



samen sterk voor werk

Spring fundamentals

Deze cursus is eigendom van de VDAB©

Inhoudsopgave

1	INLEIDING.....	8
1.1	Doelstelling.....	8
1.2	Vereiste voorkennis.....	8
1.3	Nodige software	8
2	BROWSERS, WEBSERVERS, WEBSITES, URL'S, WEBFARM, TOMCAT	9
2.1	Port number	9
2.2	Websites en URL's	9
2.3	Webfarm	10
2.4	Tomcat.....	10
2.4.1	Downloaden.....	11
2.4.2	Installeren	11
2.4.3	Starten	11
2.4.4	Stoppen.....	11
2.4.5	Website installeren	11
3	REQUESTS EN RESPONSES	13
3.1	Request onderdelen	13
3.1.1	Method	13
3.1.2	Headers.....	14
3.1.3	Body	14
3.1.4	Query string	14
3.2	Response onderdelen.....	15
3.2.1	Status code	15
3.2.2	Headers.....	15
3.2.3	Body	15
3.3	Samenvatting.....	15
4	WEBSERVER ALS CONTAINER – EMBEDDED WEBSERVER	16
5	ECLIPSE INSTELLINGEN	17
6	SPRING BOOT EN START.SPRING.IO.....	18
7	EERSTE WEBSITE MET STATIC CONTENT	20
7.1	Statische pagina	20

7.2	Welkompagina	20
7.2.1	Pagina	20
7.2.2	Afbeeldingen.....	21
7.2.3	CSS	21
7.2.4	Uittesten	21
7.2.5	WAR	21
8	CONTROLLER	22
8.1	Automatisch herstarten bij wijzigingen	23
8.2	Nadelen van de huidige manier van werken.....	23
9	CONTROLLER EN JSP COMBINEREN	24
9.1	pom.xml.....	24
9.2	JSP.....	24
9.3	Controller bean	26
9.4	Data doorgeven van de controller naar de JSP	26
9.5	application.properties	27
9.6	Meerdere data doorgeven aan de JSP	27
10	<code>\${EXPRESSION LANGUAGE}</code>.....	28
10.1	Primitief datatype.....	28
10.2	Object	28
10.3	Onbestaande data	28
10.4	Hard gecodeerde waarden.....	28
10.5	Wiskundige operatoren.....	28
10.6	Vergelijkingsoperatoren.....	28
10.7	Logische operatoren.....	28
10.8	Conditionele operator ? :	28
10.9	Operator empty.....	28
10.10	Één element uit een verzameling lezen	29
10.10.1	Array	29
10.10.2	List.....	29
10.10.3	Map.....	29
10.11	Resultaat van een method oproep.....	29

11	JAVABEAN	30
11.1	Getters en setters.....	30
11.2	ReadOnly attributen.....	30
11.3	Getters en Setters maken met Eclipse	31
11.4	Constructors	31
11.5	Constructors maken met Eclipse.....	31
11.6	EL en JavaBeans.....	31
11.7	Geneste attributen	32
12	JSTL (JSP STANDARD TAG LIBRARY)	33
12.1	Tag names en tag URI's	33
12.2	<c:forEach>	33
12.2.1	Itereren over een verzameling JavaBeans	34
12.2.2	Itereren over een Map.....	35
12.2.3	begin, step en end attributen	35
12.2.4	Itereren zonder verzameling.....	35
12.2.5	varStatus attribuut.....	35
12.3	<c:if>	36
12.4	<c:choose>.....	36
12.5	<c:out>.....	37
12.6	<c:url>.....	37
12.7	<c:import>	37
12.7.1	Parameters	38
12.8	<c:set>	38
13	RELATIEVE URL'S.....	39
13.1	Probleem	39
13.2	Oplossing	39
14	PARAMETERS IN DE QUERY STRING	40
15	CLEAN URL'S MET PATH VARIABLEN	41
15.1	Clean URL's maken in je JSP	41
15.2	Clean URL verwerken in een @GetMapping method	41

15.3	URL met meerdere path variabelen	42
16	UNIT TEST VAN EEN CONTROLLER	43
17	REQUEST HEADERS	44
18	COOKIES	45
18.1	Cookie.....	45
19	MULTITHREADING	47
19.1	AtomicInteger voorbeeld	47
20	STATELESS	48
21	MODEL-VIEW-CONTROLLER	50
22	ARCHITECTUUR VAN EEN ENTERPRISE APPLICATION	51
22.1	Repositories.....	51
22.2	RestClients.....	52
22.3	RestServices.....	52
22.4	Services.....	52
22.4.1	Samenwerking tussen de layers.....	53
23	DEPENDENCY INJECTION	54
23.1	Dummy objecten gebruiken bij unit testen	54
23.1.1	Interface	54
23.1.2	Service	55
23.1.3	Test van de service.....	55
23.2	Eerste implementatie van de dependency	56
23.3	Dependency injection in de controller.....	57
23.4	Als een dependency ontbreekt	59
24	DE IOC CONTAINER	60
25	MEERDERE IMPLEMENTATIES VAN EEN DEPENDENCY	61
25.1	@Primary.....	62
25.2	@Qualifier	63
25.2.1	@Qualifier bij de bean classes	63
25.2.2	@Qualifier bij constructor injection	63

25.3	Meerdere dependencies injecteren.....	63
26	APPLICATION.PROPERTIES	65
26.1	application.properties	65
26.2	Beans	65
27	INTEGRATION TEST VAN EEN SPRING BEAN.....	66
28	DATABASE – DATASOURCE	67
28.1	Database.....	67
28.2	DataSource	67
28.3	JDBC driver	68
28.4	Integration test.....	68
29	REPOSITORIES LAYER	70
29.1	Exceptions	70
29.2	PizzaNietGevondenException	70
29.3	Interface	71
29.4	JdbcTemplate	71
29.5	Repository bean	71
29.5.1	Scalar value lezen.....	72
29.5.2	Update of delete SQL statement met één parameter	72
29.5.3	Update of delete SQL statement met meerdere parameters.....	72
29.5.4	Record toevoegen.....	72
29.5.5	RowMapper	73
29.6	Integration test.....	75
30	SERVICES EN TRANSACTIES.....	79
30.1	Interface	79
30.2	Implementatie	79
30.3	Transactie eigenschappen.....	80
30.3.1	Isolation level.....	80
30.3.2	Read-only	80
30.3.3	Timeout.....	81
30.4	@Transactional	81
30.5	Propagation	81

