de URI die hoort bij de te verwijderen entity.

Voorbeeld: de client doet een DELETE request naar /filialen/3 om filiaal 3 te verwijderen.



HTML gebruikt het onderliggende HTTP protocol, maar ondersteunt enkel GET requests en POST requests.

## 8.3 Formaat van de data

De data in de response kan in XML formaat uitgedrukt zijn, of in JSON formaat, ...

De client geeft in de request header Accept het data formaat aan waarin hij de data in de response verwacht. Vb.: de client doet een GET request naar /filialen/3 met Accept:application/xml. Dit betekent: geef de data van filiaal 3 aub in XML formaat.

De response bevat het filiaal in XML formaat:

```
<?xml version="1.0" encoding="UTF-8"?>
<filiaal id="3">
  <naam>Gavdos</naam>
  <straat>Europalaan</straat>
  <huisnr>37</huisnr>
  <postcode>3600</postcode>
  <gemeente>Genk</gemeente>
  <waardegebouw>3000</waardegebouw>
  <ingebruikname>2009-03-03</ingebruikname>
  <hoofdfiliaal>false</hoofdfiliaal>
</filiaal>
```



Eerder is vermeld dat XML niet meer populair is om configuratie te beschrijven. XML wordt wel nog gebruikt om data uit te wisselen tussen applicaties.

### 8.3.1 Content-type header

De response bevat een header Content-type. Die geeft het formaat van de response data aan. Deze header bevat bijvoorbeeld application/xml als de response data in XML is uitgedrukt.



## 8.4 Response status code

De meest gebruikte response status codes:

Code	Betekenis	Omschrijving
200	OK	De request is correct verwerkt.
201	Created	De client deed een POST request om een entity toe te voegen. Die request is correct verwerkt.
400	Bad request	De client deed een POST request of een PUT request. De entity, die hij meestuurde in de request body, bevat verkeerde data (bvb. een werknemer met een negatieve wedde).
401	Unauthorized	De client deed een request op een beveiligde URI. De client heeft zich nog niet geauthenticeerd, of heeft een verkeerde gebruikersnaam/paswoord meegegeven.
403	Forbidden	De client deed een request op een beveiligde URI. De client heeft zich correct geauthenticeerd, maar heeft niet de rechten om deze request te doen.
404	Not found	De client deed een request naar een URI die de service niet kent.
405	Method not allowed	De client deed een request op een URI met een HTTP method die de URI niet ondersteunt. Voorbeeld: een DELETE request naar <a href="#">/filialen</a>
406	Not acceptable	De service ondersteunt de waarde in request header Accept niet. Voorbeeld: de request header Accept bevat application/xml. De service kan echter enkel responses in JSON formaat produceren.
409	Conflict	De request probeert een entity in een verkeerde toestand te plaatsen. Voorbeeld: een DELETE request naar <a href="#">/filialen/3</a> Filiaal 3 heeft nog werknemers en kan men daarom niet verwijderen.
415	Unsupported Media Type	De request body bevat data in een formaat aangegeven in de header Content. De service ondersteunt dit formaat echter niet. Voorbeeld een POST request body bevat data in XML formaat. De service kan echter enkel data in JSON formaat verwerken.
500	Internal server error	De service heeft interne (bvb. database) problemen om de request te verwerken.

## 8.5 HATEOAS

HATEOAS is een afkorting van Hypermedia As The Engine Of Application State.

HATEOAS betekent dat de response naast data ook links bevat naar andere URI's.

Een pagina, die je in een browser opent, bevat tekst en afbeeldingen.

De pagina bevat (meestal) ook hyperlinks. Die verwijzen naar andere interessante pagina's.

Dit is handig: je hoeft niet de URL's van alle filialen te kennen, enkel één URL: [/filialen](#)

De hyperlinks op die pagina reiken je de URL van elk individueel filiaal aan.

Je gebruikt bij REST dit principe ook als je de client data aanbiedt in XML of JSON formaat.

Als de client een GET request doet naar [/filialen](#) bevat de response, naast data over filialen, ook verwijzingen naar de URI's van de individuele filialen:

```
<?xml version='1.0' encoding='UTF-8'?>
<filialen>
  <filiaal id='1' naam='Andros'>
  <filiaal id='2' naam='DeLos'>
  <link rel='detail.1' href='http://www.mysite.org/filialen/1'>
  <link rel='detail.2' href='http://www.mysite.org/filialen/2'>
</filialen>
```

❶

❷

- (1) Deze regel bevat data over filiaal 1.



- (2) Deze regel bevat de URI van filiaal 1. REST gebruikt hiervoor een link element. href bevat de URI, rel bevat een omschrijving van de data op die URI. De client moet dus de URI waarmee hij de detail van één filiaal opvraagt niet verzinnen, maar krijgt deze hier aangereikt.

Als de client de URI bij (1) volgt en er een GET request naar doet, krijgt hij deze response:

```
<?xml version="1.0" encoding="UTF-8"?>
<filiaal id="1">
  <naam>Andros</naam>
  <straat>Keizerslaan</straat>
  <huisnr>11</huisnr>
  <postcode>1000</postcode>
  <gemeente>Brussel</gemeente>
  <waardegebouw>1000</waardegebouw>
  <ingebruikname>2009-01-01</ingebruikname>
  <hoofdfiliaal>true</hoofdfiliaal>
  <link rel='werknemers' href='http://www.mysite.org/filialen/1/werknemers'>❶
</filiaal>
```

- (1) Dit is weer een HATEOAS voorbeeld: de URI van de werknemers van het huidige filiaal.

## 9 NIET-BROWSER CLIENTS REST REQUESTS

Je leert hier hoe je in de REST service requests verwerkt van niet-browser clients.

### 9.1 FiliaalController

FiliaalController verwerkt enkel browser requests, geen requests van andere clients.

Je wijzigt daartoe de @RequestMapping regel voor deze class:

```
@RequestMapping(path = "filialen", produces = MediaType.TEXT_HTML_VALUE)
```

❶

- (1) produces bevat het data formaat van de response.  
De MediaType constante TEXT\_HTML\_VALUE bevat text/html. Je geeft dus aan dat de response data in HTML formaat bevat. Spring stuurt vanaf nu requests waarvan de URL begint met /filialen naar de controller, als de request header Accept de waarde text/html bevat. Dit is het geval bij browsers, niet bij andere clients.

### 9.2 FiliaalRestController

Je maakt een package be.vdab.groenetenen.restservices

en daarin een class FiliaalRestController. Deze verwerkt niet-browser requests.

```
package be.vdab.groenetenen.restservices;
// enkele imports
@RestController
@RequestMapping("/filialen")
class FiliaalRestController {
    private final FiliaalService filiaalService;
    FiliaalRestController(FiliaalService filiaalService) {
        this.filiaalService = filiaalService;
    }
}
```

❶

❷

- (1) Je gebruikt @RestController bij een controller die niet-browser requests verwerkt.
- (2) De URL is dezelfde als bij FiliaalController, maar zonder produces=MediaType.TEXT\_HTML\_VALUE.  
FiliaalRestController verwerkt daarom requests waarvan de URL begint met /filialen, maar waarvan de request header Accept verschilt van tekst/html.

### 9.3 Java objecten omzetten van en naar XML en JSON

- Als een niet-browser client een GET request verstuurt, verwacht hij een response met data in XML of JSON formaat. Spring helpt Java objecten te converteren naar data in deze formaten.
- Als een niet-browser client een POST of PUT request verstuurt, stuurt de client in de request data mee in XML of JSON formaat. Spring helpt data in deze formaten te converteren naar Java objecten.

#### 9.3.1 Libraries

- Spring converteert Java objecten van en naar XML met de standaard Java library JAXB (Java Architecture for XML Binding).
- Spring converteert Java objecten van en naar JSON met de open source library Jackson.

#### 9.3.2 LocalDate

Standaard kan JAXB geen LocalDate converteren van/naar een String. Je maakt een package be.vdab.groenetenen.adapters.

Je maakt daarin een class toe die JAXB helpt deze conversie te doen:

```

package be.vdab.groenetenen.adapters;
import java.time.LocalDate;
import javax.xml.bind.annotation.adapters.XmlAdapter;
public class LocalDateAdapter extends XmlAdapter<String, LocalDate> {      ❶
    @Override
    public LocalDate unmarshal(String string) throws Exception {          ❷
        return LocalDate.parse(string);
    }
    @Override
    public String marshal(LocalDate date) throws Exception {              ❸
        return date.toString();
    }
}

```

- (1) Een class die JAXB helpt data te converteren erft van XmlAdapter.
- (2) JAXB roept deze method op als JAXB een String naar een LocalDate moet omzetten.
- (3) JAXB roept deze method op als JAXB een LocalDate naar een String moet omzetten.

### 9.3.3 Annotations

Je converteert Filiaal objecten (en hun bijbehorende Adres objecten) van en naar XML of JSON.

Je schrijft daartoe enkele annotations in deze classes. JAXB annotations beginnen met @Xml.

Jackson annotations beginnen met @Json

1. Je tikt @XmlRootElement voor de class Filiaal. JAXB vereist dat je dit schrijft bij een class die het root element voorstelt in XML. De naam van dit root element is gelijk aan de naam van de class, met de eerste letter in kleine letters: <filiaal>.
2. Je tikt @XmlTransient en @JsonIgnore voor de variabele werknemers. Wanneer Spring een Filiaal object omzet naar XML of naar JSON slaat Spring de werknemer data over. Je beperkt zo de omvang van de XML of JSON data.
  - a. Bij een GET request naar /filialen/1 bevat de response algemene data (zonder werknemers) van het filiaal 1.
  - b. Pas bij een GET request naar /filialen/1/werknemers zal de response de werknemers van het filiaal 1 bevatten.
3. Je tikt @XmlJavaTypeAdapter(value = LocalDateAdapter.class) voor de variabele inGebruikName. Je geeft zo de class aan waarmee JAXB de LocalDate in deze variabele kan converteren van/naar een String.
4. Je tikt @XmlAccessorType(XmlAccessType.FIELD) voor de classes Filiaal en Adres. JAXB converteert standaard een object van en naar XML met getters en setters. Met deze annotation heeft JAXB geen getters of setters nodig. Zo blijft de value object class Adres immutable (geen setters) en hoef je geen setId method te schrijven in Filiaal.
5. Je tikt ook @JsonAutoDetect(fieldVisibility=Visibility.ANY) voor dezelfde classes. Jackson converteert standaard een object van en naar JSON met getters en setters. Met deze annotation heeft Jackson geen getters of setters nodig.

## 9.4 GET request naar één entity

### 9.4.1 FiliaalRestController

Je maakt in de class FiliaalRestController een method read.

Die verwerkt een GET request naar /filialen/{filiaal} (bvb. /filialen/1).

De response bevat de data van het gevraagde filiaal.

```

@GetMapping("/{filiaal}")
Filiaal read(@PathVariable Optional<Filiaal> filiaal) {                      ❶
    return filiaal.get();
}

```

- (1) Spring converteert in een `@RestController` class de returnwaarde van een `@GetMapping` method naar XML of JSON en vult hiermee de response body.
  - Als de request een Accept header bevat gelijk aan `application/xml`, vult Spring de response met een XML voorstelling van het filiaal. Spring plaatst de Content-type response header op `application/xml`. Spring doet dit ook als in de request de Accept header ontbreekt.
  - Als de request een Accept header bevat gelijk aan `application/json`, vult Spring de response met een JSON voorstelling van het filiaal. Spring plaatst de Content-type response header op `application/json`.
 Spring data converteert de path variabele met het id van het filiaal object naar een Filiaal object door dit Filiaal object uit de database te lezen.

#### 9.4.2 Postman

Je test requests met de utility Postman. Deze dient om REST requests te testen.

Je downloadt Postman op <https://www.getpostman.com>. Postman start na zijn installatie.

Je kan in het beginvenster een account te maken. Dit hoeft niet:

je kan onder in het venster Skip signing in and take me straight to the app kiezen.

Postman toont zijn hoofdscherm en midden daarin een subscherm. Je mag dit subscherm sluiten.

1. Je tikt naast GET de URL naarwaar je een request stuurt:  
`http://localhost:8080/filialen/1`
2. Je voegt een request header toe onder Headers.
  - a. Je tikt onder Key de header naam: Accept.
  - b. Je tikt bij Value de header waarde: `application/xml`.
3. Je verstuurt de request met de knop Send.
4. Je ziet in de onderste helft onder Body de response body: XML data over filiaal 1.
5. Je ziet in verticaal in het midden, horizontaal rechts de status code: 200 OK.
6. Als je verticaal in het midden (naast Body en Cookies) Headers aanklikt, zie je de response headers. Spring vulde bijvoorbeeld de header Content-type met `application/xml`.
7. Je wijzigt bovenaan bij Headers de Accept header naar `application/json`. Je verstuurt de request. Je ziet nu op de onderste helft bij Body een response met JSON data over filiaal 1.

#### 9.4.3 Status code 404 (Not Found)

Als je een request doet naar een niet-bestaand filiaal, schrijft REST voor dat je een response krijgt met status code 404 ( Not Found ).

**Stap 1:** je werpt in de `@GetMapping` method, die de request binnenkrijgt, een exception als de gevraagde entity niet bestaat.

Je maakt in `be.vdab.exceptions` een class `FiliaalNietGevondenException`:

```
package be.vdab.groenetenen.exceptions;
public class FiliaalNietGevondenException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je wijzigt de code in de `FiliaalRestController` method `read`:

```
if (filiaal.isPresent()) {
    return filiaal.get();
}
throw new FiliaalNietGevondenException();
```

**Stap 2:** je maakt in de controller een method:

```
@ExceptionHandler(FiliaalNietGevondenException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
void filiaalNietGevonden() {
}
```

❶  
❷

- (1) `@ExceptionHandler` betekent dat de method exceptions behandelt die je in `@GetMethod` methods (of `@PostMethod` methods, ...) werpt. De method behandelt enkel exceptions van het type `FiliaalNietGevondenException`.
- (2) `@ResponseStatus` bevat de statuscode die je naar de browser wil sturen als een `FiliaalNietGevondenException` optreedt.

Als een `@GetMapping` method een `FiliaalNietGevondenException` werpt, roept Spring `filiaalNietGevonden` op en maakt dan een response met de status code in `@ResponseStatus`. Je kan dit uitproberen met Postman: een request naar `http://localhost:8080/filialen/666` Je ziet dat de response een status code 404 (Not Found) heeft.

## 9.5 DELETE request

Je maakt in `be.vdab.groenetenen.exceptions` de class `FiliaalHeeftNogWerknemersException`:

```
package be.vdab.groenetenen.exceptions;
public class FiliaalHeeftNogWerknemersException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je maakt een method declaratie in `FiliaalService`:

```
void delete(long id);
```

Je implementeert deze method in `DefaultFiliaalService`:

```
@Override
@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
public void delete(long id) {
    Optional<Filiaal> optionalFiliaal = filiaalRepository.findById(id);
    if (optionalFiliaal.isPresent()) {
        if (!optionalFiliaal.get().getWerknemers().isEmpty()) {
            throw new FiliaalHeeftNogWerknemersException();
        }
        filiaalRepository.deleteById(id);
    }
}
```

Je maakt methods in `FiliaalRestController`:

```
@DeleteMapping("{filiaal}")
void delete(@PathVariable Optional<Filiaal> filiaal) {
    if (!filiaal.isPresent()) {
        throw new FiliaalNietGevondenException();
    }
    filiaalService.delete(filiaal.get().getId());
}
@ExceptionHandler(FiliaalHeeftNogWerknemersException.class)
@ResponseStatus(HttpStatus.CONFLICT)
String filiaalHeeftNogWerknemers() {
    return "filiaal heeft nog werknemers";
}
```

- (1) Het returntype van de method is `void`. Spring maakt dan een response met een lege body.
- (2) De method `filiaalNietGevonden` (vroeger in de code) vertaalt deze exception naar een response met status code 404 (Not Found).
- (3) Deze method oproep kan een `FiliaalHeeftNogWerknemersException` werpen. De method `filiaalHeeftNogWerknemers` (verder in deze code) vertaalt die exception naar een response met status code 409 (Conflict) en een response body `filiaal heeft nog werknemers`.

Je test dit met de Postman. Je wijzigt GET naar DELETE. Je tikt als URL bijvoorbeeld `http://localhost:8080/filialen/1` en je verstuurt de request.

## 9.6 POST request

### 9.6.1 @RequestBody

De client stuurt een POST request naar /filialen om een nieuw filiaal toe te voegen.  
De request body bevat data (in XML of JSON formaat) over het toe te voegen filiaal.

Je maakt een @PostMapping method die deze requests verwerkt.

De method heeft een parameter van het type Filiaal. Je tikt voor die parameter @RequestBody.

- Als de request header Content-type gelijk is aan application/xml, maakt Spring een Filiaal object op basis van de XML data in de request body. Spring geeft dit object door als de method parameter van het type Filiaal.
- Als de request header Content-type gelijk is aan application/json, maakt Spring een Filiaal object op basis van de JSON data in de request body. Spring geeft dit object door als de method parameter van het type Filiaal.

Je kan voor de parameter ook @Valid schrijven.

Spring valideert dan de data in het Filiaal object ten opzichte van de validation annotations.

Als er validatiefouten zijn, voert Spring de @PostMapping niet uit, maar werpt een MethodArgumentNotValidException.

Je leert hoe je deze exception vertaalt naar een response met een status code 400 (Bad Request).