30.6	Services laag oproepen in de controller.....	83
31	HTML FORMS.....	85
31.1	Form object	85
31.1.1	Form tonen aan de gebruiker	85
31.1.2	Ingetikte waarden lezen	85
31.1.3	Voorwaarden voor de command object class.....	85
31.1.4	Package voor de command object class	85
31.1.5	Command object voorbeeld	86
31.2	Form tonen aan de gebruiker.....	86
31.3	Spring form tag library	86
31.4	Form verwerken na de submit	87
31.5	Lege invoervakken tonen	88
31.6	Invoer valideren	88
31.6.1	Verkeerd type data	88
31.6.2	Foutboodschappen	88
31.6.3	Foutboodschappen maken in de controller bean	89
32	BEAN VALIDATION	91
32.1	Annotations	91
32.2	@Valid	92
32.3	Foutboodschappen.....	92
32.4	Valideren met @Valid	93
32.5	Form object class.....	93
32.6	Controller class wijzigingen	93
32.7	Unit test.....	94
33	CLIENT SIDED VALIDATIE	96
34	FORM MET METHOD = POST	97
34.1	Het refresh probleem en POST-REDIRECT-GET	98
34.1.1	Request parameters meegeven bij een redirect.....	99
34.2	Dubbele submit vermijden.....	100
35	CROSS-SITE SCRIPTING (XSS)	101
36	SESSION SCOPED BEANS	102

36.1	Session.....	102
36.2	Serializable	103
36.2.1	Session persistence.....	103
36.2.2	Session replication	103
36.3	Session identificatie.....	103
36.4	Webserver verwijdert een session	104
36.5	Voorbeeld 1	104
36.5.1	Session data als Spring bean	104
36.6	Voorbeeld 2	106
36.7	Session fixation.....	109
37	GETALOPMAAK, DATUMOPMAAK, TIJDOPMAAK	110
37.1	Getalopmaak	110
37.2	spring:eval	111
37.3	Datumopmaak, tijdopmaak.....	111
38	CUSTOM TAGS	113
38.1	TLD bestand.....	113
38.2	vdab.tld.....	113
38.3	menu.tag	113
38.4	Custom tag gebruiken in JSP	114
38.5	Custom tag attributen	114
39	STAPPENPLAN	115
40	HERHALINGSOEFENINGEN	116
41	COLOFON.....	117

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 onderdelen: database toegang, web toegang, ...

1.2 Vereiste voorkennis

- | | |
|---|--|
| <ul style="list-style-type: none">• Java• JDBC | <ul style="list-style-type: none">• Maven• Unit testing |
|---|--|

1.3 Nodige software

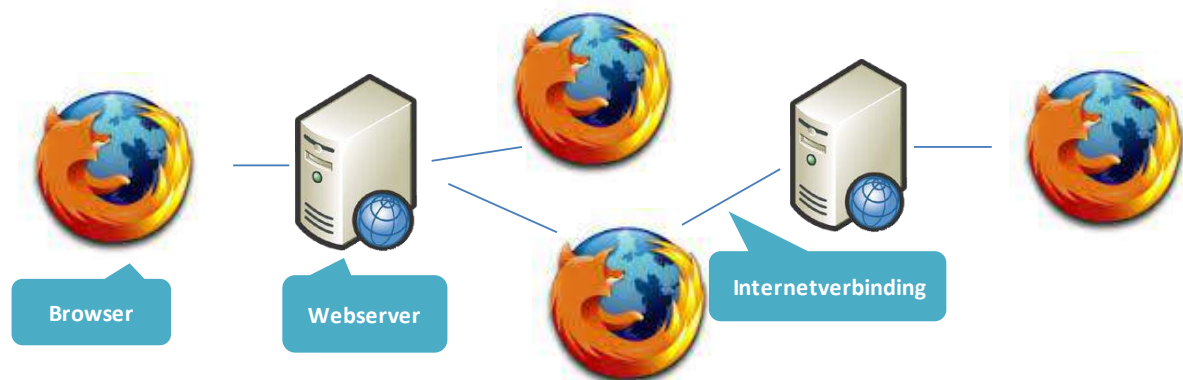
- een JDK (Java Developer Kit) met versie 8 of hoger.
- een relationele database. Je gebruikt in de cursus MySQL. (www.mysql.com)
- Tomcat 8.
- Eclipse IDE for Java EE Developers (versie Photon).

2 BROWSERS, WEBSERVERS, WEBSITES, URL'S, WEBFARM, TOMCAT

De computers waarop de browsers draaien en de computers waarop webserver draaien zijn via het internet of intranet (bedrijfsnetwerk) met mekaar verbonden.

Browsers en webserver wisselen data uit met HTTP (HyperText Transfer Protocol).

HTTP gebruikt zelf TCP/IP: Transmission Control Protocol / Internet Protocol.



Ook andere diensten gebruiken TCP/IP. Voorbeelden van zo'n diensten:

- SMTP Simple Mail Transfer Protocol (om mails te versturen).
- FTP File Transfer Protocol (om bestanden uit te wisselen).

2.1 Port number

Op één computer kunnen meerdere programma's TCP/IP gebruiken.

Elk programma krijgt op die computer een uniek identificatiegetal: het port number:

- 80 Webserver
- 21 FTP (File Transfer Protocol) server
- 25 Mail server

2.2 Websites en URL's

Een webserver bevat één of meerdere websites. Een website bevat pagina's. Elke pagina heeft een unieke identificatie: de URL (Uniform Resource Locator). Een URL heeft volgende opbouw:

<http://pizzaluigi.be/pizzas>

protocol

domeinnaam

pad

- Alle pagina's van een website delen dezelfde domeinnaam.
- Een domeinnaam is niet hoofdlettergevoelig, een pad wel.
- Je kan surfen naar een URL en geen pad meegeven (pizzaluigi.be).
Je ziet dan de 'welkompagina' van die website.
- Als een webserver afwijkt van het standaard TCP/IP port number (80),
vermeld je, bij het surfen, in de URL ook het port number: pizzaluigi.be:8080/pizzas.



Opmerking: URI (Universal Resource Identifier) is een identifier voor een stukje data.

Elke URL is een URI, maar niet elke URI is een URL.