Je maakt een method declaratie in FiliaalService:

```
void create(Filiaal filiaal);
```

Je implementeert deze method in DefaultFiliaalService:

```
@Override
@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
public void create(Filiaal filiaal) {
    filiaalRepository.save(filiaal);
}
```

Je maakt in FiliaalRestController methods create en filiaalMetVerkeerdeProperties:

```
@PostMapping
void create(@RequestBody @Valid Filiaal filiaal) {
    filiaalService.create(filiaal);
}

@ExceptionHandler(MethodArgumentNotValidException.class)
@ResponseStatus(HttpStatus.BAD_REQUEST)
String filiaalMetVerkeerdeProperties(MethodArgumentNotValidException ex) {
    StringBuilder fouten = new StringBuilder();
    ex.getBindingResult().getFieldErrors().forEach(error ->
        fouten.append(error.getField()).append(':')
            .append(error.getDefaultMessage()).append('\n'));
    fouten.deleteCharAt(fouten.length() - 1);
    return fouten.toString();
}
```

- (1) Spring vertaalt met @RequestBody de request body naar een Filiaal object. Spring valideert met @Valid dit Filiaal object ten opzichte van de validation annotations. Als er validatiefouten zijn, voert Spring de method create niet uit, maar werpt een MethodArgumentNotValidException.
- (2) Je voegt het filiaal toe aan de database.
- (3) Je geeft hier aan dat de method die op deze annotation volgt een response naar de browser zal sturen als een MethodArgumentNotValidException optreedt.
- (4) Je geeft hier de status code van die response aan.
- (5) Er zijn validatiefouten opgetreden in de filiaal data. Spring plaatst die fouten in de MethodArgumentNotValidException parameter.
- (6) getBindingResult().getFieldErrors() geeft een verzameling FieldError objecten: de Filiaal properties met validatiefouten.

- (7) Je voegt per validatiefout aan de response body een regel toe met:
  - de naam van de verkeerde property (`error.getField()`)
  - een : teken
  - de foutboodschap (`error.getDefaultMessage()`)
  - een enter teken ("`\n`")
- (8) Je verwijdert het overtollige laatste Enter teken na de laatste regel.

Je test dit met Postman

1. Je kiest POST.
2. Je tikt als URL `http://localhost:8080/filialen`.
3. Je wijzigt onder Body Text naar XML (`application/xml`).  
Postman voegt zelf een request header Content-type toe met de waarde `application/xml`.
4. Je tikt daar onder de request body:

```
<filiaal>
  <naam>Nieuw filiaal</naam>
  <adres>
    <straat>Nieuwstraat</straat>
    <huisNr>1</huisNr>
    <postcode>1000</postcode>
    <gemeente>Brussel</gemeente>
  </adres>
  <hoofdFiliaal>true</hoofdFiliaal>
  <waardeGebouw>777</waardeGebouw>
  <inGebruikName>2011-07-07</inGebruikName>
</filiaal>
```

5. Je verstuurt de request.
6. Je krijgt een response met de status code 200 (OK).

## 9.7 PUT request

Je maakt een method declaratie in `FiliaalService`:

```
void update(Filiaal filiaal);
```

Je implementeert deze method in `DefaultFiliaalService`:

```
@Override
@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
public void update(Filiaal filiaal) {
    filiaalRepository.save(filiaal);
}
```

Je maakt in `FiliaalRestController` een method `update`:

```
@PutMapping("{id}")
void update(@RequestBody @Valid Filiaal filiaal) {
    filiaalService.update(filiaal);
}
```

Je test dit met Postman

1. Je kiest PUT.
2. Je tikt als URL `http://localhost:8080/filialen/1`.
3. Je kiest op het tabblad Method voor PUT.
4. Je kiest boven in het tabblad Body voor String body.
5. Je wijzigt onder Body Text naar XML (`application/xml`).
6. Je tikt daar onder de request body:

```
<filiaal>
  <id>1</id>
  <naam>Andros</naam>
  <adres>
    <straat>Keizerslaan</straat>
    <huisNr>11</huisNr>
    <postcode>1000</postcode>
    <gemeente>Brussel</gemeente>
```



```

</adres>
<hoofdFiliaal>true</hoofdFiliaal>
<inGebruikName>2009-01-01</inGebruikName>
<waardeGebouw>77000</waardeGebouw>
<versie>Tik hier de versie van filiaal 1 in de database</versie>
</filiaal>

```

7. Je verstuurt de request.

8. Je krijgt een response met de status code 200 (OK).

## 9.8 HATEOAS

Bij HATEOAS bevat de response, naast data, ook links naar interessante URL's.

### 9.8.1 Configuratie per REST controller

Je moet de URL aangeven die een type entity voorstelt (/filialen stelt Filiaal entities voor).

Je tikt `@ExposesResourceFor(Filiaal.class)` voor de class `FiliaalRestController`.

Spring HATEOAS zoekt de `@RequestMapping` regel bij de controller class en vindt daarin dat `Filiaal` entities zich bevinden op de URL `/filialen`.

### 9.8.2 EntityLinks

Spring maakt een bean met als interface `EntityLinks`. Je maakt met deze bean `<link ...>` elementen in XML. Je injecteert deze bean in de class `FiliaalRestController`.

Je voegt een private variabele toe:

```
private final EntityLinks entityLinks;
```

Je wijzigt de constructor:

```

FiliaalRestController(FiliaalService filiaalService, EntityLinks entityLinks) {
    this.filiaalService = filiaalService;
    this.entityLinks = entityLinks;
}

```

Je maakt met een `EntityLinks` object `Link` objecten. Die stellen XML `<link ...>` element voor

- `entityLinks.linkToCollectionResource(Filiaal.class)` geeft een `Link` object dat volgend XML element voorstelt: `<link rel='self' href='/filialen'/>`  
`linkToCollectionResource` maakt dus de URL die alle filialen voorstelt.
- `entityLinks.linkToSingleResource(Filiaal.class, 1)` geeft een `Link` object dat volgend XML element voorstelt: `<link rel='self' href='/filialen/1'/>`  
`linkToSingleResource` maakt dus een URL met de filiaal id uit de 2° parameter.
- `entityLinks.linkToSingleResource(Filiaal.class, 1).withRel("Filiaal:1")` geeft een `Link` object dat volgend XML element voorstelt:  
`<link rel='Filiaal:1' href='/filialen/1'/>`  
`withRel` bepaalt dus de inhoud van het attribuut `rel`.

### 9.8.3 Location response header

REST bepaalt dat je na het toevoegen van een entity een response stuurt met

- status code 201 (Created)
- een Location header met de URL van de toegevoegde entity

Je wijzigt de method `create`:

```

@PostMapping
@ResponseStatus(HttpStatus.CREATED)
HttpHeaders create(@RequestBody @Valid Filiaal filiaal) {
    filiaalService.create(filiaal);
    HttpHeaders headers = new HttpHeaders();
    Link link =
        entityLinks.linkToSingleResource(Filiaal.class, filiaal.getId());
    headers.setLocation(URI.create(link.getHref()));
    return headers;
}

```



- (1) Als de method geen exception werpt, vult Spring de response status code met de waarde uit `@ResponseStatus`. De constante `CREATED` bevat de waarde 201 (Created).
- (2) Je maakt in het interne geheugen een voorstelling van het XML element `<link rel='self' href='http://localhost:8080/filialen/11'/>`  
Het href attribuut bevat de URL van het toegevoegde filiaal (als de id 11 is).
- (3) Je maakt met het href attribuut een URI. Je vult de response header Location met die URI

Je kan dit uitproberen met Postman.

#### 9.8.4 Link elementen opnemen in de response body

Bij een GET request naar één filiaal zal de response

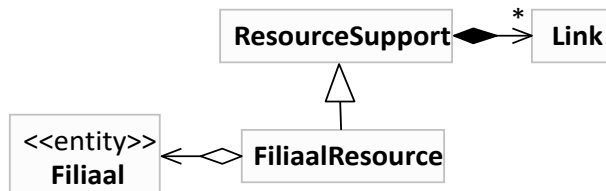
- een "self" link naar het filiaal zelf bevatten
- een link naar de werknemers van dat filiaal bevatten

Voorbeeld:

```
<?xml version='1.0' encoding='UTF-8'?>
<filiaalResource xmlns:atom='http://www.w3.org/2005/Atom'>
  <atom:link href='/filialen/1' rel='self'/>
  <atom:link href='/filialen/1/werknemers' rel='werknemers'/>
  <filiaal>
    <!--hier komen elementen met id, naam, adres, ... van het filiaal -->
  </filiaal>
</filiaalResource>
```

- (1) De opbouw van de link elementen hier onder is met Atom: een XML standaard voor weblogs en nieuws feeds.
- (2) De link bevat de URL van het filiaal zelf. Als de client de response onthoudt, vindt hij later in die link de URL waarmee hij het filiaal kan wijzigen, verwijderen, ...
- (3) De link bevat de URL van interessante data over dit filiaal: de werknemers die er werken

Je maakt een class `FiliaalResource` die deze response voorstelt:



- `FiliaalResource` bevat een `Filiaal`. De XML voorstelling van een `FiliaalResource` bevat zo ook de data van dit `Filiaal`.
- `FiliaalResource` erft van `ResourceSupport` een verzameling `Link` objecten. De XML voorstelling van een `FiliaalResource` bevat zo ook `<link .../>` elementen

```
package be.vdab.groenetenen.restservices;
// enkele imports
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
class FiliaalResource extends ResourceSupport {
  @SuppressWarnings("unused")
  private Filiaal filiaal;
  FiliaalResource() {} // JAXB heeft een default constructor nodig
  FiliaalResource(Filiaal filiaal, EntityLinks entityLinks) {
    this.filiaal = filiaal;
    this.add(entityLinks.linkToSingleResource(
      Filiaal.class, filiaal.getId()));
    this.add(entityLinks.linkForSingleResource(Filiaal.class, filiaal.getId())
      .slash("werknemers").withRel("werknemers"));
  }
}
```

- (1) Je erft van de class ResourceSupport en je hebt zo een verzameling Link objecten.
- (2) Je leest de variabele filiaal nergens in je code.  
Je vermijdt hier de warning die de compiler daarbij kan geven.
- (3) Je neemt een Filiaal object op. De XML voorstelling hiervan wordt dan een onderdeel van de XML voorstelling van een FiliaalResource.
- (4) Je voegt aan de verzameling Link objecten een link toe naar het filiaal zelf.  
De XML voorstelling van een FiliaalResource object zal ook deze link bevatten.
- (5) Je voegt aan de verzameling Link objecten een link toe naar de werknemers van het filiaal.  
De XML voorstelling van een FiliaalResource object zal ook deze link bevatten.  
Je bouwt de link op als de link naar het filiaal zelf, een slash en het woord werknemers.  
Je vult het rel attribuut van de link met het woord werknemers

Je gebruikt deze class in de FiliaalRestController method read:

```
@GetMapping("{filiaal}")
FiliaalResource read(@PathVariable Optional<Filiaal> filiaal) {
    if (filiaal.isPresent()) {
        return new FiliaalResource(filiaal.get(), entityLinks);
    }
    throw new FiliaalNietGevondenException();
}
```

Je test dit met Postman, via een GET request naar <http://localhost:8080/filiaalen/1>.

### 9.8.5 Response met een verzameling entities

Als de client een request doet naar /filialen, zal hij volgende reponse krijgen:

```
<?xml version="1.0" encoding="UTF-8"?>
<filialenResource xmlns:atom="http://www.w3.org/2005/Atom">
  <atom:link href="/filialen/1" rel="detail.1"/>
  <atom:link href="/filialen/2" rel="detail.2"/>
  ...
  <atom:link href="/filialen" rel="self"/>
  <filiaal id="1" naam="Andros"/>
  <filiaal id="2" naam="Delos"/>
  ...
</filialenResource>
```

- (1) De response bevat per filiaal de URL van dat filiaal.
- (2) De response bevat per filiaal de belangrijkste data: id en naam.  
Als de client detail van dat filiaal wil, doet hij een GET request naar de URL bij (1).

Je maakt een class die één regel zoals bij (2) voorstelt:

```
package be.vdab.groenetenen.restservices;
// enkele imports
@XmlAccessorType(XmlAccessType.FIELD)
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
class FiliaalIdNaam {
    @XmlAttribute
    private long id;
    @XmlAttribute
    private String naam;
    FiliaalIdNaam() {} // JAXB heeft een default constructor nodig
    FiliaalIdNaam(Filiaal filiaal) {
        this.id = filiaal.getId();
        this.naam = filiaal.getNaam();
    }
}
```

- (1) Met deze annotation wordt id een XML attribuut in plaats van een XML element.

Je maakt een class die de voorstelling is van de volledige response body:

```
package be.vdab.groenetenen.restservices;
// enkele imports
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
class FilialenResource extends ResourceSupport {
    @XmlElement(name="filiaal")
    @JsonProperty("filialen")
    private final List<FiliaalIdNaam> filialenIdNaam = new ArrayList<>();
    FilialenResource() {} // JAXB heeft een default constructor nodig
    FilialenResource(Iterable<Filiaal> filialen, EntityLinks entityLinks) {
        for (Filiaal filiaal : filialen) {
            this.filialenIdNaam.add(new FiliaalIdNaam(filiaal));
            this.add(entityLinks.linkToSingleResource(Filiaal.class, filiaal.getId())
                .withRel("detail." + filiaal.getId()));
        }
        this.add(entityLinks.linkToCollectionResource(Filiaal.class));
    }
}
```

- (1) Standaard zou het XML element dat één filiaal voorstelt filialenIdNaam heten, gebaseerd op de naam van de private variabele. Met deze annotatie wijzig je deze naam naar filiaal.
- (2) Standaard zou de JSON property die de verzameling filialen voorstelt filialenIdNaam heten. Met deze annotatie wijzig je deze naam naar filialen.
- (3) Je voegt per filiaal een item toe met het id en de naam van het filiaal.
- (4) Je voegt per filiaal een link met de URL van dat filiaal toe. Je vult het rel attribuut van deze link met detail., gevolgd door het filiaalnummer.
- (5) Je voegt aan de response een link toe naar alle filialen.

Je maakt een method declaratie in FiliaalService:

```
List<Filiaal> findAll();
```

Je implementeert deze method in DefaultFiliaalService:

```
@Override
public List<Filiaal> findAll() {
    return filiaalRepository.findAll();
}
```

Je maakt in FiliaalRestController de method findAll:

```
@GetMapping
FilialenResource findAll() {
    return new FilialenResource(filiaalService.findAll(), entityLinks);
}
```

Je test dit met Postman, via een GET request naar <http://localhost:8080/filialen>



Je commit de sources. Je publiceert op GitHub.



REST: zie takenbundel

## 10 INTEGRATION TEST VAN EEN REST SERVICE

Je leert hier hoe je in een integration test HTTP requests stuurt naar je REST service, om de correcte werking van die service te testen.

Je maakt de source folder `src/test/resources`:

1. Je klikt met de rechtermuisknop op je project in de Project Explorer.
2. Je kiest New, Source Folder.
3. Je tikt `src/test/resources` bij Folder name.
4. Je kiest Finish.

Je maakt `insertFiliaal.sql` in deze folder:

```
insert into filialen(naam,hoofdFiliaal,straat,huisNr,postcode,gemeente, versie)
values ('test', false, 'test', 'test', 1000, 'test', 0);
```

Je maakt in de folder `src/test/java` een package `be.vdab.groenetenen.restservices`.

Je maakt daarin een class `FiliaalRestControllerTest`:

```
package be.vdab.groenetenen.restservices;
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.xpath;
// enkele andere imports
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
@Sql("/insertFiliaal.sql")
public class FiliaalRestControllerTest
    extends AbstractTransactionalJUnit4SpringContextTests{
    @Autowired
    private MockMvc mvc;
    private long idVanTestFiliaal() {
        return super.jdbcTemplate.queryForObject(
            "select id from filialen where naam='test'", Long.class);
    }
    @Test
    public void filiaalLezenDatNietBestaat() throws Exception {
        mvc.perform(get("/filialen/-1")
            .accept(MediaType.APPLICATION_XML))
            .andExpect(status().isNotFound());
    }
    @Test
    public void filiaalDatBestaatLezenInXMLFormaat() throws Exception {
        long id = idVanTestFiliaal();
        mvc.perform(get("/filialen/" + id)
            .accept(MediaType.APPLICATION_XML))
            .andExpect(status().isOk())
            .andExpect(content()
                .contentTypeCompatibleWith(MediaType.APPLICATION_XML))
            .andExpect(xpath("/filiaalResource/filiaal/id")
                .string(String.valueOf(id)));
    }
}
```

```

@Test
public void filiaalDatBestaatLezenInJSONFormaat() throws Exception {
    long id = idVanTestFiliaal();
    mvc.perform(get("/filialen/" + id)
        .accept(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(content()
            .contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.filiaal.id").value((int) id));
}
}

```

12

- (1) `@SpringBootTest` maakt alle beans (Controllers, Services, Repositories, ...) en injecteert ze in mekaar.
- (2) `@AutoConfigureMockMvc` maakt een object van de class `MockMvc`. Je kan met zo'n object HTTP requests versturen in de integration test.
- (3) Je injecteert dit `MockMvc` object.
- (4) Je stuurt een GET request naar de URL van een niet-bestaand filiaal. De method `get` is een static method van de class `MockMvcRequestBuilders`. Je importeerde deze method boven in de source met `static import`.
- (5) Je plaatst de request header `accept` op `application/xml`.
- (6) Je controleert of de response status code `Not Found (404)` is. De method `status` is een static method van de class `MockMvcResultMatchers`. Je importeerde deze method boven in de source met `static import`.
- (7) Je stuurt een GET request naar de URL van het filiaal dat je aan de database toevoegde.
- (8) Je controleert of de response status code `OK (200)` is.
- (9) Je controleert of de response header `content-type` gelijk is aan `application/xml`.
- (10) Je zoekt met `XPath` in de XML data in de response body in het element `filiaalResource` het child element `filiaal` en daarin het child element `id`. De method `xpath` is een static method van de class `MockMvcResultMatchers`. Je importeerde deze method boven in de source met `static import`.
- (11) Je controleert of de inhoud van dit element `id` gelijk is aan de `id` van het filiaal dat je aan de database toevoegde.
- (12) Je zoekt met `JSONPath` in de response body in het JSON object dat de volledige response body vult (voorgesteld door `$`) de eigenschap `filiaal` en daarin de eigenschap `id`. De method `jsonPath` is een static method van de class `MockMvcResultMatchers`. Je importeerde deze method boven in de source met `static import`. Je controleert of de inhoud van deze property `id` gelijk is aan de `id` van het filiaal dat je aan de database toevoegde.