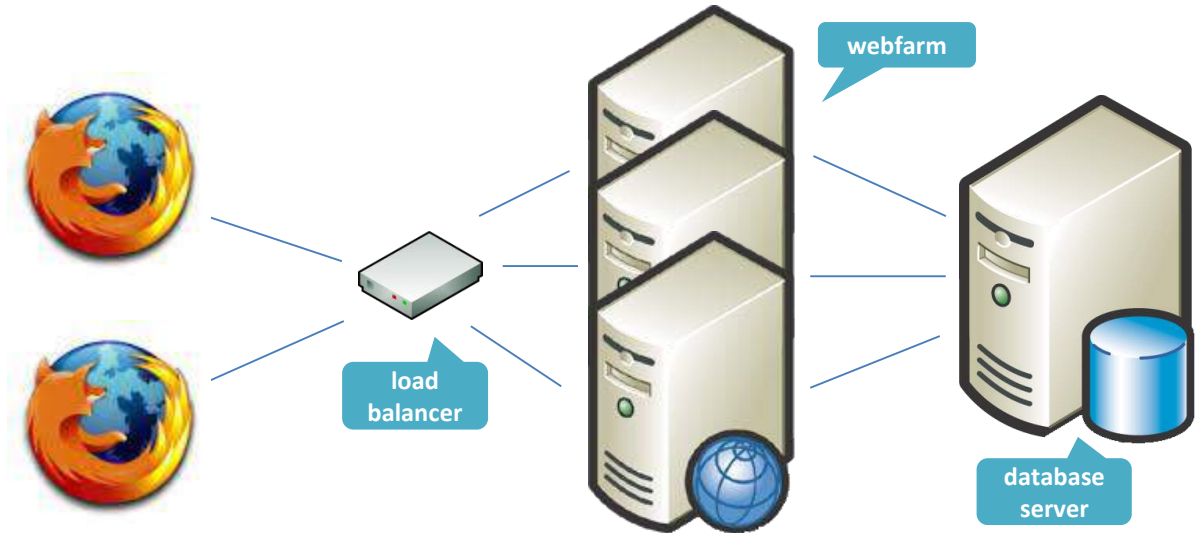
Een URI is een URL als de URI een protocol (zoals http) gebruikt om de plaats van de data aan te geven.

2.3 Webfarm

Een website draait meestal niet op één, maar op *meerdere* webserver. Zo'n groep webserver heet een webfarm of een cluster.

Een load balancer is een apparaat in een webfarm.

Hij verdeelt het werk (de pagina's leveren die de browsers bezoeken) over die webserver.



Voordelen

- Als één webserver uitvalt, verdeelt de load balancer het werk over de andere webserver in de webfarm. De website blijft zo ter beschikking.
- De load balancer verdeelt het werk (de browser vragen) over de webserver in de webfarm. Zo raakt één webserver niet overladen met werk en blijft de website performant.

2.4 Tomcat

Veel webserver ondersteunen Java. Veel gebruikte webserver zijn:

Webserver	Van de firma
Tomcat	Apache
Jetty	Eclipse
WildFly	JBoss
WebLogic	Oracle
WebSphere	IBM

Je kan je Java website uitvoeren op al die webserver, zonder de code te wijzigen.

Je gebruikt in de cursus de populairste Java webserver: Tomcat

2.4.1 Downloaden

Je kan Tomcat downloaden op tomcat.apache.org

1. Je klikt links in het onderdeel Download op de hyperlink Tomcat 9.
2. Je kiest daar bij Binary Distributions voor Core.
3. Je kiest daar de hyperlink zip (...).



2.4.2 Installeren

Je pakt het ZIP bestand uit in een directory op je computer.

Je maakt in je besturingssysteem een environment variabele met de naam JAVA_HOME.

Die variabele bevat het absolute pad van de directory waarin Java geïnstalleerd is.

Je doet dit op Windows als volgt:

1. Je kiest het Control Panel.
2. Je kiest System.
3. Je kiest Advanced system settings.
4. Je kiest het tabblad Advanced.
5. Je kiest Environment Variables en je kiest New onder User variables for ...
6. Je tikt JAVA_HOME bij Variable name.
7. Je tikt het absolute pad naar de directory waarin Java geïnstalleerd is (bijvoorbeeld C:\Program Files\Java\jdk1.8.0_1) bij Variable value.
8. Je kiest drie keer OK.

2.4.3 Starten

Je dubbelklikt startup.bat in de subdirectory bin van de Tomcat Directory.

- Je ziet een Command-Prompt venster waarin Tomcat opstart.
- Je ziet in dit venster enkele diagnostische meldingen tijdens het opstarten.
- Je ziet als laatste melding INFO: Server startup in x ms.

Tomcat is nu gestart. Je laat dit venster openstaan, anders sluit je Tomcat terug af.

Tomcat is in uitvoering op je computer en gebruikt TCP poort 8080.

In een netwerk is localhost een synoniem voor je eigen computer.

Je tikt in de browser adresbalk localhost:8080 en je ziet de Tomcat welkompagina.

2.4.4 Stoppen

1. Je drukt Ctrl+C in het Command-prompt venster waarin Tomcat draait of.
2. Je dubbelklikt shutdown.bat in de bin directory van Tomcat.
3. Het Command-Prompt venster met Tomcat verdwijnt na enkele seconden.

2.4.5 Website installeren

2.4.5.1 WAR bestanden



Een WAR (web archive) is een bestand met de extensie war, maar is intern een ZIP. Het bevat alle ingrediënten van een Java website (code, afbeeldingen, CSS, ...).

2.4.5.2 Tomcat gebruikers

Enkel geregistreerde Tomcat gebruikers kunnen via de browser een website op Tomcat installeren. Elke gebruiker heeft één of meerdere rollen (roles).

Enkel gebruikers met de role manager-gui kunnen via de browser een website installeren.

Het bestand tomcat-users.xml in de Tomcat subdirectory conf bevat de gebruikers.

Je maakt in dit bestand een gebruiker met de role manager-gui.

Je opent dit bestand met Eclipse: menu File, Open File.

1. Je maakt een blanco regel juist onder de regel `<tomcat-users ...>` en je tikt daarin `<user username="cursist" password="cursist" roles="manager-gui,manager-script"/>`
2. Je slaat het bestand op.
3. Je herstart Tomcat.