Je kan de integration test uitvoeren.



Je commit de sources. Je publiceert op GitHub.

## 11 REST CLIENT

### 11.1 RestTemplate

Je doet requests naar een REST service met de Spring class RestTemplate.

#### 11.1.1 DELETE request om een filiaal te verwijderen

Als de variabele restTemplate van het type RestTemplate is, stuur je met volgend code fragment een DELETE request naar `http://localhost:8080/filialen/1`:

```
try {
    restTemplate.delete("http://localhost:8080/filialen/1");
} catch (HttpStatusCodeException ex) {
    switch (ex.getStatusCode()) {
        case NOT_FOUND:
            // hier komt code als het filiaal niet gevonden is
        case CONFLICT:
            // hier komt code als het filiaal niet kon verwijderd worden
            // omdat het bijvoorbeeld nog werknemers heeft
    }
}
```

- (1) Je stuurt een DELETE request naar de URL die je als parameter meegeeft.
- (2) Als de response status code een fout aangeeft, werpt RestTemplate een HttpStatusCodeException.
- (3) De response status code is 404 (Not Found).
- (4) De response status code is 409 (Conflict).

#### 11.1.2 POST request om een filiaal toe te voegen

Als de variabele filiaal een object is van een class die de XML structuur van een filiaal voorstelt, stuur je met volgend code fragment een POST request naar `http://localhost:8080/filialen`:

```
try {
    URI uri = restTemplate.postForLocation(
        "http://localhost:8080/filialen", filiaal);
} catch (HttpStatusCodeException ex) {
    if (ex.getStatusCode() == HttpStatus.BAD_REQUEST) {
        String foutMeldingen = ex.getResponseBodyAsString();
        // hier komt code als de request verkeerde data bevatte
    }
}
```

- (1) Je verstuurt een POST request naar de URL die je als parameter meegeeft. Je geeft als tweede parameter een object mee met filiaal data. RestTemplate converteert de data naar XML en plaatst die in de request body. Je krijgt de inhoud van de response header Location als een URI object.
- (2) De response status code is 400 (Bad request).
- (3) Je leest de inhoud van de response body (die misschien foutmeldingen bevat).

#### 11.1.3 PUT request om een filiaal te wijzigen

Je stuurt met volgend code fragment een PUT request naar `http://localhost:8080/filialen/1`:

```
try {
    restTemplate.put("http://localhost:8080/filialen/1", filiaal);
} catch (HttpStatusCodeException ex) {
    switch (ex.getStatusCode()) {
        case NOT_FOUND:
            // hier komt code als het filiaal niet bestaat
        case BAD_REQUEST:
            String foutMeldingen = ex.getResponseBodyAsString();
            // hier komt code als de request verkeerde data bevatte
    }
}
```

- (1) Je verstuurt een PUT request naar de URL die je als parameter meegeeft.  
Je geeft als tweede parameter een object mee met filiaal data.  
RestTemplate converteert de data naar XML en plaatst die in de request body.

#### 11.1.4 GET request om een filiaal te lezen

Je stuurt met volgend code fragment een GET request naar

<http://localhost:8080/filialen/1>:

```
try {
    Filiaal filiaal = restTemplate.getForObject(
        "http://localhost:8080/filialen/1", Filiaal.class);
} catch (HttpStatusCodeException ex) {
    if (ex.getStatusCode() == HttpStatus.NOT_FOUND) {
        // hier komt code als het filiaal niet bestaat
    }
}
```

❶

- (1) Je verstuurt een GET request naar de URL die je als parameter meegeeft.  
Je geeft als tweede parameter een class mee.  
RestTemplate converteert de XML (of JSON) van de response body  
naar een object van deze class en geeft je dit object als returnwaarde.

## 11.2 Voorbeeld

Je zal een GET request sturen naar

[http://data.fixer.io/api/latest?access\\_key=TikHierJeKey&symbols=USD](http://data.fixer.io/api/latest?access_key=TikHierJeKey&symbols=USD).

De response bevat JSON:

Je toont daarmee de gebouwwaarde  
van een filiaal in \$.

Je zal terug je fixer API key gebruiken uit de cursus  
Spring Fundamentals.

```
{
  "success":true,
  "timestamp":1526459827,
  "base":"EUR",
  "date":"2018-05-16",
  "rates":{"USD":1.183572}
}
```

### 11.2.1 Structuur van de XML data en/of JSON data

Je stelt de structuur van de XML of JSON data voor als Java classes en member variabelen.

Een class die enkel als doel heeft de data voor te stellen die uitgewisseld worden tussen een REST client en een REST service heet een DTO class. DTO staat voor Data Transfer Object.

Je maakt enkel voorstellingen van data die je applicatie nodig heeft.

Je maakt bijvoorbeeld geen voorstelling van het attribuut date in ons voorbeeld.

Je maakt een package `be.vdab.groenetenen.restclients` met een class `Rates`.

Die stelt het onderdeel rates voorstelt in de JSON data:

```
package be.vdab.groenetenen.restclients;
// enkele imports
class Rates {
    @JsonProperty("USD")
    private BigDecimal usd; // en een getter en een setter
}
```

❶

- (1) Standaard denkt Spring dat bij de variabele `usd` een JSON attribuut `usd` hoort.  
Je geeft hier aan dat bij de variabele een JSON attribuut `USD` hoort.

Je maakt ook een class `USDRate`, die de gehele JSON data voorstelt.

De naam van deze class is vrij te kiezen.

```
package be.vdab.groenetenen.restclients;
class USDRate {
    private Rates rates; // en een getter en een setter
}
```

❶



- (1) Deze variabele stelt het attribuut rates voor in de JSON data.

### 11.2.2 URI van de service

De service URI kan wijzigen nadat de applicatie af is. Je neemt hem daarom op in `application.properties`:

`fixerKoersURL=http://data.fixer.io/api/latest?access_key=TikHierJeKey&symbols=USD`

### 11.2.3 Exceptions

Je maakt in `be.vdab.groenetenen.exceptions` een class `KanKoersNietLezenException`:

```
package be.vdab.groenetenen.exceptions;
public class KanKoersNietLezenException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

### 11.2.4 RestClient

Je maakt in `be.vdab.groenetenen.restclients` een interface `KoersClient`:

```
package be.vdab.groenetenen.restclients;
import java.math.BigDecimal;
public interface KoersClient {
    BigDecimal getDollarKoers();
}
```

Je implementeert de interface in een class `FixerKoersClient`:

```
package be.vdab.groenetenen.restclients;
// enkele imports
@Component
class FixerKoersClient implements KoersClient {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(FixerKoersClient.class);
    private final URI fixerURL;
    private final RestTemplate restTemplate;
    FixerKoersClient(@Value("${fixerKoersURL}") URI fixerURL,
        RestTemplateBuilder restTemplateBuilder) {
        this.fixerURL = fixerURL;
        this.restTemplate = restTemplateBuilder.build();
    }
    @Override
    public BigDecimal getDollarKoers() {
        try {
            USDRate rate = restTemplate.getForObject(fixerURL, USDRate.class);
            return rate.getRates().getUSD();
        } catch (RestClientException ex) {
            LOGGER.error("kan koers niet lezen", ex);
            throw new KanKoersNietLezenException();
        }
    }
}
```

- (1) Spring bevat een bean van het type `RestTemplateBuilder`. Deze class is een toepassing van het factory design pattern: ze dient om een `RestTemplate` object te helpen maken.

Je schrijft een integration test voor `FixerKoersClient` in `src/test/java`:

```
package be.vdab.groenetenen.restclients;
// enkele imports
@RunWith(SpringRunner.class)
@SpringBootTest
public class FixerKoersClientTest {
    @Autowired
    private FixerKoersClient client;
```



```

@Test
public void deKoersMoetPositiefZijn() {
    assertTrue(client.getDollarKoers().compareTo(BigDecimal.ZERO) > 0);
}
}

```

- (1) Je hebt niet enkel een bean nodig van de te testen class (FixerKoersClient).  
 Je hebt daarnaast ook een bean nodig die FixerKoersClient gebruikt: een bean van de class RestTemplateBuilder (). RestTemplateBuilder gebruikt op zijn beurt intern andere beans.  
 Je laadt daarom niet enkel de te testen bean, maar alle beans van je applicatie.

Je voert de test uit. Hij lukt.

Je maakt in be.vdab.groenetenenen.services een interface EuroService:

```

package be.vdab.groenetenenen.services;
import java.math.BigDecimal;
public interface EuroService {
    BigDecimal naarDollar(BigDecimal euro);
}

```

Je implementeert de interface in een class EuroServiceImpl in dezelfde package:

```

package be.vdab.groenetenenen.services;
// enkele imports
@Service
class EuroServiceImpl implements EuroService {
    private final KoersClient koersClient;
    EuroServiceImpl(KoersClient koersClient) {
        this.koersClient = koersClient;
    }
    @Override
    public BigDecimal naarDollar(BigDecimal euro) {
        return euro.multiply(koersClient.getDollarKoers())
            .setScale(2, RoundingMode.HALF_UP);
    }
}

```

Je maakt in be.vdab.groenetenenen.web een class EuroDollar:

```

package be.vdab.groenetenenen.web;
// enkele imports
class EuroDollar {
    @NumberFormat
    private final BigDecimal euro;
    @NumberFormat
    private final BigDecimal dollar;
    // je maakt een geparametriseerde constructor en getters
}

```

Je maakt in be.vdab.groenetenenen.web een class EuroController:

```

package be.vdab.groenetenenen.web;
// enkele imports
@Controller
@RequestMapping("euro")
class EuroController {
    private static final String VIEW = "euro/naardollar";
    private final EuroService euroService;
    EuroController(EuroService euroService) {
        this.euroService = euroService;
    }
}

```

```

@GetMapping("/{euro}/naardollar")
ModelAndView naarDollar(@PathVariable BigDecimal euro) {
    ModelAndView modelAndView = new ModelAndView(VIEW);
    try {
        modelAndView.addObject(new EuroDollar(euro, euroService.naarDollar(euro)));
    }
    catch (KanKoersNietLezenException ex) {
        modelAndView.addObject("fout", "Kan koers niet lezen");
    }
    return modelAndView;
}
}

```

Je maakt in de map templates een map euro. Je maakt daarin naardollar.html:

```

<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace='fragments::head(title='Naar dollar')'></head>
<body>
<nav th:replace='fragments::menu'></nav>
<div th:if='${fout == null}'>
<h1>&euro;
<span th:text='${euroDollar.euro}'></span>
is $
<span th:text='${euroDollar.dollar}'></span>
</h1>
</div>
<div th:if='${fout != null}'>
<div th:text='${fout}' class='fout'></div>
</div>
</body>
</html>

```

Je vervangt in filiaal.html <dd th:text='\${filiaal.waardeGebouw}'></dd> door

```

<dd><span th:text='${filiaal.waardeGebouw}'></span>
<a th:href='@{/euro/{euro}/naardollar(euro=${filiaal.waardeGebouw})}'>in
$</a></dd>

```

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



Tip: je vindt op <http://www.programmableweb.com/apis/directory> een overzicht van interessante rest services.



Tempratuur: zie takenbundel

## 12 JAVASCRIPT CODE ALS REST CLIENT (AJAX REQUESTS)



Je maakt in dit hoofdstuk een pagina perid waarin JavaScript code de rol speelt van REST client. Je kan dit voorbeeld volgen zonder voorkennis van JavaScript.

De gebruiker tikt een filiaal id in een textbox in de pagina en klikt daarna op een knop Zoeken. Jij doet dan met JavaScript een GET request naar de URL van dat filiaal (bvb. /filialen/1). Je toont in JavaScript, met de JSON data uit de response, de naam en het adres van dat filiaal.

Je doet dit zonder de pagina te submitten. Je stuurt enkel JSON data over het te zoeken filiaal van de webserver naar de browser. Dit leidt tot snelle interactieve pagina's.

De gebruiker ziet de pagina als hij een request doet naar /filialen/perid

Je maakt een method die deze request verwerkt in FiliaalController:

```
private static final String PER_ID_VIEW = "filialen/perid";
@GetMapping("perid")
String findById() {
    return PER_ID_VIEW;
}
```

Je maakt perid.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Filialen per id')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>Filialen per id</h1>
<label>Id:
<input id='filiaalId' required type='number' min='1' autofocus/></label>
<button id='zoeken'>Zoeken</button>
<dl>
<dt>Naam</dt>
<dd id='naam'></dd>
<dt>Adres</dt>
<dd id='adres'></dd>
</dl>
<script th:src='@{/scripts/perid.js}'></script>
</body>
</html>
```

Je maakt in de map static een map scripts. Je maakt daarin perid.js:

```
document.getElementById('zoeken').onclick = zoekFiliaal;
function zoekFiliaal() {
    const filiaalIdVak = document.getElementById('filiaalId');
    if (! filiaalIdVak.validity.valid) {
        alert("Ongeldige id");
        filiaalIdVak.focus();
        return;
    }
    const request = new XMLHttpRequest();
    request.open("GET", filiaalIdVak.value, true);
    request.setRequestHeader('accept', 'application/json');
    request.onload = responseIsBinnengekomen;
    request.send();
}
function responseIsBinnengekomen() {
    switch (this.status) {
        case 200:
            const filiaalResource = JSON.parse(this.responseText);
            const filiaal = filiaalResource.filiaal;
```

①  
②  
③  
④  
⑤  
⑥  
⑦  
⑧  
⑨  
⑩  
⑪  
⑫  
⑬  
⑭

```

document.getElementById('naam').innerHTML = filiaal.naam;
const adres = filiaal.adres;
document.getElementById('adres').innerHTML = adres.straat + ' '
    + adres.huisNr + ' ' + adres.postcode + ' ' + adres.gemeente;
break;
case 404:
    alert('Filiaal bestaat niet');
    break;
default:
    alert("Technisch probleem");
}
}

```

- (1) Je zoekt de knop: het HTML element met het attribuut id gelijk aan zoeken. Je geeft aan dat, bij een klik op de knop, de functie zoekFiliaal moet uitgevoerd worden.
- (2) Je zoekt het invoervak: het HTML element met het attribuut id gelijk aan filiaalId.
- (3) Als de inhoud van dit vak verkeerd is (leeg gelaten, geen positief getal, ...)
- (4) toon je een popup venster met een foutmelding
- (5) en plaats je de cursor in dit invoervak.
- (6) Je maakt een variabele request en verwijst die naar een object van het type XMLHttpRequest. Je verstuurt met zo'n object HTTP requests vanuit JavaScript.
- (7) Je geeft aan dat je een GET request wil sturen naar de URL van het te zoeken filiaal. De 2° parameter is de URL waarnaar je de request zal versturen. Je geeft de inhoud van het invoervak mee. De URL wordt dan de URL van de huidige pagina (/filialen/perid) waarin perid vervangen wordt door de inhoud (value) van het invoervak. Je geeft true mee als 3° parameter. De browser zal de request dan versturen in een achtergrondtaak en de response verwerken in dezelfde achtergrondtaak. Op die manier blokkeert de browser niet gedurende het verwerken van de request. Zo kan de gebruiker ondertussen andere handelingen doen in de browser. Op dit moment verstuur je de request nog niet.
- (8) Je vult de request header accept met application/json. Je vraagt zo dat de data in de response body zal uitgedrukt zijn in JSON. JavaScript kan makkelijk JSON data verwerken.
- (9) Je geeft aan dat, als de response binnengekomen is (onload), de functie responseIsBinnengekomen moet uitgevoerd worden.
- (10) Je verstuurt hier pas de request. Zo'n request heet een AJAX request.
- (11) Je controleert de response status.
- (12) Als die gelijk is aan 200 (OK), heeft de webserver de request correct verwerkt.
- (13) Je vindt de response body in responseText. Je converteert met de method parse van het type JSON de JSON data in die response body naar een JavaScript object.
- (14) Je zoek in dit JavaScript object de eigenschap filiaal. Deze eigenschap is zelf ook een JavaScript object met filiaal eigenschappen (id, naam, ...)
- (15) Je zoekt het element met de id naam. Dit is het eerste <dd> element in de pagina. innerHTML stelt de tekst voor tussen <dd ...> en </dd>. Je vervangt deze tekst door de naam van het filiaal.
- (16) De response status is 404 (Not Found).
- (17) Je toont een popup venster met een foutmelding.
- (18) De response status verschilt van 200 en van 404.

Je kan de pagina uitproberen.



Je commit de sources. Je publiceert op GitHub.

## 13 MAIL STUREN

Je kan met Spring vanuit je code een mail sturen.

Je zal, als voorbeeld, wanneer een offerte toegevoegd werd, een mail sturen.

### 13.1 Mail server configuratie

Zoals een applicatie een databaseserver aanspreekt om met een database te werken, spreekt een applicatie een mailserver aan om mails te versturen.

Je hoeft in deze cursus op je PC geen mailserver te installeren.

Als je een Google account hebt, kan je de mailserver van Google gebruiken.