2.4.5.3 Installatie via de browser

1. Je start Tomcat.
2. Je surft met een browser naar localhost:8080.
3. Je kiest in de Tomcat welkompagina de knop Manager App.
4. Je tikt als gebruikersnaam én als paswoord cursist en je logt in.
5. Je kiest de knop naast Select WAR file to upload.
6. Je duidt sterrenbeelden.war (bestand bij cursus) aan op je harde schijf.
7. Je kiest de knop Deploy (to deploy = installeren).
8. Je ziet na enkele seconden /sterrenbeelden in de lijst van websites (Applications):

/sterrenbeelden	None specified		true	0	Start Stop Reload Undeploy
					Expire sessions with idle ≥ 30 minutes

9. De hyperlink /sterrenbeelden brengt je naar de website. Je kan die testen.
10. Je stopt de website met de knop Stop. Als je daarna surft naar de website, zie je een pagina met status code 404 (Not found): de website is niet actief.
11. Je start de website terug met de knop Start.
12. Je herstart de website met de knop Reload.
13. Je verwijdert de website van Tomcat met de knop Undeploy.
14. Een website kan per gebruiker data bijhouden in het RAM geheugen (zoals een winkelmandje). Je verwijdert die data, als die gedurende 30 minuten niet gelezen of gewijzigd werd, met de knop Expire sessions.

Gelieve de website te verwijderen, want je installeert hem straks via het bestandsbeheer.

2.4.5.4 Installatie via het bestandsbeheer

Je kopieert sterrenbeelden.war naar de Tomcat subdirectory webapps.

Tomcat installeert elke WAR in die directory automatisch als een website.

Je surft naar de hoofdlettergevoelige URL van de website: localhost:8080/sterrenbeelden.

Je verwijdert de website door sterrenbeelden.war te verwijderen uit de directory webapps.



Je laat de Tomcat draaien. Je hebt hem nodig in het volgende hoofdstuk.

3 REQUESTS EN RESPONSES

Telkens je

- een URL tikt in de browser adresbalk
- of een URL kiest in de browser favorieten
- of een hyperlink aanklikt
- of een knop aanklikt in een formulier van een webpagina



stuurt de browser een request (vraag) naar een webserver en krijgt een response (antwoord) terug. Dit antwoord bevat HTML, CSS, JavaScript en/of afbeelding(en).

De browser hertekent met dit antwoord zijn beeld.

3.1 Request onderdelen

Een request bevat een method, headers, een optionele body en een optionele query string.

3.1.1 Method

De method definieert het *soort* request met één woord: GET of POST. HTTP schrijft voor:

GET	<p>Gebruik GET bij elke request waarmee de gebruiker enkel <i>data vraagt</i>. Voorbeeld: een request met de method GET naar <code>pizzaluigi.be/producten</code> vraagt producten op.</p> <p>Een request heeft als method GET als</p> <ul style="list-style-type: none"> • de gebruiker een URL tikt in de adresbalk van de browser en Enter drukt. • de gebruiker een hyperlink aanklikt • de gebruiker een formulier verstuurt waarvan de method op get staat.
POST	<p>Gebruikt POST bij elke request die meer doet dan enkel data vragen. Voorbeeld: een request met de method POST naar <code>pizzaluigi.be/producten/toevoegen</code> voegt een product toe.</p> <p>Een request heeft als method POST als</p> <ul style="list-style-type: none"> • de gebruiker een formulier verstuurt waarvan de method op post staat.

Je ziet hiervan voorbeelden met Firefox.

1. Je kiest rechts boven in Firefox .
2. Je kiest Web Developer.
3. Je kiest Network.

Je ziet vanaf nu onder in het venster technische informatie over de requests en responses.

4. Je surft naar <http://localhost:8080/sterrenbeelden>

Je ziet onder in het venster een request naar de welkompagina:

● 200 GET ☐ /sterrenbeelden/

De pagina bevat een verwijzing naar `default.css`. Je ziet dus ook een request naar `default.css`:

● 200 GET ☐ default.css

Beide requests vragen enkel data. De requests gebruiken daarbij de method GET.

Tweede voorbeeld: je tikt een naam en een bericht en je kiest de knop Toevoegen

De klik op de knop veroorzaakt een request met de method POST. De request doet meer dan data vragen: de request voegt een item toe aan het gastenboek. Daarna volgt een request met de method GET die de pagina terug opvraagt en een request naar `default.css`.

▲ 302 POST ☐ /sterrenbeelden/
 ● 200 GET ☐ /sterrenbeelden/
 ● 200 GET ☐ default.css

Je leert verder in de cursus hoe je in je website requests verwerkt met de GET en de POST method.

Als je GET en POST niet gebruikt zoals het HTTP protocol het voorschrijft, krijg je problemen. Voorbeeld: een pagina toont informatie over een product. Deze pagina bevat ook een hyperlink Verwijderen. Als de gebruiker hierop klikt, verwijdert de website het produkt uit de database.

Naast mensen bezoeken ook zoekrobots de pagina. Zoekrobots volgen elke hyperlink in een pagina, in de hoop dat die hen naar andere interessante pagina's zal leiden. Ze mogen dit ook doen: een hyperlink volgen is een GET request en het HTTP protocol schrijft voor dat een GET request enkel data leest. De hyperlink Verwijderen. (en de bijbehorende GET request) volgt het HTTP protocol niet. Gevolg: de zoekrobot verwijdert het product bij het volgen van de hyperlink.

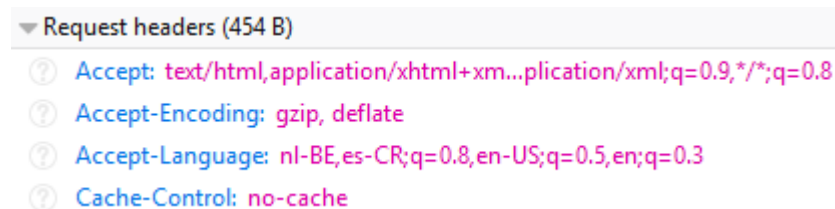
Als je het HTTP protocol volgt, vervang je de hyperlink Verwijderen door een knop Verwijderen. Deze is een onderdeel van een form met het attribuut method gelijk aan post. Zoekrobots volgen geen POST requests, omdat ze weten dat ze zo je website kunnen beschadigen.

3.1.2 Headers

Headers bevatten browserinformatie. Elke header heeft een naam en een waarde.