Je voegt regels, met eigenschappen van de mailserver, toe aan `application.properties`.

Let er op dat de regels op het einde geen overvloedige spaties bevatten !

```
spring.mail.host=smtp.gmail.com
spring.mail.port=465
spring.mail.protocol=smtps
spring.mail.username=eenGebruikersNaam@gmail.com
spring.mail.password=paswoordVanDieGebruiker
```

❶  
❷  
❸  
❹  
❺

- (1) De netwerknnaam van de computer waarop de mailserver geïnstalleerd is.
- (2) Het TCP-IP nummer van de mailserver.
- (3) Het protocol van de mailserver.  
SMTP (Simple Mail Transfer Protocol) is het standaard protocol waarmee je mails verstuurt. SMTPS beveiligt het SMTP protocol met authenticatie en versleuteling van de boodschap.
- (4) De gebruikersnaam onder wiens naam je mails stuurt.  
Gebruik hier niet je persoonlijke Google account, maar een nieuwe Google account. Zo raakt je account informatie niet bekend als je de project sources deelt met anderen (wat bijvoorbeeld gebeurt als je het project publiceert op GitHub).  
Om vanuit code mails te versturen moet je op de pagina <https://www.google.com/settings/security/lesssecureapps> inschakelen kiezen.
- (5) Het paswoord van deze gebruiker.

Spring maakt een bean die de interface `JavaMailSender` implementeert en initialiseert deze class met de mail eigenschappen uit `application.properties`. Als je in jouw code een mail wil versturen injecteer je deze bean in je code. De bean helpt je een mail te versturen.

### 13.2 Bean die de mail stuurt

Je maakt eerst een exception class:

```
package be.vdab.groenetenen.exceptions;
public class KanMailNietZendenException extends RuntimeException {
    private static final long serialVersionUID = 1L;
}
```

Je maakt een package `be.vdab.groenetenen.mail` en daarin een interface `MailSender`:

```
package be.vdab.groenetenen.mail;
import be.vdab.groenetenen.entities.Offerte;
public interface MailSender {
    void nieuweOfferte(Offerte offerte);
}
```

Je implementeert de interface in de class `DefaultMailSender`:

```
package be.vdab.groenetenen.mail;
// enkele imports
@Component
class DefaultMailSender implements MailSender {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(DefaultMailSender.class);
    private final JavaMailSender sender;
```

```

DefaultMailSender(JavaMailSender sender) {
    this.sender = sender;
}
@Override
public void nieuweOfferte(Offerte offerte) {
    try {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setTo(offerte.getEmailAdres());
        message.setSubject("Nieuwe offerte");
        message.setText("Uw offerte heeft het nummer" + offerte.getId());
        sender.send(message);
    } catch (MailException ex) {
        LOGGER.error("Kan mail nieuwe offerte niet versturen", ex);
        throw new KanMailNietZendenException();
    }
}
}

```

- (1) SimpleMailMessage stelt een email zonder opmaak voor.
- (2) Je vult de eigenschappen van de email in.
- (3) Je verstuurt de email.

### 13.3 Services layer

Je injecteert deze bean in de class DefaultOfferteService.

Je voegt een private variabele toe:

```
private final MailSender mailSender;
```

Je injecteert de bean met een uitbreiding van de constructor:

```

DefaultOfferteService(OfferteRepository offerteRepository, MailSender mailSender) {
    this.offerteRepository = offerteRepository;
    this.mailSender = mailSender;
}

```

Je voegt een opdracht toe als laatste in de method create:

```
mailSender.nieuweOfferte(offerte);
```

Je kan de website uitproberen. Opmerking: een mail versturen vraagt wat tijd.

### 13.4 Mail met opmaak en een hyperlink

Je maakt een hyperlink naar <localhost:8080/offertes/9> in de tekst van de email, als de nieuwe offerte het nummer 9 heeft.

Als de mail ontvanger deze mail aanklikt, ziet hij in je website de detail van zijn offerte.

Je voegt een HttpServletRequest parameter toe aan de OfferteController method create.

HttpServletRequest bevat low-lever informatie over de huidige browser request, waaronder de URL van de request. Deze URL is gedurende het ontwikkelen <localhost:8080/offertes/toevoegen> en in productie [groenetenen.com/offertes/toevoegen](https://groenetenen.com/offertes/toevoegen).

Je baseert de hyperlink in de email op deze URL.

Je vervangt in de method de regel `offerteService.create(offerte);` door:

```

String offresURL = request.getRequestURL().toString().replace("toevoegen", "");
offerteService.create(offerte, offresURL);

```

Je wijzigt in OfferteService de method declaratie create:

```
void create(Offerte offerte, String offresURL);
```

Je wijzigt in DefaultOfferteService de method create:

```

@Override
@Transactional(readonly = false, isolation = Isolation.READ_COMMITTED)
public void create(Offerte offerte, String offresURL) {
    offerteRepository.save(offerte);
    mailSender.nieuweOfferte(offerte, offresURL);
}

```

Je wijzigt in MailSender de method declaratie nieuweOfferte:

```
void nieuweOfferte(Offerte offerte, String offertesURL);
```

Je wijzigt in DefaultMailSender de method nieuweOfferte:

```
@Override
public void nieuweOfferte(Offerte offerte, String offertesURL) {
    try {
        MimeMessage message = sender.createMimeMessage();
        MimeMessageHelper helper = new MimeMessageHelper(message);
        helper.setTo(offerte.getEmailAdres());
        helper.setSubject("Nieuwe offerte");
        String offerteURL = offertesURL + offerte.getId();
        helper.setText("Uw offerte heeft het nummer <strong>" + offerte.getId() +
            "</strong>. Je vindt de offerte <a href='" + offerteURL +
            "'>hier</a>.", true);
        sender.send(message);
    } catch (MessagingException | MailException ex) {
        LOGGER.error("Kan mail nieuwe offerte niet versturen", ex);
        throw new KanMailNietZendenException();
    }
}
```

(1) MimeMessage stelt een email met HTML opmaak voor.

(2) MimeMessageHelper helpt je de eigenschappen van de email in te stellen.

(3) Je maakt de URL van de nieuwe offerte.

Dit de URL [localhost:8080/offertes/](http://localhost:8080/offertes/) waaraan je het offertenummer toevoegt.

(4) Je gebruikt HTML tags (<strong>, <a>, ...). Je moet de 2° parameter dan op true plaatsen.

Je maakt in OfferteController een method die de request verwerkt als de gebruiker op de hyperlink in de email klikt:

```
private static final String OFFERTE_VIEW = "offertes/offerte";
@GetMapping("{offerte}")
ModelAndView read(@PathVariable Offerte offerte) {
    return new ModelAndView(OFFERTE_VIEW).addObject(offerte);
}
```

Je maakt offerte.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Offerte')"></head>
<body>
<nav th:replace="fragments::menu"></nav>
<h1 th:text="Offerte ${offerte.id}|"></h1>
<dl>
<dt>Voornaam</dt><dd th:text='${offerte.voornaam}'></dd>
<dt>Familiennaam</dt><dd th:text='${offerte.familiennaam}'></dd>
<dt>Email adres</dt><dd th:text='${offerte.emailAdres}'></dd>
<dt>Oppervlakte</dt><dd th:text='${offerte.oppervlakte}'></dd>
<dt>Aangemaakt</dt><dd th:text='${offerte.aangemaakt}'></dd>
</dl>
</body>
</html>
```

Je kan de website uitproberen.



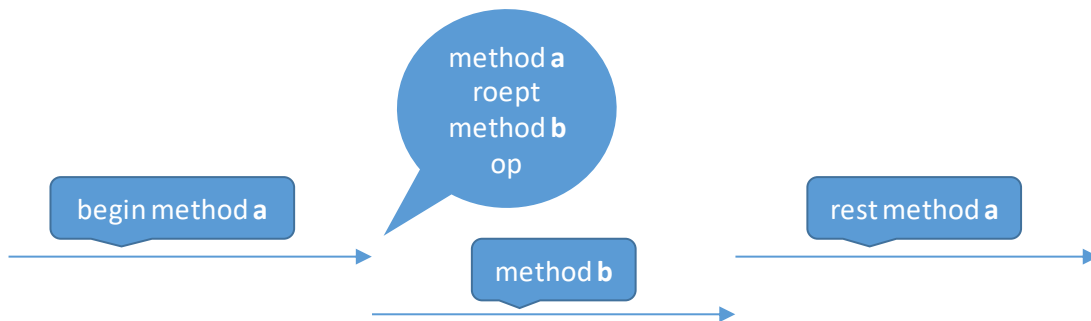
Je commit de sources. Je publiceert op GitHub.



Mail: zie takenbundel

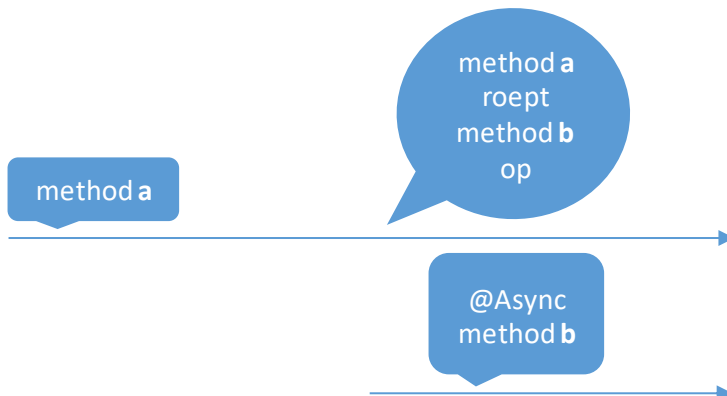
## 14 ASYNCHRONE VERWERKING

Als je een bean method oproept, voert Java die method synchroon uit.  
Dit betekent: als je in een method **a** een bean method **b** oproept,  
wacht Java tot de method **b** uitgevoerd is, vooraleer de code van method **a** verder uit te voeren:



### 14.1 @Async

Als je voor de bean method **b** `@Async` tikt, voert Spring de method **b** asynchroon uit.  
Dit betekent: als je in een method **a** de `@Async` bean method **b** oproept,  
voert Spring de code van method **b** uit in een aparte thread.  
Spring voert tegelijk de code van de method **a** verder uit in de thread die hoort bij **a**.  
Je verkrijgt dus multithreading.  
Dit werkt bij Spring enkel als de method **b** behoort tot een andere class dan de method **a**



### 14.2 @EnableAsync

`@Async` werkt als je voor de class `GroenetenenApplicatie` (of voor een class voorzien van `@Configuration`) `@EnableAsync` tikt.

### 14.3 Voorbeeld

Je verstuurt de mail (trage operatie) in een aparte thread met `@Async`.  
De oorspronkelijke thread stuurt intussen een response naar de browser.  
Je tikt `@Async` voor de `DefaultMailSender` method `nieuweOfferte`.  
Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



## 15 TERUGKERENDE TAKEN

Je applicatie moet soms terugkerende taken uitvoeren die de gebruiker niet start, maar automatisch starten. Voorbeelden:

- De applicatie verwijdt om het uur tijdelijke bestanden.
- De applicatie stuurt op de 1<sup>o</sup> dag van elke maand een mail met een rapport.



### 15.1 @Scheduled

Spring voert een bean method automatisch uit op terugkerende momenten als je voor die method `@Scheduled` schrijft.

`@Scheduled` heeft meerdere parameters waarmee je het terugkerend moment aangeeft:

- `@Scheduled(fixedDelay = 5000)`  
Spring voert de method uit. Nadat de method volledig uitgevoerd is, wacht Spring 5000 miliseconden en voert de method opnieuw uit ...
- `@Scheduled(fixedRate = 5000)`  
Spring voert de method om de 5000 miliseconden uit.
- `@Scheduled(cron = "0 0/1 * 1/1 * ? *")`  
Spring voert de method uit op 0 minuten, 0 uur, 1<sup>o</sup> dag van de maand van elke maand van elk jaar, ongeacht welke dag in de week dit is. De string is een cron expressie (een expressie die een tijdstip bepaalt), zie <http://en.wikipedia.org/wiki/Cron>



Tip: De syntax van een cron expressie kan moeilijk zijn.

Op [www.cronmaker.com](http://www.cronmaker.com) stel je op menselijke manier een cron expressie op en zie je het resultaat als een cron expressie.

### 15.2 @EnableScheduling

`@Scheduled` werkt enkel als je voor de class `GroenetenenApplicatie` (of voor een class voorzien van `@Configuration`) `@EnableScheduling` tikt.

### 15.3 Voorbeeld

Je stuurt op de eerste dag van elke maand een mail met het aantal offertes naar de webmaster.

Je voegt een regel toe aan `application.properties` met het email adres van de webmaster:

`emailAdresWebMaster=tikHierEenEmailAdres`

Je maakt een method declaratie in `OfferteService`:

```
void aantalOffertesMail();
```

Je implementeert deze method in `DefaultOfferteService`:

```
@Override
@Scheduled(*cron = "0 0/1 * 1/1 * ? * "*/ fixedRate=60000)
// test = om de minuut
public void aantalOffertesMail() {
    mailSender.aantalOffertesMail(offerteRepository.count());
}
```

Je maakt een method declaratie in `MailSender`:

```
void aantalOffertesMail(long aantal);
```

Je maakt een variabele in `DefaultMailSender`:

```
private final String emailAdresWebMaster;
```

Je wijzigt de constructor:

```
DefaultMailSender(JavaMailSender sender,
    @Value("${emailAdresWebMaster}") String emailAdresWebMaster) {
    this.sender = sender;
    this.emailAdresWebMaster = emailAdresWebMaster;
}
```

Je maakt een method:

```
@Override public void aantalOffertesMail(long aantal) {  
    try {  
        MimeMessage message = sender.createMimeMessage();  
        MimeMessageHelper helper = new MimeMessageHelper(message);  
        helper.setTo(emailAdresWebMaster);  
        helper.setSubject("Aantal offertes");  
        helper.setText("Aantal offertes:<strong>" + aantal + "</strong>", true);  
        sender.send(message);  
    } catch (MessagingException | MailException ex) {  
        LOGGER.error("Kan mail aantal offertes niet versturen", ex);  
        throw new KanMailNietZendenException();  
    }  
}
```

Je kan de website uitproberen.

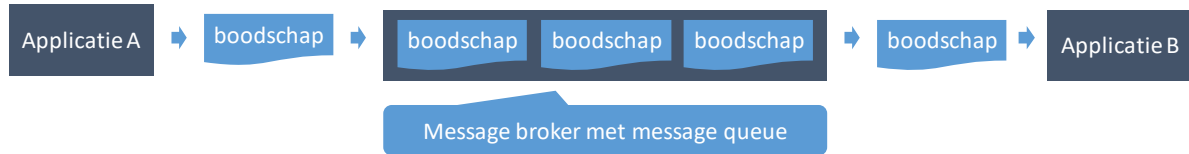


Je commit de sources. Je publiceert op GitHub.

## 16 JMS

Je gebruikt tot nu REST om twee applicaties te laten samenwerken. Je leert hier een alternatief: messaging. Een applicatie A wil de diensten oproepen van een applicatie B. De applicatie A stuurt daartoe een boodschap naar een wachtrij: een message queue. Aan de andere kant van de message queue leest applicatie B de boodschappen en verwerkt deze.

De message queue draait op een message broker: een applicatie die de message queue beheert.



Voordelen van messaging:

- De communicatie verloopt asynchroon.  
Applicatie A gaat na het afleveren van de boodschap aan de message broker verder met zijn werk en wacht niet tot applicatie B de boodschap ontvangen en verwerkt heeft. Bij REST verloopt de communicatie synchroon.  
Applicatie A stuurt een request naar applicatie B en wacht met verder te gaan met zijn werk tot hij een response van applicatie B terugkrijgt.
- Applicatie B moet niet noodzakelijk online zijn opdat applicatie A zou kunnen werken. Terwijl applicatie B offline is, kan applicatie A boodschappen blijven sturen naar de message broker. Zodra applicatie B terug online is, verwerkt hij deze boodschappen.
- Applicatie A kan tijdelijk sneller boodschappen versturen dan applicatie B deze kan verwerken. De message broker houdt deze boodschappen bij in de message queue.

Applicatie A en applicatie B zijn niet altijd in dezelfde programmeertaal geprogrammeerd.

Om mekaars boodschappen te kunnen verwerken is het belangrijk dat de boodschappen een dataformaat gebruiken dat alle programmeertalen kennen: XML of JSON.

Er bestaan verschillende message brokers: ActiveMQ, RabbitMQ, IBM MQ, .... Deze hebben allen een andere manier van aanspreken. JMS (Java Messaging System) laat toe al deze message brokers op eenzelfde manier aan te spreken vanuit je applicatie (zoals JDBC toelaat databases van verschillende merken op eenzelfde manier aan te spreken).

### 16.1 ActiveMQ

ActiveMQ is een populaire en open source message broker.

Je kan ActiveMQ downloaden via de pagina <http://activemq.apache.org/download.html>.

Je downloadt het ZIP bestand en je extraheert het ZIP bestand in de root van je computer.

Je opent een command prompt en je plaatst je in de bin folder van ActiveMQ met de opdracht `cd /naamVanDeActiveMQFolder/bin`. Je start ActiveMQ met de opdracht `activemq start`.

Je laat ActiveMQ draaien en je surft met een browser naar het beheerscherm van ActiveMQ:

<http://localhost:8161/admin>. Je tikt als gebruikersnaam admin en als paswoord admin.

Je maakt een queue met de naam `nieuweOfferteQueue`:

1. Je kiest de hyperlink Queues.
2. Je tikt `nieuweOfferteQueue` bij Queue Name.
3. Je kiest Create.

## 16.2 Boodschappen versturen

Je wijzigt de applicatie. Als een nieuwe offerte wordt toegevoegd, stuurt je applicatie een boodschap naar de message broker. Een andere applicatie luistert naar de message broker en verwerkt deze boodschap.

Je voegt een regel toe aan `applications.properties` met de URL van de message broker:

```
spring.activemq.broker-url=tcp://localhost:61616
```

❶

- (1) Je applicatie spreekt de message broker aan met het TCP/IP protocol.  
De message broker gebruikt TCP/IP poort 61616 om te communiceren met je applicatie.

Je voegt ook een regel toe met de naam van de queue die je gebruikt, omdat deze naam nog kan wijzigen nadat het programma af is:

```
nieuweOfferteQueue=nieuweOfferteQueue
```

Je maakt een package `be.vdab.groenetenen.messaging`.

Je maakt daarin de class `OfferteEnOffertesURL`:

```
package be.vdab.groenetenen.messaging;
// enkele imports
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class OfferteEnOffertesURL {
    private Offerte offerte;
    private String offertesURL;
    // je maakt getters en een geparametriseerde constructor
    // je maakt ook een protected default constructor die JAXB nodig heeft
}
```

❶

❷

- (1) Je zal boodschappen met objecten van de huidige class naar de message broker sturen.  
Spring moet die objecten eerst omzetten naar XML. Spring gebruikt hierbij JAXB.  
JAXB stelt als voorwaarde dat er `@XmlRootElement` staat voor een class waarvan de objecten naar XML worden omgezet.
- (2) De class bevat geen setters. Met deze regel benadert JAXB de private variabelen rechtstreeks, zonder tussenkomst van getters en setters.

Je tikt voor de class `Offerte`:

```
@XmlAccessorType(XmlAccessType.FIELD)
```

❶

- (1) De class bevat geen setter voor `id`. Met deze regel benadert JAXB de private variabelen rechtstreeks, zonder tussenkomst van getters en setters.

Je tikt voor de private variabele aangemaakt:

```
@XmlJavaTypeAdapter(value = LocalDateAdapter.class)
```

❶

- (1) Standaard kan JAXB geen `LocalDate` converteren van/naar een `String`.  
Je verwijst naar de class (die je ook al gebruikte bij REST) die helpt deze conversie te doen.  
`private LocalDate aangemaakt = LocalDate.now();`

Je voegt een dependency toe aan `pom.xml`. De library van deze dependency bevat functionaliteit om Java objecten te converteren naar XML en XML te converteren naar Java objecten:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-oxm</artifactId>
</dependency>
```

Je maakt in de package `be.vdab.groenetenen.messaging` een class `MessagingConfig`:

```
package be.vdab.groenetenen.messaging;
// enkele imports
@Configuration
class MessagingConfig {
    @Bean
    Jaxb2Marshaller marshaller() {
        Jaxb2Marshaller marshaller = new Jaxb2Marshaller();
        marshaller.setClassesToBeBound(OfferteEnOffertesURL.class);
        return marshaller;
    }
    @Bean
    MarshallingMessageConverter converter (
        Jaxb2Marshaller marshaller) {
        return new MarshallingMessageConverter(marshaller, marshaller);
    }
}
```

- (1) De class `Jaxb2Marshaller` converteert Java objecten naar XML en converteert XML naar Java objecten. De class gebruikt daartoe intern JAXB.
- (2) Je geeft de classes aan waarvan `Jaxb2Marshaller` objecten moet converteren van en naar XML.
- (3) Spring gebruikt een bean van de class `MarshallingMessageConverter` om een Java object, dat je naar de message broker stuurt, te converteren naar een bepaald data formaat.
- (4) Je ziet hier voor de eerste keer een method, voorafgegaan door `@Bean`, die een parameter heeft. Spring zoekt dan een bean van hetzelfde type en injecteert deze bean in die parameter. Je hebt daarjuist een bean van het type `Jaxb2Marshaller` gemaakt.
- (5) Je geeft als eerste parameter het object aan dat Spring moet gebruiken om Java objecten te converteren naar XML.  
Je geeft als tweede parameter het object aan dat Spring moet gebruiken om XML te converteren naar Java objecten.

Je wijzigt de class `DefaultOfferteService`. Je zal hier een message versturen naar de message broker als de gebruiker een nieuwe offerte maakt.

Je maakt private variabelen:

```
private final JmsTemplate jmsTemplate;
private final String nieuweOfferteQueue;
```

- (1) Je stuurt een message naar de message broker met de class `JmsTemplate`.  
Spring maakt bij de start van de website zelf een bean van deze class.
- (2) Deze variabele zal de naam bevatten van de queue waarnaar je messages stuurt.

Je wijzigt de constructor:

```
DefaultOfferteService(OfferteRepository offerteRepository,
    MailSender mailSender, JmsTemplate jmsTemplate,
    @Value("${nieuweOfferteQueue}") String nieuweOfferteQueue) {
    this.offerteRepository = offerteRepository;
    this.mailSender = mailSender;
    this.jmsTemplate = jmsTemplate;
    this.nieuweOfferteQueue = nieuweOfferteQueue;
}
```

Je wijzigt de method create:

```
@Override
@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
public void create(Offerte offerte, String offertesURL) {
    offerteRepository.save(offerte);
    OfferteEnOffertesURL offerteEnOffertesURL =
        new OfferteEnOffertesURL(offerte, offertesURL);
    jmsTemplate.convertAndSend(nieuweOfferteQueue, offerteEnOffertesURL);
}
```

- (1) Je maakt een object dat je zal opnemen in de message die je verstuurt.
- (2) Je converteert het OfferteEnOffertesURL object.  
Spring gebruikt daarbij de MarshallingMessageConverter bean die je definieerde.  
Spring converteert daarmee het object naar XML.  
Je verstuurt daarna deze XML naar de message broker queue nieuweOfferteQueue.

Je kan de website uitproberen. Iedere keer je een nieuw filiaal toevoegt, komt er een nieuwe message in de queue. Je kan dit zien op <http://localhost:8161/admin>:

Name	Number Of Pending Messages
------	----------------------------

nieuweOfferteQueue 1

Als je in deze pagina nieuweOfferteQueue aanklikt, zie je een lijst met de messages in de queue:

Message ID ↑	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID: Dualboot-20504-1513584875955-29:1:1:1		Persistent	4	false		2017-12-18 11:03:42:199 CET		Delete
ID: Dualboot-20504-1513584875955-29:2:1:1		Persistent	4	false		2017-12-18 11:06:29:346 CET		Delete

Als je één van de message id's aanklikt, zie je een pagina met de detail informatie over de message. Je ziet bij Message Details de inhoud van de message, in (niet opgemaakte) XML.

Er wordt geen email meer verstuurd naar het email adres in de nieuwe offerte.



Opmerking: als je een message zou sturen naar een queue die op dat moment nog niet bestaat, wordt deze queue automatisch aangemaakt.

## 16.3 Boodschappen ontvangen

Tot nu is er geen applicatie die messages ontvangt van de message broker.

De message broker houdt de messages bij in de queue tot er wel zo'n applicatie is.

Je zal nu zo'n applicatie maken. Normaal is dit een nieuwe applicatie, voor de eenvoud zal de huidige applicatie deze rol spelen. De applicatie zal per gelezen message een email sturen naar het email adres in de nieuwe offerte.

Als de applicatie een message leest, moet de inhoud van die message terug geconverteerd worden van XML naar een Java object. Je maakt daartoe een bean definitie in MessagingConfig:

```
@Bean
DefaultJmsListenerContainerFactory factory(
    ConnectionFactory connectionFactory,
    MarshallingMessageConverter converter) {
    DefaultJmsListenerContainerFactory factory =
        new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory);
    factory.setMessageConverter(converter);
    return factory;
}
```

- (1) De class DefaultJmsListenerContainerFactory is een toepassing van het factory design pattern. Spring gebruikt deze class om JmsListener objecten te maken. JmsListener objecten lezen messages uit de message broker.
- (2) Spring injecteert in deze parameter de bean van de class ConnectionFactory. Spring maakt deze bean zelf aan. De class ConnectionFactory is ook een toepassing van het factory design pattern. Spring gebruikt deze class om verbindingen met de message broker te maken.

- (3) Spring injecteert in deze parameter de bean van de class `MarshallingMessageConverter`. Je hebt deze bean vroeger in dit hoofdstuk gedefinieerd. Deze bean converteert Java objecten die je in een te versturen message plaatst naar XML en converteert XML in een te lezen message naar Java objecten.

Je maakt in de package `be.vdab.groenetenen.messaging` een class `NieuweOfferteListener`:

```
package be.vdab.groenetenen.messaging;
// enkele imports
@Component
class NieuweOfferteListener {
    private final MailSender mailSender;
    NieuweOfferteListener(MailSender mailSender) {
        this.mailSender = mailSender;
    }
    @JmsListener(destination = "${nieuweOfferteQueue}")
    void ontvangBoodschap(OfferteEnOffertesURL offerteEnOffertesURL) {
        mailSender.nieuweOfferte(offerteEnOffertesURL.getOfferte(),
            offerteEnOffertesURL.getOffertesURL());
    }
}
```

- (1) Je injecteert de bean waarmee je emails kan versturen.
- (2) Je tikt `@JmsListener` voor een method waarin je messages van de message broker wil ontvangen. Je duidt daarbij de queue aan waaruit je boodschappen wil ontvangen. Je tikt de naam van deze queue niet letterlijk (hard coded), maar je verwijst naar de instelling met de naam `nieuweOfferteQueue` in `application.properties`.
- (3) Je geeft de method een parameter van het type `OfferteEnOffertesURL`. Telkens de method een boodschap van de message broker ontvangt, converteert Spring de XML in deze boodschap naar een `OfferteEnOffertesURL` object en biedt het aan in deze parameter.
- (4) Je verwerkt de boodschap. Je stuurt een email naar het email adres in de offerte.

Je mag in de class `DefaultMailSender` `@Async` verwijderen bij de method `nieuweOfferte`: het verwerken van messages van een message broker gebeurt sowieso in een achtergrondtaak.

Je kan de website uitproberen.

De messages worden verwerkt en verdwijnen daarna uit de queue van de message broker:

Name	Number Of Pending Messages
<code>nieuweOfferteQueue</code>	0



Je commit de sources. Je publiceert op GitHub.



Opmerking: als de instelling `spring.activemq.broker-url` ontbreekt in `application.properties`, start Spring een embedded message broker (een message broker die ingebakken is in je website). Spring gebruikt die message broker dan bij het versturen en ontvangen van messages.



Opmerking: naast JMS bestaan andere producten waarmee een applicatie een boodschap naar een queue stuurt en een andere applicatie die boodschap uit de queue leest en verwerkt. Populaire producten zijn RabbitMQ en Apache Kafka.



JMS: zie takenbundel

## 17 CUSTOM ERROR PAGES

### 17.1 404

Een browser request kan een fout veroorzaken.

Het bekendste voorbeeld is een request naar een URL die niet bestaat in de website.

De webserver stuurt dan een response met een status code 404 (Not Found)

en een response body (HTML) die Spring zelf aanmaakt.

Je ziet de “lelijke” response als je surft naar <http://localhost:8080/xxx>.

Je kunt een eigen mooiere foutpagina associëren met een status code.

Als een fout met die status code optreedt, stuurt de webserver een response met die pagina.

Je maakt in de map templates een map error. De naam van deze map ligt vast.

Je maakt in deze map 404.html.

De naam van de pagina is de status code waarmee je de pagina wil associëren.

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Pagina niet gevonden')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>We hebben de gevraagde pagina niet gevonden</h1>
<img alt='Pagina niet gevonden' th:src='@{/images/404.jpg}'>
</body>
</html>
```

Je kan de website uitproberen.

### 17.2 500

Als een exception optreedt in je applicatie die je niet opvangt, stuurt de webserver een response met status code 500 (Internal Server Error) en een response body die Spring zelf aanmaakt.

Deze pagina bevat ook informatie over de opgetreden exception.

Je kan dit zien door ActiveMQ te stoppen en in de website een offerte aanvraag te maken.

Dit geeft problemen:

- ⊖ Een gewone gebruiker is overdonderd door de technische informatie in de pagina.
- ⊖ Een hacker leert de binnenkant van de website kennen. Hij weet nu dat je JMS gebruikt.

Om dit op te lossen maak je in de map error ook 500.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Probleem')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>Probleem</h1>
<p>We kunnen dit niet uitvoeren wegens een technische storing.<br>
Gelieve de helpdesk te contacteren.</p>
<img alt='Probleem' th:src='@{/images/500.jpg}'>
</body>
</html>
```

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



## 18 SPRING SECURITY

### 18.1 Security woordenschat

- **Principal**  
De gebruiker (of applicatie) die je applicatie wil gebruiken.
- **Authentication**
  - Identificatie van de principal.  
De principal geeft zijn identiteit aan via zijn userid.
  - Controle of de principal de identiteit heeft die hij beweert te hebben.  
De principal bewijst zijn identiteit door een bijbehorend paswoord te tikken.
- **Authority**  
Een rol die een principal speelt. Voorbeelden: manager, magazijnier.  
Er is een veel op veel relatie tussen authority en principal:
  - Meerdere principals kunnen eenzelfde authority hebben.  
De twee bazen van een firma (Joe en Jack) hebben beiden de authority manager.
  - Één principal kan meerdere authorities hebben.  
In de firma heeft Averell twee authorities: helpdeskmedewerker en magazijnier.

		Principals		
		Joe	Jack	Averell
Authorities	Manager	✓	✓	
	Helpdeskmedewerker			✓
	Magazijnier			✓

- **Authorization**  
Welke authorities mogen welke use case uitvoeren ? Voorbeelden:

Use case	Authorities die deze use case mogen uitvoeren
Stockopname	Magazijnier
Weddeverhoging	Manager
Personeelslijst	Manager, Helpdeskmedewerker

- **Access control**  
Wanneer een principal een use case wil uitvoeren, controleer je met authorization of hij die use case mag uitvoeren. Vb. Averell wil de use case Stockopname uitvoeren.  
Dit mag: Averell heeft de authority Magazijnier, die mag Stockopname uitvoeren.

### 18.2 Configuratie

#### 18.2.1 pom.xml

Je voegt de Spring security dependencies toe aan pom.xml :

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



Opmerking: als je een nieuw project maakt op [start.spring.io](https://start.spring.io), kan daar je bij de dependencies direct security toevoegen.

#### 18.2.2 Default security

Standaard is er één gebruiker met de gebruikersnaam user. Spring maakt voor die gebruiker een random paswoord bij de start van de applicatie. Je vindt dit paswoord in de boodschappen in het Eclipse venster Console. Het paswoord wordt voorafgegaan door de tekst  
Using generated security password:

Als je de website bezoekt met een browser, moet je de gebruikersnaam en het paswoord intikken voor je toegang krijgt tot de website. Eens correct ingelogd heb je standaard toegang tot alle onderdelen van de website.

### 18.2.3 Java config

Je maakt een package `be.vdab.groenetenen.security` en daarin een class `SecurityConfig`:

```
package be.vdab.groenetenen.security;
// enkele imports
@EnableWebSecurity
class SecurityConfig extends WebSecurityConfigurerAdapter {
    private static final String MANAGER = "manager";
    private static final String HELPDESKMEDEWERKER = "helpdeskmedewerker";
    private static final String MAGAZIJNIER = "magazijnier";
    @Bean
    InMemoryUserDetailsManager inMemoryUserDetailsManager() {
        return new InMemoryUserDetailsManager(
            User.builder().username("joe").password("{noop}theboss")
                .authorities(MANAGER).build(),
            User.builder().username("averell").password("{noop}hungry")
                .authorities(HELPDESKMEDEWERKER, MAGAZIJNIER).build());
    }
    @Override
    public void configure(WebSecurity web) throws Exception {
        web.ignoring()
            .mvcMatchers("/images/**")
            .mvcMatchers("/css/**")
            .mvcMatchers("/scripts/**");
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.formLogin()
            .and().authorizeRequests()
            .mvcMatchers("/offertes/toevoegen").hasAuthority(MANAGER)
            .mvcMatchers("/werknemers")
                .hasAnyAuthority(MAGAZIJNIER, HELPDESKMEDEWERKER);
    }
}
```

- (1) `@EnableWebSecurity` integreert Spring security en Spring MVC.
- (2) Je erft van de class `WebSecurityConfigurerAdapter`.  
Je kan in je class methods overriden om Spring security te configureren.
- (3) Je maakt een bean van het type `InMemoryUserDetailsManager`. Je houdt met deze bean principals bij in het interne geheugen. Je leest ze verder in de cursus uit een database.  
Zodra je zo'n bean maakt, maakt Spring zelf geen gebruiker met de naam user meer aan.
- (4) Je geeft aan de `InMemoryUserDetailsManager` constructor enkele Users (principals) mee.  
Je maakt een User met het builder design pattern. Je vermeldt de gebruikersnaam, het paswoord en de authorities. Voor het paswoord staat `{noop}`.  
Dit betekent Spring het paswoord niet-versleuteld onthoudt.  
Je leert verder in de cursus werken met versleutelde paswoorden.
- (5) Je overridet de method die de web eigenschappen van Spring security configureert.
- (6) Spring security moet geen beveiliging doen op URL's die passen bij `/images/**`.  
`**` betekent dat het patroon ook subfolders van `/images` bevat.
- (7) Je overridet de method die de HTTP beveiliging van Spring security configureert.
- (8) De gebruiker authenticceert zich door zijn gebruikersnaam en paswoord te tikken in een HTML form.
- (9) Je definieert autorisatie: enkel ingelogde gebruikers met de authority manager kunnen de URL `/offertes/toevoegen` aanspreken.