Voorbeeld: je surft naar <http://localhost:8080/sterrenbeelden>

Je klikt de request onder in het venster aan. Je kiest daarna Headers onder in het venster. Je ziet onder andere de headers van de request:



- De user-agent header bevat het browsertype en het besturingssysteem.
- De accept-language header bevat de voorkeur talen en landsinstellingen van de gebruiker, bijvoorbeeld: en-US, nl-BE.

3.1.3 Body

Een GET request (request met de method GET) bevat geen body.

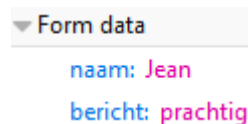
Een POST request (request met de method POST) kan een body hebben.

Die bevat data waarmee de website de request verwerkt.

- Elk stuk data heeft een naam en een waarde, gescheiden door =
- Meerdere stukken data worden van mekaar gescheiden door &

Voorbeeld in sterrenbeelden: je tikt een naam en een bericht en je kiest de knop Toevoegen.

De klik op de knop veroorzaakt een request met de method POST. Je klikt deze request onder in het venster aan. Je kiest daarna Params onder in het venster. Je ziet de data van de body in tabelvorm:



3.1.4 Query string

Een GET request heeft geen body. Een GET request kan wel data meegeven in de query string. Deze zit op het einde van de URL, begint met ? en bevat één of meerdere parameters.

- Elke parameter heeft een naam en een waarde, gescheiden door =
- Meerdere parameters worden gescheiden door &

Voorbeeld in sterrenbeelden: je tikt een geboortedatum en je kiest de knop Sterrenbeeld.

Je ziet onder in het venster een request met een query string met een parameter datum:



3.2 Response onderdelen

3.2.1 Status code

De status code geeft met een getal aan hoe de website de request verwerkte. Voorbeelden:

- 200 (OK) De request is correct verwerkt.
- 404 (Not Found) De URL bestaat niet in de website.

3.2.2 Headers

Headers bevatten informatie over de response. Elke header heeft een naam en een waarde.

Voorbeeld: de header content-type bevat het MIME type (datatype) van de data in de body:

- text/html HTML
- text/css CSS
- text/javascript JavaScript
- image/png Een afbeelding in PNG formaat

Voorbeeld in sterrenbeelden: je tikt een geboortedatum en je kiest de knop Sterrenbeeld.

Je klikt de request met de method GET onder in het venster aan.

Je kiest daarna Headers onder in het venster. Je ziet onder andere de headers van de response:

```

▼ Response headers (114 B)
  ? Content-Length: 970
  ? Content-Type: text/html; charset=UTF-8
  ? Date: Fri, 17 Nov 2017 12:12:09 GMT

```

3.2.3 Body

De body bevat data die de request vraagt. Dit kan HTML zijn, CSS, een afbeelding, ...

Voorbeeld in sterrenbeelden: je tikt een geboortedatum en je kiest de knop Sterrenbeeld.

Je klikt de request met de method GET onder in het venster aan.

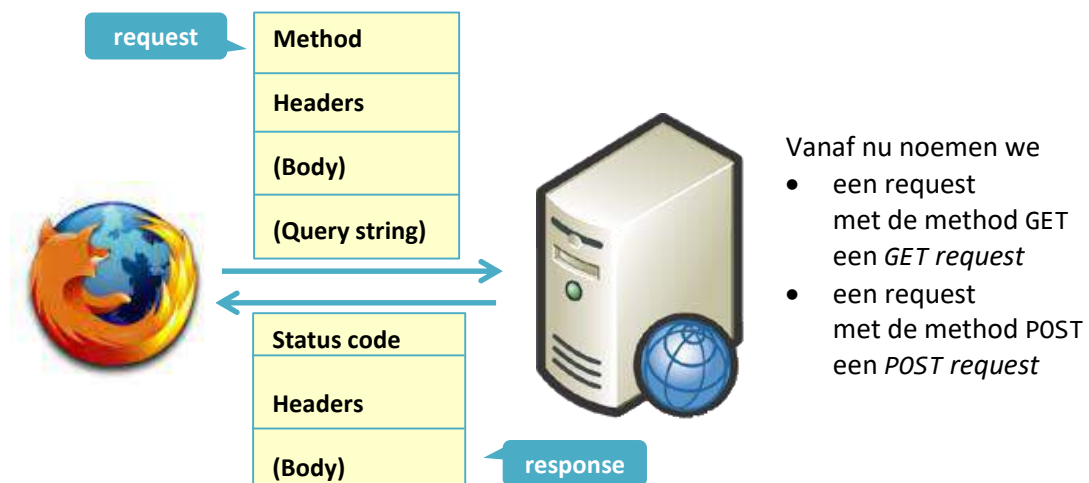
Je kiest daarna Response onder in het venster. Je ziet de HTML in de body van de response.

```

▼ Response payload
1 <!DOCTYPE html>
2 <html lang="nl">
3 <head>
4 <title>Sterrenbeelden</title>

```

3.3 Samenvatting



Je stopt nu Tomcat, zodat die niet in conflict komt met een andere Tomcat die je straks zal gebruiken..

4 WEBSERVER ALS CONTAINER – EMBEDDED WEBSERVER

Één webserver kan meerdere websites bevatten.

Je hebt daarbij het risico dat als één van die websites slecht werkt, dit ook nadelen geeft voor de andere websites op dezelfde webserver.

Om dit te verhinderen installeert men meestal slechts 1 website op een webserver.

De webserver speelt de rol van container, die de website in zich bevat.



In moderne omgevingen gaat men nog een stap verder.

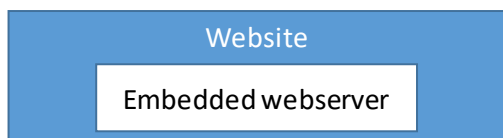
De webserver wordt een onderdeel van de website.

Men spreekt dan over een embedded webserver.

Je hoeft dan niet een webserver én de website te installeren.

Het volstaat de website te installeren.

Je zal in deze cursus ook een embedded webserver gebruiken.



Een website kan verpakt zijn in een WAR bestand of in een JAR bestand.

Je verpakt je website in een WAR bestand als

- Je de website wil installeren in een webserver die draait als een container (bovenste afbeelding) of
- Je in de website JSP's gebruikt om HTML naar de browser te sturen. Je leert verder in de cursus JSP's kennen.

In andere gevallen kan je je website verpakken in een JAR bestand. Je kan met enkele stappen een website, die oorspronkelijk verpakt was in een JAR bestand, verpakken in een WAR bestand.

Uiteindelijk blijft er dus één reden over om een website onmiddellijk te verpakken in een WAR bestand: als je JSP's gebruikt om HTML naar de browser te sturen. Je zal dit doen in deze cursus.

In de cursus "Spring advanced" zal je op een andere manier HTML naar de browser sturen.

Je zal je website dan verpakken in een JAR bestand.

5 ECLIPSE INSTELLINGEN

Je gebruikt UTF-8 encoding om de menselijke tekens van je webpagina's voor te stellen. UTF-8 kan alle menselijke tekens ter wereld voorstellen.

Eclipse gebruikt voor enkele bestandstypes die bij webpagina's horen standaard de oudere ISO-8859-1 encoding. Deze kan niet alle menselijke tekens te wereld voorstellen.

Je wijzigt enkele instellingen in Eclipse om dit op te lossen:

1. Je kiest in het menu Windows de opdracht Preferences.
2. Je dubbelklikt links Web.
3. Je kiest links CSS Files.
4. Je kiest rechts ISO 10646/Unicode(UTF-8) bij Encoding.
5. Je kiest links HTML Files.
6. Je kiest rechts ISO 10646/Unicode(UTF-8) bij Encoding.
7. Je kiest links JSP Files.
8. Je kiest rechts ISO 10646/Unicode(UTF-8) bij Encoding.
9. Je kiest Apply and Close.



Als je een andere workspace kiest, moet je voor die workspace deze instellingen opnieuw doen.

6 SPRING BOOT, START.SPRING.IO

Vooraleer Spring Boot bestond, had je veel werk om een nieuw Spring project op te zetten:

- Je moest een Maven project maken.
- Je moest dependencies toevoegen aan `pom.xml` voor het Spring framework en voor andere libraries.
- Je moest heel wat Java initialisatiecode schrijven voor een website, terwijl deze initialisatiecode voor elke website dezelfde was.

Spring Boot doet deze stappen voor je, zodat je productiviteitswinst hebt.

Spring Boot heeft een website: <http://start.spring.io>. Deze

- Maakt het Maven project voor je project.
- Voegt de nodige dependencies toe aan `pom.xml`.

Je gebruikt deze website om een project te maken voor een website voor je klant Pizza Luigi.

1. Je surft naar <http://start.spring.io>.
2. Je kiest Switch to the full version.
3. Je tikt `be.vdab` bij Group.
4. Je tikt `pizzaluigi` bij Artifact.
5. Je kiest `War` bij Packaging.
6. Je plaatst een vinkje bij Web.
`pom.xml` zal dan de dependencies die horen bij een website bevatten.
`pom.xml` zal ook de dependencies voor een embedded Tomcat webserver bevatten.
Je kan een ander merk webserver gebruiken. Je moet dan manueel `pom.xml` aanpassen.
Dit valt buiten het bereik van deze cursus.
7. Je plaatst een vinkje bij DevTools.
Telkens je iets aan de website wijzigt, herstart DevTools vliegensvlug de website.
Je ziet onmiddellijk het resultaat van je wijzigingen in de browser zonder dat je zelf de website moet herstarten.
8. Je kiest Generate Project.

Je importeert dit project in Eclipse:

1. Je kopieert in de Windows File Explorer de map `pizzaluigi` uit het zip bestand naar je workspace map van Eclipse.
2. Je kiest in Eclipse in het menu File de opdracht Import.
3. Je kiest Maven, Existing Maven Projects.
4. Je kiest Next.
5. Je kiest Browse.
6. Je kiest de map `pizzaluigi` en je kiest OK.
7. Je kiest Finish.

Eclipse zal nu een tijdje (tot enkele minuten) werk hebben om het project te importeren.

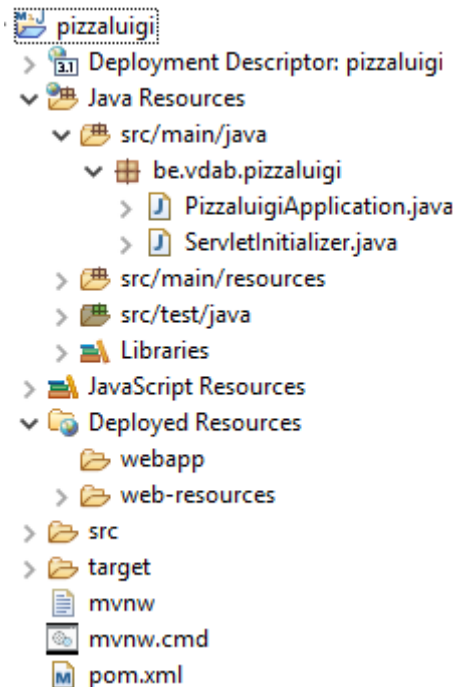
Je kan dit zien aan de meldingen rechts onder in het Eclipse venster.

Je opent het Java EE perspective. Een perspective is een verzameling vensters geopend in Eclipse. Standaard staat het Java perspective open. Het Java perspective is enkel handig als je *eenvoudige* Java applicaties ontwikkelt. Het Java EE perspective is handig bij het ontwikkelen van *enterprise* applicaties in Java. Een website valt ook onder de categorie *enterprise* applicatie.

1. Je kiest in het menu Window, Perspective, Open Perspective, Other, Java EE.
2. Je kiest Open.

Het project bevat directories volgens de Maven standaard.

Het project bevat ook het Maven projectbestand pom.xml:



- src/main/java
bevat Java sources.
- PizzaluigiApplication.
bevat de opstartmethod
public static void main(String[] args).
Voor de class staat @SpringBootApplication.
Deze annotatie initialiseert Spring Boot
bij het starten van de website.
- ServletInitializer
Spring gebruikt deze class
als je de website installeert op een webserver.
- src/main/resources
bevat configuratiebestanden.
- src/test/java
bevat testen.
- webapp
bevat web onderdelen (HTML bestanden, CSS
bestanden, JavaScript bestanden, afbeeldingen).

Het is belangrijk, voor de correcte werking van je website, je eigen classes te maken in subpackages van de package (be.vdab.pizzaluigi) van die class.

Een voorbeeld van een correcte subpackage is be.vdab.pizzaluigi.entities.