- (10) Enkel ingelogde gebruikers met de authority magazijnier of helpdeskmedewerker kunnen de URL /werknemers aanspreken.

Je kan de applicatie uitproberen.

Als je de eerste keer Offertes, toevoegen of Werknemers kiest, zie je een inlogpagina.

- Je logt bijvoorbeeld in met de gebruiker joe en het paswoord theboss.
- Als je een onbestaande userid of een verkeerd paswoord tikt zie je de pagina opnieuw.
- Als je een bestaande userid en een correct paswoord tikt zie je
  - de gewenste pagina als de principal met die userid de URL mag aanspreken.
  - een pagina met fout 403 (forbidden) als de principal de URL niet mag aanspreken.

Op sommige websites kan een anonieme gebruiker slechts enkele pagina's zien (bvb. de welkompagina en de inlogpagina). Hij moet minstens ingelogd zijn om andere pagina's te zien.

Je vervangt in de SecurityConfig method configure(HttpSecurity http) de ; door:

```
.mvcMatchers("/", "/login").permitAll()           ❶
.mvcMatchers("/**").authenticated();              ❷
```

- (1) Je geeft alle (ook anonieme) gebruikers toegang tot de welkompagina en de loginpagina.
- (2) Voor alle andere URL's moet de gebruiker minstens ingelogd zijn.

De volgorde waarmee je meerdere keren de method mvcMatchers oproept is belangrijk.

Je moet eerst de oproepen doen met de meest specifieke URL patronen (die zonder wildcards).

Je doet daarna de oproepen met de meer algemene URL patronen (die met wildcards).

Als Spring Security een browser request verwerkt, overloopt hij de URL patronen in de volgorde zoals je ze met mvcMatchers() toegevoegd hebt. Hij gebruikt het eerste URL patroon dat bij de browser request past en slaat de daaropvolgende URL patronen over.

Als je bovenstaande code zou schrijven als

```
.mvcMatchers("/**").authenticated();
.mvcMatchers("/", "/login").permitAll();
```

zou een browser request naar /login horen bij het URL patroon /\*\*

en zouden verkeerdelijk enkel reeds ingelogde gebruikers de login pagina kunnen bereiken!

Je kan de applicatie uitproberen.

### 18.3 Eigen 403 (forbidden) pagina

Je maakt een eigen pagina die hoort bij de fout 403 (forbidden).

Je maakt in de map error 403.html:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace='fragments::head(title='Verboden')'></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>U hebt geen toegang tot dit onderdeel.</h1>
<img alt='Verboden' th:src='@{/images/403.jpg}'>
</body>
</html>
```

Je kan de applicatie uitproberen.

### 18.4 CSRF



CSRF (Cross-site request forgery) is een hacker techniek. Een argeloze gebruiker stuurt via een website van een hacker onbewust requests naar een andere website waarop hij ingelogd is. Spring security heeft beveiliging tegen CSRF.

Je leert CSRF kennen en daarna hoe Spring security je website beveiligt tegen CSRF.

Je breidt het programma uit met een use case "afschrijven".

Deze wijzigt de gebouwwaarde van een filiaal naar 0.

Je zal deze use case later nodig hebben bij de uitleg over CSRF.

Je maakt een method in Filiaal:

```
public void afschrijven() {
    waardeGebouw = BigDecimal.ZERO;
}
```

Je maakt een method declaratie in FiliaalService:

```
void afschrijven(long id);
```

Je implementeert deze method in DefaultFiliaalService:

```
@Override
@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)
public void afschrijven(long id) {
    filiaalRepository.findById(id).ifPresent(filiaal -> filiaal.afschrijven());
}
```

Je maakt een method in FiliaalController:

```
private static final String REDIRECT_NA_AFSCHRIJVEN = "redirect:/filialen/{id}";
@PostMapping("/{id}/afschrijven")
String afschrijven(@PathVariable long id, RedirectAttributes redirectAttributes){
    filiaalService.afschrijven(id);
    redirectAttributes.addAttribute("id", id);
    return REDIRECT_NA_AFSCHRIJVEN;
}
```

- (1) De URL, waarnaar de redirect gebeurt, bevat een path variabele id.  
Je vervangt deze hier door de id van het huidig filiaal.

Je voegt regels toe aan filiaal.html, voor </body>:

```
<form th:action="@{/filialen/{id}/afschrijven(id=${filiaal.id})}" method='post'>
<input type='submit' value='Afschrijven'>
</form>
```

Je kan de website uitproberen.

CSRF beveiliging is standaard actief in Spring security. Je zet die af om CSRF aan het werk te zien.

Je tikt in de SecurityConfig method configure(HttpSecurity http) juist na  
http.csrf().disable()

Je simuleert nu stap per stap de CSRF hack:

- Je start de website en logt in als joe.
- Als je de website opent op een tweede tabblad, moet je niet meer inloggen: toen je inlogde op het eerste tabblad heeft Spring security je identiteit onthouden in een HTTP session attribuut. Je deelt je HTTP session over alle tabbladen van je browser.
- Je opent in een derde tabblad de website van de hacker. Je opent daartoe tombola.htm (materiaal bij de cursus) met de sneltoets Ctrl+O. De body van deze pagina is:

```
<form method='post' action='http://localhost:8080/filialen/1/afschrijven'>
  <input type='submit' value='Klik hier om een mooie prijs te winnen'>
</form>
```

Als je op de submit knop klikt, stuur je onbewust een POST request naar de website waarop je (via het eerste tabblad) ingelogd bent. Ook op het derde tabblad moet je niet meer inloggen. In dit voorbeeld schrijft de hacker een filiaal af, maar hij kan met deze techniek in een bank applicatie ook geld overschrijven, ...

Je activeert terug de CSRF beveiliging van Spring security. Je verwijdert in de SecurityConfig method configure(HttpSecurity http) de expressie .csrf().disable()

De beveiliging werkt als volgt:

- Als een gebruiker inlogt maakt Spring security een session attribuut met de naam \_csrf en als inhoud een random string.
- Spring security voegt aan elke HTML form, die je met Thymeleaf maakt, en als method POST heeft, een verborgen veld \_csrf toe. De inhoud van dit veld is gelijk aan het session attribuut \_csrf.

- Als de gebruiker de form submit, verstuurt hij ook de inhoud van dit verborgen veld. Spring security controleert of dit verborgen veld meegestuurd is en of de inhoud gelijk is aan het session attribuut `_csrf`. Als dit zo is, wordt de request normaal verwerkt. Als dit niet zo is stuurt Spring security de 'forbidden' pagina naar de browser.

Je ziet het verborgen veld dat Spring security toevoegt aan een HTML form als je de detailpagina van een filiaal opent in de browser en rechtermuisklik, view source vraagt.

Een hacker kan niet raden welke random string hij in het verborgen veld `_csrf` moet opnemen: deze string wijzigt bij elke gebruikerslogin.

Als je nu terug de stappen uitvoert om CSRF te simuleren, zie je de 'forbidden' pagina.

## 18.5 Eigen inlogpagina

Je vervangt de ingebouwde inlogpagina door een eigen inlogpagina. De naam is vrij te kiezen. Je maakt in templates `login.html`:

```
<!doctype html>
<html lang='nl' xmlns:th='http://www.thymeleaf.org'>
<head th:replace="fragments::head(title='Inloggen')"></head>
<body>
<nav th:replace='fragments::menu'></nav>
<h1>Inloggen</h1>
<form th:action="@{/Login}" method='post'>
<label>Gebruikersnaam:<input name='username' required autofocus></label>
<label>Paswoord:<input name='password' type='password' required></label>
<input type='submit' value='Inloggen'>
</form>
</body>
</html>
```

- (1) Je submit de form met een POST request naar dezelfde URL waarop je straks de form toont. Spring security verwerkt deze POST request en doet authenticatie.
- (2) De name van het invoervak voor de userid moet username zijn.
- (3) De name van het invoervak voor het paswoord moet password zijn.

Je maakt `LoginController`:

```
package be.vdab.groenetenen.web;
// enkele imports
@Controller
@RequestMapping("/login")
class LoginController {
    private static final String VIEW="login";
    @GetMapping
    String login() {
        return VIEW;
    }
}
```

Je definieert deze URL als inlogpagina in de `SecurityConfig` method `configure(HttpSecurity http)`. Je tikt `.loginPage("/login")` na de expressie `formLogin()`.

Je kan de applicatie uitproberen.

## 18.6 Login fouten

Als de gebruiker in de inlogpagina een verkeerde userid of paswoord tikt, doet Spring security een redirect terug naar die inlogpagina en geeft daarbij een parameter `error` mee.

Je toont in dat geval een foutmelding. Je breidt `login.html` uit, na `</form>`:

```
<div th:if='${param.error != null}' class='fout'>Verkeerde gebruikersnaam of
paswoord</div>
```

Je kan de website uitproberen.

## 18.7 Expliciet inloggen en uitloggen

Je voegt in `fragments.html` een hyperlink naar de inlogpagina, voor `</ul></nav>`:

```
<li><a th:href="@{/Login}">Aanmelden</a></li>
```

Spring security houdt de identiteit van de principal bij in een `HttpSession` variabele.

De principal behoudt dus zijn identiteit zolang die `HttpSession` variabele bestaat.

De principal kan zijn identiteit sneller laten vergeten, als je een uitlog knop aan je applicatie toevoegt. Een klik op de button verwijdert de `HttpSession` variabele.

Je activeert de uitlog functionaliteit in de `SecurityConfig` method

```
configure(HttpSecurity http). Je tikt code na de expressie loginPage("/login").
.logout()
```

Je wordt uitgelogd bij een HTTP POST request naar de URL `/logout`.

Je voegt in `fragments.html` een menu puntje, voor `</ul></nav>`:

```
<li>
<form method='post' th:action="@{/Logout}" id='Logoutform'>
<input type='submit' value='Afmelden' id='Logoutbutton'>
</form>
</li>
```

Je kan de website uitproberen.

Spring security toont na het afmelden standaard de inlogpagina.

Je kan dit bijvoorbeeld wijzigen naar de welkompagina.

```
Je tikt in de SecurityConfig method configure(HttpSecurity http) na de expressie logout()
.logoutSuccessUrl("/")
```

Je kan de website uitproberen.

## 18.8 Security in Thymeleaf

Thymeleaf bevat functionaliteit waarmee je security informatie kan lezen.

### 18.8.1 pom.xml

Je voegt een dependency toe:

```
<dependency>
  <groupId>org.thymeleaf.extras</groupId>
  <artifactId>thymeleaf-extras-springsecurity4</artifactId>
</dependency>
```

### 18.8.2 authorize tag

Je toont met `sec:authorize` een stuk HTML conditioneel, afhankelijk van de authorities van de huidige gebruiker. Je bouwt de conditie op, met één van volgende Spring security SpEL functies:

- `isAnonymous()`  
geeft true terug als de gebruiker niet ingelogd is.
- `isAuthenticated()`  
geeft true terug als de gebruiker ingelogd is.
- `hasAuthority('manager')`  
geeft true terug als de gebruiker ingelogd is en de authority manager heeft.
- `hasAnyAuthority('helpdeskmedewerker', 'magazijnier')`  
geeft true terug als de gebruiker ingelogd is en de authority helpdeskmedewerker of magazijnier heeft.

Voorbeelden van de `authorize` tag:

```
<div sec:authorize="hasAuthority('manager')">
```

Enkel gebruikers met de role manager zien deze tekst

```
</div>
```

```
<div sec:authorize='isAuthenticated()'>
```

Enkel ingelogde gebruikers zien deze tekst

```
</div>
```

Je gebruikt dit in `fragments.html`:

- Je voegt `xmlns:sec='http://www.thymeleaf.org/extras/spring-security'` toe aan de `<html>` tag.
- Je toont de hyperlink Aanmelden enkel aan niet-ingelogde gebruikers.  
Je voegt `sec:authorize='isAnonymous()'` toe aan de `<li>` van het aanmelden.
- Je toont de hyperlink Afmelden enkel aan ingelogde gebruikers.  
Je voegt `sec:authorize='isAuthenticated()'` toe aan de `<li>` van het aanmelden.

Je kan de website uitproberen.

Je toont met `sec:authorize-url` een stuk HTML enkel als de gebruiker toegang heeft tot de URL die je in dit attribuut vermeldt.

Je toont Werknemers → Lijst enkel als de gebruiker toegang heeft tot `/werknemers`.

Je voegt `sec:authorize-url='/werknemers'` toe aan de `<li>` van de werknemerslijst.

Je kan de website uitproberen.

### 18.8.3 Naam van de huidige gebruiker

Je toont met `sec:authentication='name'` de naam van de ingelogde gebruiker.

Je gebruikt dit in `fragments.xml`. Je voegt code toe voor `<input type='submit' ...>`:

```
<span sec:authentication='name'></span>
```

Je kan de website uitproberen.



Als je in een controller method die requests verwerkt de naam van de huidige gebruiker wil weten, voeg je aan die method een parameter toe van het type `Principal`. Je roept op die parameter de method `getName` op.  
Je krijgt een `String` met de naam van de huidige gebruiker.

## 18.9 Principals en authorization in een database

### 18.9.1 Password encoding



Het is sterk afgeraden om paswoorden letterlijk op te nemen in een database. Als een hacker de database te pakken krijgt, weet hij alle gebruikersnamen en paswoorden. Hij kan inloggen alsof hij één van de geregistreerde gebruikers is.

Je vermijdt dit door de paswoorden in geëncrypteerde (versleutelde) vorm te bewaren in de database (password encoding).

Er bestaan algoritmes (MD4, MD5, SHA, bcrypt ...) die one-way encryptie doen. Dit betekent dat je

- een leesbaar paswoord kan omzetten in geëncrypteerde vorm.
- een geëncrypteerd paswoord niet kan omzetten naar de leesbare vorm.

Één van de beste algoritmes is bcrypt. Dit algoritme encrypteert bijvoorbeeld het leesbaar paswoord `theboss` naar `$2a$10$3DPuiwz0.I2UYgge1Be8NuCHdd7Jb1z2cu8K0ZkkguQZYnCIA4u50`

Als een hacker de database met geëncrypteerde paswoorden steelt heb je geen probleem.

Hij kan `$2a$10$3DPuiwz0.I2UYgge1Be8NuCHdd7Jb1z2cu8K0ZkkguQZYnCIA4u50` niet terugvormen naar `theboss` en kan dus niet als joe inloggen op de website.

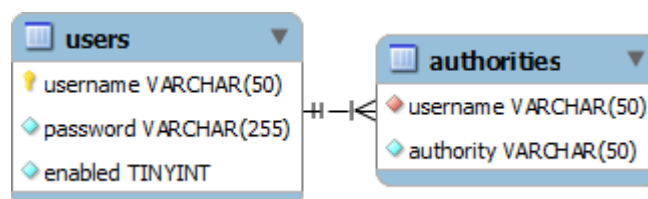
Je kan met volgende code fragment de bcrypt geëncrypteerde versie van een paswoord krijgen:

```
String geencrypteerd = new BCryptPasswordEncoder().encode(origineelPaswoord);
```

### 18.9.2 Default tabel structuren

Je onthoudt de principals en hun authorities in een database.

Spring security verwacht deze informatie standaard in twee tables met volgende structuur:





- De database groenetenen bevat deze twee tables:  
users bevat records voor joe en voor averell, authorities bevat hun authorities.
- De kolom password in de table users bevat het geëncrypteerde paswoord (theboss voor Joe en hungry voor Averell). Spring security verwacht dat dit paswoord wordt voorafgegaan door de naam van het encryptie algoritme tussen accolades {bcrypt}.
- De kolom enabled in de table users bevat 1 (true) of 0 (false). Als de kolom false bevat, kan je met de bijbehorende principal niet inloggen. Je kan zo een principal tijdelijk uitschakelen.

Je vervangt de method `inMemoryUserDetailsManager` in `SecurityConfig`:

@Bean

```
JdbcDaoImpl jdbcDaoImpl(DataSource dataSource) {
    JdbcDaoImpl impl = new JdbcDaoImpl();
    impl.setDataSource(dataSource);
    return impl;
}
```

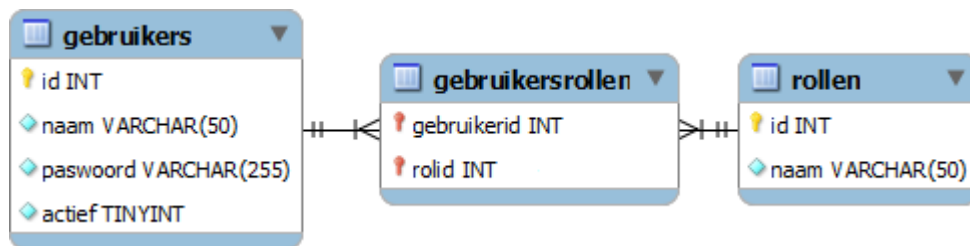
❶

- (1) De class `JdbcDaoImpl` leest principals uit de database. Deze class moet weten uit welke database hij de principals moet lezen. Je injecteert daartoe de `DataSource` die al naar de database groenetenen verwijst in de huidige method.

Je kan de website uitproberen.

### 18.9.3 Afwijkende tabel structuren

De database groenetenen bevat de tables gebruikers, rollen en gebruikersrollen. Deze bevatten ook informatie over de principals joe en averell, maar hebben een andere structuur dan de default structuur van Spring security.



Spring security kan met tables samenwerken die niet de default structuur hebben, als je Spring security twee SQL statements aanbiedt:

- Een SQL statement dat één gebruiker leest aan de hand van een gebruikersnaam. Dit statement geeft maximaal 1 rij terug en moet 3 kolommen teruggeven:
  - username gebruikersnaam
  - password bijbehorende paswoord
  - enabled is de gebruiker bruikbaar of uitgeschakeld

Je maakt een variabele met zo'n SQL statement in `SecurityConfig`:

```
private static final String USERS_BY_USERNAME =
    "select naam as username, paswoord as password, actief as enabled" +
    "from gebruikers where naam = ?";
```

- Een SQL statement dat de authorities van één gebruiker leest aan de hand van een gebruikersnaam. Dit statement geeft X rijen terug met 2 kolommen:
  - username gebruikersnaam
  - authorities rol

Je maakt een variabele met zo'n SQL statement in `SecurityConfig`:

```
private static final String AUTHORITIES_BY_USERNAME =
    "select gebruikers.naam as username, rollen.naam as authorities" +
    "from gebruikers inner join gebruikersrollen" +
    "on gebruikers.id = gebruikersrollen.gebruikerid" +
    "inner join rollen" +
    "on rollen.id = gebruikersrollen.rolid" +
    "where gebruikers.naam= ?";
```



Je gebruikt deze variabelen in de method `jdbcTemplate`, voor het return statement:

```
impl.setUsersByUsernameQuery(USERS_BY_USERNAME);
impl.setAuthoritiesByUsernameQuery(AUTHORITIES_BY_USERNAME);
```

Je kan de website uitproberen.

## 18.10 Services layer

Je kan security ook toepassen in de services layer:

- Als je een use case via meerdere URL's kan starten, is het interessanter de beveiliging één keer te definiëren op de services layer method die bij de use case hoort, dan de beveiliging te herhalen op al die URL's.
- Sommige ontwikkelaars doen per use case access control in de presentation layer én nog eens in de services layer, om de beveiliging te maximaliseren.

### 18.10.1 @PreAuthorize activeren

Je beveiligt een method van de service class met `@PreAuthorize`.

Standaard staat het gebruik van `@PreAuthorize` af.

Je activeert dit door voor de class `SecurityConfig` een annotation te schrijven:

```
@EnableGlobalMethodSecurity(prePostEnabled = true)
```

### 18.10.2 @PreAuthorize

Je geeft aan `@PreAuthorize` een Spring security SpEL expressie mee.

Spring security voert de method, waarvoor je `@PreAuthorize` schrijft,

enkel uit als deze SpEL expressie `true` teruggeeft.

Voorbeelden

- `@PreAuthorize("hasAuthority('manager')")`  
`public void beveiligdeMethod()`  
Enkel gebruikers met de authority `manager` kunnen `beveiligdeMethod` uitvoeren.
- `@PreAuthorize("hasAnyAuthority('helpdeskmedewerker', 'magazijnier')")`  
`public void beveiligdeMethod()`  
Enkel gebruikers met de authority `helpdeskmedewerker` of `magazijnier` kunnen `beveiligdeMethod` uitvoeren.

Je kan `@PreAuthorize` schrijven

- Voor een class `@PreAuthorize` geldt dan voor alle methods van die class.
- Voor een method `@PreAuthorize` geldt dan enkel voor die method.

Als je bij een class én bij een method `@PreAuthorize` tikt,

overschrijft `@PreAuthorize` bij de method `@PreAuthorize` bij de class.

Je schrijft voor de `DefaultFiliaalService` method `findByPostcode`

```
@PreAuthorize("hasAuthority('manager')")
```

Je start de website en meldt je aan als `averell`.

Je kiest Filialen, Van tot postcode. Nadat je een correcte waarde intikt bij Van en Tot en op Zoeken klikt, roept Spring de beveiligde method `findByPostcode` op.

Gezien `averell` de rol `manager` niet heeft, voert Spring de method niet uit en zie je `403.html`.

## 18.11 Niet-browser REST clients

### 18.11.1 Authenticatie

Een gebruiker authenticceert zich met een browser door zijn gebruikersnaam en paswoord te tikken in een HTML form in een inlogpagina van de website.

Als een gebruiker een REST service wil aanspreken met een niet-browser REST client is die manier van authenticatie niet toepasbaar. Als je bijvoorbeeld Postman gebruikt, zal je je gebruikersnaam en paswoord tikken in een dialogvenster (dat geen HTML form is).

De gebruikte authenticatie heet *basic authentication*. Daarbij geeft de client bij elke request naar de REST service de gebruikersnaam en paswoord mee in de request header `authentication`.

Je activeert in de website basic authentication als een tweede authenticatie manier (naast form authenticatie). Je voegt daartoe een opdracht achteraan toe aan de 3<sup>e</sup> method configure van de class SecurityConfig.

```
http.httpBasic();
```

Je maakt nu met Postman een GET request naar de beveiligde url /filialen:

1. Je kiest GET.
2. Je als URL `http://localhost:8080/filialen`
3. Je kiest in Authorization onder Type voor Basic Auth.
4. Je tikt joe bij Username en theboss bij Password.
5. Je verstuurt de request .  
Postman geeft dan in de request de gebruikersnaam en het paswoord mee in de request header authentication.

Je moet ook in FiliaalRestControllerTest de request header authentication meegeven om de tests te doen lukken.

```
Je voegt in filiaalLezenDatNietBestaat code toe na mvc.perform(get("/filialen/-1")  
.header(HttpHeaders.AUTHORIZATION,  
"Basic " + Base64Utils.encodeToString("joe:theboss".getBytes()))
```

 ❶

- (1) Je plaatst de inhoud van de request hreader authentication op Basic, gevolg door een spatie, gevolgd door een gebruikersnaam, gevolgd door :, gevolgd door het paswoord. Je moet de gebruikersnaam en het paswoord meegeven in Base64 encoding. Deze encoding zet bytes om in ASCII tekens, zoals Basic authentication het voorschrijft.

Je voegt hetzelfde stuk code ook toe in de methods filiaalDatBestaatLezenInXMLFormaat en filiaalDatBestaatLezenInJSONFormaat.

Je voert de test uit. Hij lukt.



Je commit de sources. Je publiceert op GitHub.



Security: zie takenbundel

## 19 ASPECT ORIENTED PROGRAMMING (AOP)

Je centraliseert bij AOP terugkerende taken in je code op één plaats.

Een ander woord voor een terugkerende taak is een crosscutting concern.

### 19.1 Je hebt al AOP toegepast bij CSS

In de plaats van bij elke h1 tag en bij elke h2 tag aan te geven dat die rood moet zijn:

```
<h1 style='color:red'>AOP</h1>
<h2 style='color:red'>Algemeen</h2>
<h2 style='color:red'>Concreet</h2>
```

definieer je deze terugkerende taak met CSS één keer:

```
h1, h2 {
  color: red;
}
```

CSS past deze opmaak toe op h1 en h2 elementen. Deze elementen moeten daartoe zelf geen inspanning doen (ze moeten geen speciale attributen hebben om de opmaak te krijgen):

```
<h1>AOP</h1>
<h2>Algemeen</h2>
<h2>Concreet</h2>
```

AOP heeft een eigen woordenschat, die we aan de hand van dit voorbeeld uitleggen:

<b>Advice</b>	De gecentraliseerde taak: het toepassen van de kleur rood: <code>color: red;</code>
<b>Pointcut</b>	Het wildcard patroon waarmee je advice toepast op applicatie onderdelen. Dit is <code>h1,h2</code> in de CSS: het patroon dat <code>h1</code> en <code>h2</code> elementen aanduidt.
<b>Aspect</b>	De combinatie van advice en pointcut. Dit is <code>h1, h2 {color: red;}</code> .
<b>Join point</b>	Een concreet applicatie onderdeel dat overeenkomt met een pointcut. <code>&lt;h2&gt;Algemeen&lt;/h2&gt;</code> is een join point: een concreet <code>h2</code> element dat past bij het wildcard patroon <code>h1,h2</code> in de CSS
<b>Target</b>	Een applicatie onderdeel dat één of meerdere join points bevat. Elke pagina met <code>h1</code> en/of <code>h2</code> elementen is een target.

### 19.2 Voordelen van het centraliseren van terugkerende taken

- **Hogere productiviteit.**

Als je de taak maar één keer schrijft win je tijd bij het schrijven, aanpassen en testen.

- **Leesbare code**

Je moet de terugkerende taak niet meer schrijven in de join points.

Die bevatten zo minder code en zijn leesbaarder.

- **Flexibiliteit**

Je kan de terugkerende taak vlot activeren of deactiveren. Het volstaat in een CSS bestand de opmaakregels van H1 in commentaar te plaatsen om deze opmaak te deactiveren.

Je kan AOP ook toepassen op Spring beans. Als je in meerdere beans een terugkerende taak hebt, hoef je die niet uit te programmeren in elke bean, maar slechts één keer in een centrale class.

Je duidt met een pointcut (wildcard patroon) de beans aan waarop Spring de code van die centrale class moet toepassen.

### 19.3 Pointcut expressie

Een pointcut expressie is een wildcard patroon waarmee je join points aanduidt.

Een join point is bij Spring AOP een method van een Spring bean.

Spring bevat meerdere sleutelwoorden om een pointcut expressie te definiëren.

Je leert hier het meest gebruikte en krachtigste sleutelwoord: `execution`.

#### 19.3.1 Syntax

`execution` (visibility returnType method (parameters) `throws` exceptionType)

#### 19.3.2 Onderdelen

- `visibility` (dit onderdeel is optioneel)  
De sleutelwoorden `public` of `private` of `protected` bij de method.
- `returnType`  
Het type van de returnwaarde van de method. Een `*` stelt om het even welk type voor.
- `method`  
De method naam. Je kan optioneel de class of interface van de method vermelden.  
Als je na de class of interface het teken `+` tikt, zijn ook methods van derived classes/interfaces join points.  
Je kan optioneel de package vermelden waartoe de class/interface behoort.  
Het jokerteken `*` staat voor nul of meerdere tekens.  
Je duidt met de tekenreeks `..` na een package aan dat ook alle subpackages van de package in aanmerking komen.
- `parameters`  
de parametertypes van de method. Als de parametertypes van een method afgeleid zijn van de parametertypes in het pointcut, is de method ook een join point. Voorbeelden:  
( ) een method zonder parameters.  
( .. ) een method met nul of meerdere parameters.  
( \* ) een method met één parameter van gelijk welk type.  
( \*, String ) een method met een eerste parameter van gelijk welk type en een tweede parameter van het type String.
- `throws` exceptionType (dit onderdeel is optioneel)  
De exception die de method werpt en beschreven is met `throws` bij de methoddeclaratie.

#### 19.3.3 Voorbeelden

- `execution(public * *(..))`  
Elke method met public visibility.
- `execution(* opslaan*(..))`  
Elke method waarvan de naam begint met opslaan.
- `execution(* be.vdab.groenetenen.services.FiliaalService.*(..))`  
Elke method van de interface FiliaalService in be.vdab.groenetenen.services.
- `execution(* be.vdab.groenetenen.services.*.*(..))`  
Elke method van interfaces en classes in be.vdab.groenetenen.services.
- `execution(* be.vdab.groenetenen..*.*(..))`  
Elke method van interface en classes in be.vdab.groenetenen en zijn subpackages.
- `execution(* *(..) throws java.io.IOException)`  
Elke method die een IOException (of afgeleide hiervan) werpt.

#### 19.3.4 Pointcuts combineren

Je kan pointcuts combineren met `&&` (and), `||` (or) en `!` (not). Voorbeeld:

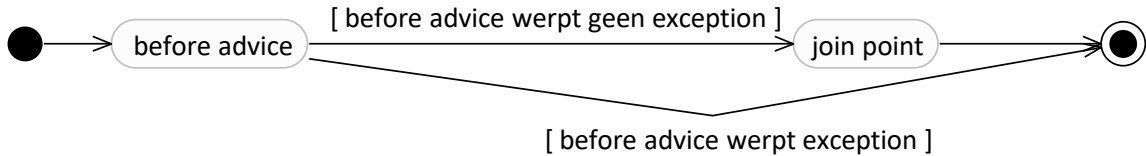
```
execution(* be.vdab.groenetenen.services.*.*(..)) || execution(* be.vdab.groenetenen.repositories.*.*(..))
```

Betekenis: elke method van interfaces en classes in be.vdab.groenetenen.services en be.vdab.groenetenen.repositories.

## 19.4 Types advice

### 19.4.1 Before

Spring voert before advice uit *vóór* het join point.



Je kan in het advice de parameterwaarden van het join point lezen.

Je kan de uitvoering van het join point verhinderen door een exception te werpen.

Auditing is een voorbeeld van before advice. Je houdt bij auditing bij welke gebruiker op welk moment welke use case uitvoert. Auditing wordt veel gebruikt in applicaties met confidentiële informatie. In een ziekenhuisapplicatie is het bijvoorbeeld belangrijk bij te houden welke medewerker welk patiëntendossier leest of bijwerkt en wanneer dit gebeurt.

Zonder AOP moet je in elke method van elke service bean code toevoegen om de auditing informatie weg te schrijven. Met AOP doe je dit slechts één keer.

Je kan de auditing informatie bijhouden in een database of (zoals in ons voorbeeld) in de logbestanden van de webserver.

Je maakt een package `be.vdab.groenetenen.aop` en daarin een class `Auditing`

```
package be.vdab.groenetenen.aop;
// enkele imports ...
@Aspect
@Component
class Auditing {
    private static final Logger LOGGER = LoggerFactory.getLogger(Auditing.class);
    @Before("execution(* be.vdab.groenetenen.services.*.*(..))")
    void schrijfAudit(JoinPoint joinPoint) {
        StringBuilder builder =
            new StringBuilder("Tijdstip\t").append(LocalDateTime.now());
        Authentication authentication =
            SecurityContextHolder.getContext().getAuthentication();
        if (authentication != null) {
            builder.append("\nGebruiker\t").append(authentication.getName());
        }
        builder.append("\nMethod\t\t")
            .append(joinPoint.getSignature().toLongString());
        Arrays.stream(joinPoint.getArgs())
            .forEach(object -> builder.append("\nParameter\t").append(object));
        LOGGER.info(builder.toString());
    }
}
```

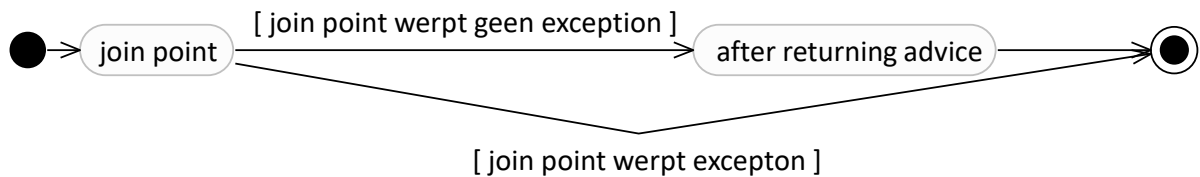
- (1) Je definieert een class als aspect met `@Aspect`.
- (2) Een aspect moet zelf ook een Spring bean zijn.
- (3) Je definieert een method als before advice met `@Before` met een pointcut expressie. De voorbeeldexpressie duidt methods uit `be.vdab.groenetenen.services` aan. Als je zo'n method oproept, voert Spring eerst de method `schrijfAudit` (zie 4) uit, daarna de opgeroepen method zelf.
- (4) Je geeft de method, die het advice voorstelt, een parameter van het type `JoinPoint`. Deze parameter verwijst naar het join point waarvóór Spring het advice uitvoert en geeft interessante informatie over dit join point.
- (5) Je bouwt de auditing informatie op in een `StringBuilder` object. Je voegt hier het tijdstip toe waarop een service method werd uitgevoerd.

- (6) Je haalt met `SecurityContextHolder.getContext().getAuthentication()` een `Authentication` object op. Dit bevat informatie over de ingelogde gebruiker.
- (7) De method `getSignature` van een `JoinPoint` geeft je de method declaratie van het join point als een `Signature` object.  
De method `toLongString` geeft je de package-, interface- én methodnaam.
- (8) `getArgs` geeft je een `Object` array met de parameterwaarden van het join point.

Je probeert de website uit. Naarmate je pagina's bezoekt die de service layer oproepen, zie je in het Eclipse venster Console extra auditing informatie in de webserver logbestanden.

#### 19.4.2 After returning

Spring voert after returning advice uit ná het join point, als dit join point géén exception werpt. Je kan de returnwaarde van het join point lezen.



Je wijzigt in de class `Auditing` de annotation bij de method `schrijfAudit`, zodat Spring deze method ná het join point uitvoert in plaats van er voor. Je wijzigt ook de method signatuur:

```

@AfterReturning(pointcut = "execution(* be.vdab.groenetenen.services.*(..))", ❶
    returning = "returnValue") ❷
void schrijfAudit(JoinPoint joinPoint, Object returnValue) { ❸
  
```

- (1) Je definieert een method als after returning advice met `@AfterReturning`.  
Je geeft een parameter `pointcut` mee met een pointcut expressie.  
De voorbeeldexpressie duidt methods uit `be.vdab.groenetenen.services` aan.  
Als je zo'n method oproept, voert Spring na die method de method `schrijfAudit` (zie 3) uit als de opgeroepen method geen exception werpt.
- (2) Je geeft een parameter `returning` mee als je de returnwaarde van het join point wil weten. Je tikt daarbij de naam van een `Object` parameter in de method `schrijfAudit`.  
Spring vult die parameter met de returnwaarde van het join point.
- (3) Je geeft de method een extra `Object` parameter.  
Spring vult die parameter in met de returnwaarde van het join point.

Je voegt code toe voor `LOGGER.info(builder.toString());`