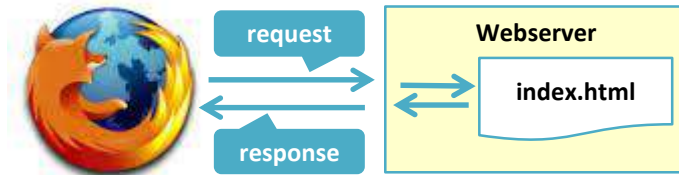
Een voorbeeld van een verkeerde subpackage is org.myorg.entities.

7 EERSTE WEBSITE MET STATIC CONTENT

7.1 Statische pagina

Een statische pagina (bijvoorbeeld `index.html`) is een bestand op de webserver.

Bij een request naar zo'n pagina, stuurt de webserver de inhoud van dit bestand als response:



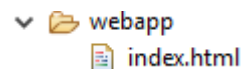
7.2 Welkompagina

De pagina met de naam `index.html` speelt de rol van welkompagina van je website.

Als je naar de website surft zonder een pagina te vermelden (je surft bijvoorbeeld naar `pizzaluigi.be`), zie je in de browser deze welkompagina.

7.2.1 Pagina

1. Je klikt met de rechtermuisknop op webapp.
2. Je kiest New, HTML file.
3. Je tikt `index` bij File name.
4. Je kiest Finish.



Je wijzigt deze pagina:

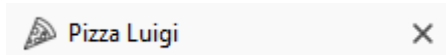
```

<!doctype html>
<html lang='nl'>
  <head>
    <meta charset='UTF-8'>
    <title>Pizza Luigi</title>
    <link rel='icon' href='images/pizza.ico' type='image/x-icon'>
    <meta name='viewport' content='width=device-width,initial-scale=1'>
    <link rel='stylesheet' href='css/pizzaluigi.css'>
  </head>
  <body>
    <h1>Pizza Luigi</h1>
    <img src='images/pizza.jpg' alt='pizza' class='fullwidth'>
  </body>
</html>

```

①
②

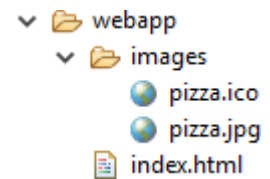
- (1) Je toont met deze regel de afbeelding `pizza.ico` in het tabblad van de browser waarin de website geopend is:



- (2) Sommige smartphones en tablets tonen een uitgezoomd beeld van de pagina's van je website. Ze hopen zo de volledige inhoud van die pagina's op één venster te kunnen tonen. De gebruiker moet dan inzoomen om de pagina weer op oorspronkelijke grootte te zien. Je verhindert dit uitzoomen met de huidige regel.

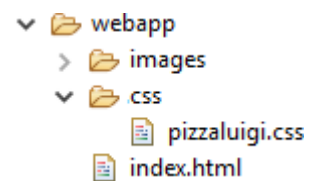
7.2.2 Afbeeldingen

1. Je klikt met de rechtermuisknop op webapp.
2. Je kiest New, Folder.
3. Je tikt images bij Folder name.
4. Je kiest Finish.
5. Je selecteert in de Windows File Explorer pizza.ico en pizza.jpg bij het theoriemateriaal bij de cursus.
6. Je klikt één afbeelding aan met de rechtermuisknop.
7. Je kiest Copy.
8. Je klikt in Eclipse met de rechtermuisknop op images.
9. Je kiest Paste.



7.2.3 CSS

1. Je klikt met de rechtermuisknop op webapp.
2. Je kiest New, Folder.
3. Je tikt css bij Folder name.
4. Je kiest Finish.
5. Je klikt in de Windows File Explorer met de rechtermuisknop op pizzaluigi.css (bij de cursus) en je kiest Copy.
6. Je klikt in Eclipse met de rechtermuisknop op css.
7. Je kiest Paste.



7.2.4 Uittesten

Je klikt in de Project Explorer met de rechtermuisknop op de class PizzaluigiApplication en je kiest Run As, Java Application. Je ziet in het venster Console, onder in Eclipse, enkele diagnostische boodschappen bij het starten van je website.

Je surft in een browser naar het adres van je website: localhost:8080.

Als je index.html wijzigt, volstaat het deze source op te slaan en in de browser F5 (refresh) te drukken om het resultaat van de wijziging te zien.




Je commit de sources. Je publiceert op GitHub.

7.2.5 WAR

Als de website af is, kan je alle bestanden verpakken in een WAR bestand.

Je kan daarmee de website op een webserver installeren die als container draait.

1. Je klikt in de Project Explorer met de rechtermuisknop op het project.
2. Je kiest Export, War file.
3. Je duidt met Browse een directory waarin Eclipse het WAR bestand plaatst en je kiest Finish.

Je stopt de website, want je zal in de takenbundel een andere website maken die ook TCP/IP poort 8080 wil gebruiken): je drukt op  (rechts naast de tabbladen onder in Eclipse).



Frituur frida: zie takenbundel

8 CONTROLLER

Een controller is een Java object dat browser requests binnenkrijgt en responses terugstuurt.

Je associeert een controller met een URL in je website. Als je een controller bijvoorbeeld associeert met de URL producten, zal deze controller requests verwerken naar pizzaluigi.be/producten.

Een speciale URL is de URL /. Deze staat voor de welkompagina.

Een controller is een 1° voorbeeld van een Spring bean.

Bij een Spring bean maak je zelf geen object. Spring maakt het object.

Bij een controller is het ook Spring die er voor zorgt dat dit object browser requests verwerkt.

Je verwijdt eerst index.html.

Je maakt een Controller. Hij verwerkt requests naar de welkompagina. Hij stuurt, naargelang het moment van de dag Goede morgen, Goede middag of Goede avond naar de browser.

1. Je klikt met de rechtermuisknop op src/main/java in de Project Explorer.
2. Je kiest New, Class.
3. Je tikt be.vdab.pizzaluigi.web bij Package.
4. Je tikt IndexController bij Name.
5. Je kiest package bij Modifiers.
6. Je kiest Finish.