```

if (returnValue != null) { ❶
    builder.append("\nReturn\t\t");
    if (returnValue instanceof Collection) { ❷
        builder.append(((Collection<?>) returnValue).size()).append(" objects"); ❸
    } else { ❹
        builder.append(returnValue.toString());
    }
}
  
```

- (1) Als het returntype van het join point `void` is, bevat `returnValue` `null`.
- (2) Als de returnwaarde een `Collection` (`List`, `Set`, `Map`) is
- (3) toon je enkel het aantal elementen in de verzameling, om de omvang van de auditing output te beperken.
- (4) Anders toon je de returnwaarde.

Je kan de applicatie uitproberen.

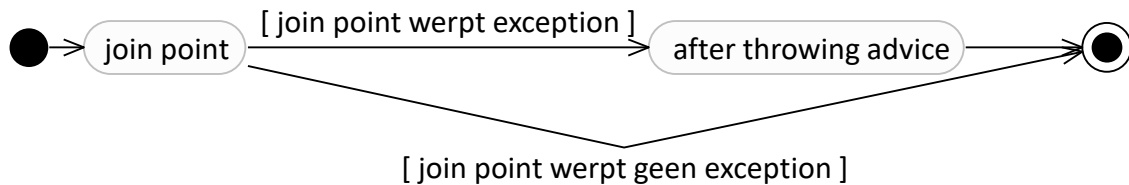
### 19.4.3 After throwing

Spring voert after throwing advice uit *na* het join point, als dit join point een exception werpt.

Je kan eventueel de geworpen exception opvangen en een andere exception werpen.

Je kan niet tegenhouden dat de geworpen exception verder naar boven borrelt (naar de code die het join point opgeroepen heeft).

Als je dit wil, moet je een ander type advice gebruiken: around advice (zie verder).



Typisch voorbeeld van after throwing advice is foutlogging:

je schrijft informatie over de geworpen exception in de logbestanden van de webserver.

Je maakt in `be.vdab.groenetenen.aop` een class `Logging`:

```

package be.vdab.groenetenen.aop;
// enkele imports
@Aspect
@Component
class Logging {
    private static final Logger LOGGER = LoggerFactory.getLogger(Logging.class);
    @AfterThrowing(pointcut = "execution(* be.vdab.groenetenen.web.*(..))",
        throwing = "ex")
    void schrijfException(JoinPoint joinPoint, Throwable ex) {
        StringBuilder builder = new StringBuilder("Tijdstip\t")
            .append(LocalDate.now());
        Authentication authentication =
            SecurityContextHolder.getContext().getAuthentication();
        if (authentication != null) {
            builder.append("\nGebruiker\t").append(authentication.getName());
        }
        builder.append("\nMethod\t\t")
            .append(joinPoint.getSignature().toLongString());
        Arrays.stream(joinPoint.getArgs())
            .forEach(object -> builder.append("\nParameter\t").append(object));
        LOGGER.error(builder.toString(), ex);
    }
}
  
```

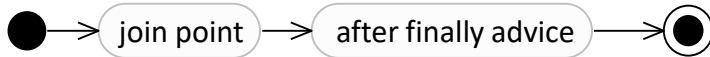
- (1) Je definieert een method als after throwing advice met `@AfterThrowing`.  
 Je geeft een parameter `pointcut` mee met een `pointcut` expressie.  
 De voorbeeldexpressie duidt methods uit `be.vdab.groenetenen.web` aan.  
 Als zo'n method een exception werpt, voert Spring na die method de method `schrijfException` uit.  
 Je geeft een parameter `throwing` mee als je de exception van het join point wil weten.  
 Je tikt daarbij de naam van een `Throwable` parameter in de method `schrijfException`.  
 Spring vult die parameter met de exception van het join point.
- (2) Je geeft de method een extra `Throwable` parameter.  
 Spring vult die parameter in met de exception van het join point.

Je stopt ActiveMQ. Je probeert in de website een offerte toe te voegen.

#### 19.4.4 After finally

Spring voert het advice uit *na* het join point, ongeacht of het al of niet een exception werpte. Dit type advice is beperkt:

- Als het join point een exception werpt, kan je niet weten welke exception dit is.
- Als het join point geen exception werpt, kan je de join point returnwaarde niet lezen.



Je houdt als voorbeeld per Service class method bij hoeveel keer die method opgeroepen werd.

Je maakt in `be.vdab.groenetenen.aop` een class `Statistieken`:

```

package be.vdab.groenetenen.aop;
// enkele imports
@Aspect
@Component
class Statistieken {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(Statistieken.class);
    private final ConcurrentHashMap<String, AtomicInteger> statistieken = ❶
        new ConcurrentHashMap<>();
    @After("execution(* be.vdab.groenetenen.services.*.*(..))") ❷
    void statistiekBijwerken(JoinPoint joinPoint) {
        String joinPointSignatuur = joinPoint.getSignature().toLongString();
        AtomicInteger vorigAantalOproepen =
            statistieken.putIfAbsent(joinPointSignatuur, new AtomicInteger(1)); ❸
        int aantalOproepen = vorigAantalOproepen == null ? 1 :
            vorigAantalOproepen.incrementAndGet(); ❹
        LOGGER.info("{} werd {} keer opgeroepen", joinPointSignatuur, ❺
            aantalOproepen);
    }
}
  
```

- (1) De key van deze map is een String met de signatuur van een join point. De value is hoeveel keer dit join point werd opgeroepen.
- (2) Je definieert een method als after finally advice met `@After` met een pointcut expressie. Elke keer je een method oproept die bij deze pointcut expressie hoort, voert Spring na de opgeroepen method de method `statistiekenBijwerken` uit, ongeacht of de opgeroepen method al of niet een exception werpt.
- (3) Als de join point signatuur nog niet voorkomt in de Map, voeg je een entry toe met als key de join point signatuur en als value 1.
- (4) Als de join point signatuur al voorkwam in de Map, verhoog je de bijbehorende value.
- (5) Spring vervangt de eerste `{}` door de inhoud van de 2° parameter (`joinPointSignatuur`). Spring vervangt de tweede `{}` door de inhoud van de 3° parameter (`aantalOproepen`).

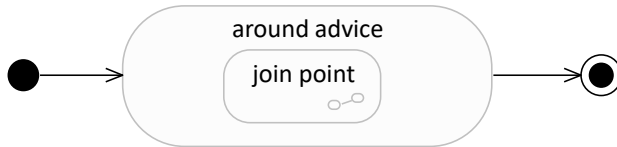
Je kan de applicatie uitproberen. Naarmate je pagina's bezoekt die de service layer oproepen, zie je statistiek informatie in de webserver logbestanden.



### 19.4.5 Around

Around advice *omsluit* het join point (het komt er voor én er na) en biedt heel wat mogelijkheden:

- Het advice kan code bevatten die Spring uitvoert vóór het join point.
- Het advice kan code bevatten die Spring uitvoert na het join point.
- Het advice kan beslissen het join point níét uit te voeren.
- Het advice kan de returnwaarde van het join point vervangen door een andere waarde.



Het join point is een optionele subactiviteit van het around advice. Het around advice kiest zelf om dit join point al of niet op te roepen.



Opmerking: around advice kan de andere types advice vervangen.

Het Spring team raadt echter af altijd around advice te kiezen.

De code van around advice is complexer dan de code van andere types advice en duidt minder goed aan wat het doel van het advice is.

Je houdt als voorbeeld per Service class method bij hoelang het uitvoeren duurde.

Je maakt in `be.vdab.groenetenen.aop` een class `Performance`:

```
package be.vdab.groenetenen.aop;
// enkele imports ...

@Aspect
@Component
class Performance {
    private static final Logger LOGGER = LoggerFactory.getLogger(Performance.class);
    @Around("execution(* be.vdab.groenetenen.services.*(..))")
    Object schrijfPerformance(ProceedingJoinPoint joinPoint)
    throws Throwable {
        long voor = System.nanoTime();
        try {
            return joinPoint.proceed();
        } finally {
            long duurtijd = System.nanoTime() - voor;
            LOGGER.info("{} duurde {} nanoseconden",
                joinPoint.getSignature().toLongString(), duurtijd);
        }
    }
}
```

- (1) Je definieert een method als around advice met `@Around` met een pointcut expressie. Elke keer je een method oproept die bij deze pointcut expressie hoort, voert Spring de method `schrijfPerformance` (zie 2) uit, niet de opgeroepen method zelf.
- (2) Je geeft aan de method die het advice voorstelt een `ProceedingJoinPoint` parameter mee. `ProceedingJoinPoint` erft van `JoinPoint` en bevat extra functionaliteit die je nodig hebt in around advice. Een around advice method moet als returntype `Object` hebben.
- (3) De method werpt alle fouten, die het join point eventueel werpt, verder naar de code die oorspronkelijk het join point opgeroepen heeft.
- (4) Voor je het join point uitvoert, onthoud je het resultaat van `System.nanoTime()`. Dit geeft je een getal dat daarna per nanoseconde met één verhoogt.
- (5) Bij around advice is het jouw keuze om het join point al of niet uit te voeren. Je voert het join point zelf uit met de `ProceedingJoinPoint` method `proceed`. Je doet deze oproep in een try blok gezien het join point een fout kan werpen. De returnwaarde van de `proceed` method bevat de returnwaarde van het join point zelf. Je geeft die door aan de code die het join point oorspronkelijk heeft opgeroepen.
- (6) Het finally blok wordt altijd opgeroepen (ook bij de return in het try blok).
- (7) Je berekent de duurtijd van het uitvoeren van het join point.

Je kan de applicatie uitproberen. Naarmate je pagina's bezoekt die de service layer oproepen, zie je performantie informatie in de webserver logbestanden.

#### 19.4.6 Gecentraliseerde pointcut expressions

De classes Auditing, Statistieken en Performance verwijzen naar dezelfde pointcut expressie. Als je deze moet wijzigen, moet je ze dan ook meerdere keren wijzigen.

Een betere oplossing is de pointcut expressie één keer centraal uit te schrijven en met een naam te associëren. Als in je applicatie deze pointcut expressie nodig hebt, verwijst je er naar met die naam.

Je beschrijft de gecentraliseerde pointcut expressies best in een aparte class.

Je maakt in `be.vdab.groenetenen.aop` een class `PointcutExpressions`:

```
package be.vdab.groenetenen.aop;
// enkele imports ...
@Aspect
class PointcutExpressions {
    @Pointcut("execution(* be.vdab.groenetenen.services.*.*(..))")
    void services() {}
}
```

❶

❷

❸

- (1) Je tikt `@Aspect` vóór de class die de gecentraliseerde pointcuts bevatten.
- (2) Je definieert een gecentraliseerde pointcut expressie met `@Pointcut` voor een method.
- (3) De methodnaam na `@Pointcut` is de naam die je associeert met de pointcut expressie. Je associeert `be.vdab.aop.PointcutExpressions.services()` met de pointcut expressie `execution(* be.vdab.groenetenen.services.*.*(..))`. De method heeft geen parameters en geeft void terug.

Je vervangt de pointcut expressie in de classes Auditing, Statistieken en Performance door:

```
be.vdab.groenetenen.aop.PointcutExpressions.services()
```

❶

- (1) Je verwijst naar de naam die je in de class `PointcutExpressions` met een pointcut expressie associeerde.

Je kan de applicatie uitproberen.

#### 19.4.7 Meerdere advices op hetzelfde pointcut

De advices uit de classes Auditing Performance en Statistieken worden opgeroepen op dezelfde methods (join points) uit de service layer.

Als Spring zo'n join point uitvoert, kan je met `@Order` bepalen in welke volgorde Spring de bijbehorende advices uitvoert. Je geeft aan `@Order` een geheel getal mee. Hoe groter dit getal, hoe dichterbij Spring het bijbehorende advice bij het join point zelf uitvoert.

Je tikt `@Order(1)` voor de class Auditing en `@Order(2)` voor de class Performance

Je kan de applicatie uitproberen.

Spring voert het join point uit, dan het Performance advice en daarna het Auditing advice.

Je wijzigt de `@Order` in Auditing naar 2 en de `@Order` in Performance naar 1.

Je kan de applicatie uitproberen.

Spring voert het join point uit, dan het Auditing advice en daarna het Performance advice.



Je commit de sources. Je publiceert op GitHub.

## 20 STAPPENPLAN

Je kan dit stappenplan volgen als je een web applicatie maakt met Spring en JPA:

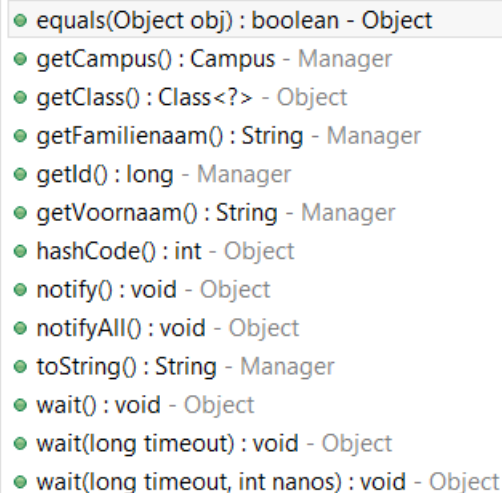
1. Converteer het project naar een JPA project.
2. Maak alle classes (Entities en Value objects) die personen en/of dingen uit de werkelijkheid voorstellen. Als je enkel Entities vindt en geen Value objects, heb je de werkelijkheid waarschijnlijk niet goed bestudeerd.

- a. Maak de Value object classes immutable (geen setters).
- b. Maak ook een Entity immutable als je die Entity enkel leest uit de database, niet toevoegt, wijzigt of verwijdert.  
Zo'n Entity heeft voor zijn associaties naar andere Entities of Value objects enkel getters, geen setters, add methods, remove methods.

Dit heeft voordelen:

- i. Je bent productief, want je schrijft minder code.
- ii. De Entity class is kort en daarom leesbaar en met minder bugs.
- iii. Als je in een andere laag kijkt welke methods je op een Entity kan toepassen, is het lijstje kort en dus overzichtelijk.

```
Manager manager;  
manager.
```





```
• equals(Object obj) : boolean - Object  
• getCampus() : Campus - Manager  
• getClass() : Class<?> - Object  
• getFamilienaam() : String - Manager  
• getId() : long - Manager  
• getVoornaam() : String - Manager  
• hashCode() : int - Object  
• notify() : void - Object  
• notifyAll() : void - Object  
• toString() : String - Manager  
• wait() : void - Object  
• wait(long timeout) : void - Object  
• wait(long timeout, int nanos) : void - Object
```

Press 'Ctrl+Space' to show Template Proposals

- c. Maak geen setters voor 'alleen-lezen' eigenschappen (bvb. autonummer kolom).
- d. Maak in elk van die classes de methods equals en hashCode.  
Als je in de applicatie een Set vult met objecten van die classes, zal die Set dan correct werken. Baseer de methods equals en hashCode niet op de private variabele die bij de primary key hoort, maar op een andere private variabele (of combinatie van private variabelen). Deze variabele (of combinatie van variabelen) moet een unieke waarde bevatten per object van de class).
- e. Beschrijf in Entities én in Value objects een relatie naar een Entity niet als een getal, maar als een reference variabele.

Deze reference variabele heeft als type die andere Entity.

 verkeerd	 correct
<pre>... public class Docent { ... <del>private long campusNr;</del> ... }</pre>	<pre>... public class Docent { ...     @ManyToOne(optional = false)     @JoinColumn(name = "campusid")     ..private Campus campus; }</pre>

- f. Plaats een berekening die met een Entity of Value object te maken heeft in de class van die Entity of Value object, niet in Controllers, Services of Repositories.

```
...
public class Artikel {
    ...
    @NumberFormat(pattern = "0.00")
    private BigDecimal prijsExclusief;
    @NumberFormat(pattern = "0.00")
    private BigDecimal btwPercentage;
    @NumberFormat(pattern = "0.00")
    public BigDecimal prijsInclusief() {
        return prijsExclusief.multiply(BigDecimal.ONE
            .add(btwPercentage.divide(BigDecimal.valueOf(100), 2,
                RoundingMode.HALF_UP)));
    }
}
```

Eerste voordeel: een berekening in een Entity of Value object kan je vanuit alle andere onderdelen van je applicatie oproepen.

Tweede voordeel: als de berekening wijzigt, doe je die wijziging één keer (en niet meerdere keren als die berekening in meerdere JSP's gebeurt).

Test de correcte werking van zo'n method met een unit test.

- g. Definieer de opmaak van BigDecimal en Date private variabelen met @NumberFormat en @DateTimeFormat
  - h. Voeg Bean validation annotations toe.
3. Doe volgende stappen per nieuwe pagina van de web applicatie
- a. Voeg Repository interfaces (afgeleid van JpaRepository) en methods toe.
  - b. Voeg Service interfaces, implementatie classes en methods toe.  
Tik @Service bij Service classes. Injecteer de nodige repositories in de constructor.  
Als een method records toevoegt, wijzigt of verwijdert doe je dit in een transactie.  
Als een method een record leest én daarna ook wijzigt,  
pas je optimistic of pessimistic record locking toe.  
Om een record te wijzigen volstaat het dit record te lezen en te wijzigen in het interne geheugen. JPA doet dezelfde wijziging in de database voor jou.
  - c. Voeg een Controller class toe, waarbij je @Controller tikt.  
Injecteer de nodige Services in de constructor.  
Als je session data, onthoud je daarin geen Entities,  
maar identifiers van die Entities.
  - d. Voeg een Thymeleaf pagina toe.
  - e. Test de pagina. Als die een N + 1 performantieprobleem heeft,  
los je dit probleem op met Named Entity Graphs.

## **21 COLOFON**

<b>Domeinexpertisemanager:</b>	Jean Smits
<b>Moduleverantwoordelijke:</b>	Hans Desmet
<b>Medewerkers:</b>	Hans Desmet
<b>Versie:</b>	13/9/2018
<b>Nummer dotatielijst:</b>	