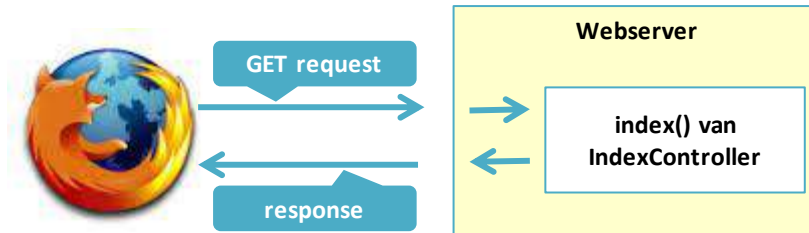
Je wijzigt deze source:

```
package be.vdab.pizzaluigi.web;
// hier komen enkele imports. Eclipse voegt ontbrekende imports toe
// aan je source met de sneltoetsen shift+ctrl+o.
@RestController
@RequestMapping("/")
class IndexController {
    @GetMapping
    String index() {
        int uur = LocalTime.now().getHour();
        if (uur < 12) {
            return "Goede morgen";
        }
        if (uur < 18) {
            return "Goede middag";
        }
        return "Goede avond";
    }
}
```

①
②
③
④
⑤
⑥

- (1) Je tikt @RestController voor een class die dient als controller.
- (2) Je associeert met @RequestMapping de controller met een URL in je website.
Dit is hier de URL die staat voor de welkompagina.
- (3) De naam van een controller class is vrij te kiezen. Deze class moet niet public zijn, ze mag ook (zoals hier) de package visibility hebben. Zoals je variabelen niet meer visibility geeft dan nodig, geef je ook classes niet meer visibility dan nodig.
- (4) Je tikt @GetMapping voor een method die browser GET requests verwerkt.
- (5) De naam van de method is vrij te kiezen. De method moet een String teruggeven.
De method moet niet public zijn, ze mag ook package visibility hebben.
- (6) De returnwaarde van de method is de response die Spring naar de browser terugstuurt.


Overzicht:



8.1 Automatisch herstarten bij wijzigingen

Als je de source opslaat, zie je in het venster Console dat de website vliegensvlug herstart. Het volstaat daarna in je browser F5 (refresh) te drukken om de nieuwe welkompagina te zien.

Telkens je een Java source van je website wijzigt en opslaat herstart de website vliegensvlug. Je moet dus de website niet zelf stoppen en starten.

Als je de website wil stoppen (omdat je bijvoorbeeld een andere website maakt die ook TCP/IP poort 8080 wil gebruiken) druk je op  (rechts naast de tabbladen onder in Eclipse).



Je commit de sources. Je publiceert op GitHub.

8.2 Nadelen van de huidige manier van werken

De String bij (6) is kort, rudimentair en geen officiële HTML. De Ststring zou moeten beginnen met `<!doctype html>` en zou een heleboel HTML tags moeten bevatten.

We stuurden op deze manier responses naar de browser enkel als kennismaking met controllers. Deze manier heeft volgende nadelen:

- ➖ Eclipse valideert geen HTML tussen “ en “.
- ➖ Als een web designer de HTML verfijnt, is de kans groot dat hij per ongeluk fouten aanbrengt in de Java code.
- ➖ Je mengt voortdurend Java code en HTML. De HTML en de Java code zijn zo minder leesbaar.
- ➖ Als je de HTML wil wijzigen, moet je een Java source wijzigen, van de applicatie een WAR bestand maken en dit WAR bestand installeren op de webserver. Dit is omslachtig.

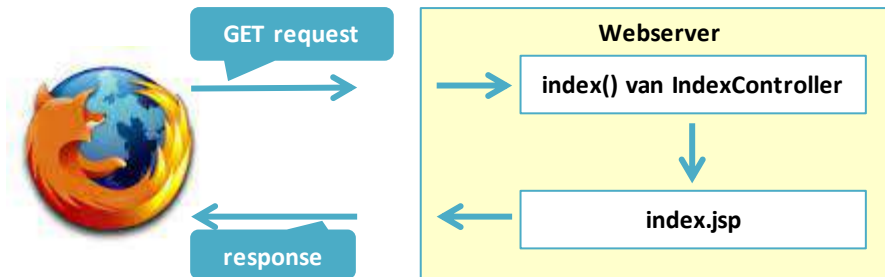


Sluitingsdagen: zie takenbundel

9 CONTROLLER EN JSP COMBINEREN

JSP is een afkorting voor Java Server Page. Het heeft dezelfde opbouw als een HTML pagina, maar je kan er ook eenvoudige code onderdelen (if, for, ...) in maken.

Vanaf nu zullen een controller en een JSP samenwerken om een browser request te verwerken en een response naar die browser terug te sturen. De request komt eerst binnen in de controller. Deze voert de nodige Java code uit om deze request te verwerken. Dit kan bijvoorbeeld JDBC code zijn om producten uit de database te lezen. Daarna geeft de controller de request door aan de JSP. De verantwoordelijkheid van die JSP is een response met HTML naar de browser te sturen.



9.1 pom.xml

Je voegt de dependencies, die horen bij JSP, toe aan pom.xml:

```

<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>[2.2,]</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>
  
```

1

(1) Je vermeldt bij deze dependency geen version.

Je pom verwijst via <parent> naar een pom van de ingenieurs van het Spring framework.

Je pom erft zo alles wat in die "parent" pom gedefinieerd is. In die "parent" pom is al een correct version nummer van veel populaire dependencies gedefinieerd.

Zo moet jij zelf van die dependencies geen version nummer meer definiëren.

Je stopt de website. Je doet dit best telkens je pom.xml wijzigt.

9.2 JSP

Je moet JSP's plaatsen in de map webapp of in een submap van webapp. Als je een JSP in webapp plaatst kan je met je browser rechtstreeks naar die JSP surfen zonder dat die request eerst door een controller verwerkt werd. Het is echter de bedoeling dat een request altijd eerst door een controller verwerkt wordt, pas daarna door een JSP. Je plaatst daartoe de JSP in een speciale submap WEB-INF. De inhoud van deze submap is niet rechtstreeks bereikbaar de de browser, maar wel bereikbaar door controllers.

Je maakt de map WEB-INF in webapp:

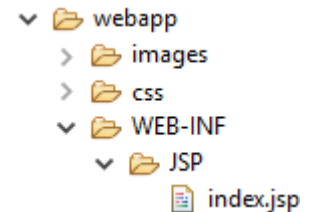
1. Je klikt met de rechtermuisknop op webapp.
2. Je kiest New, Folder
3. Je tikt WEB-INF bij Folder name.
4. Je kiest Finish.

Gezien de map WEB-INF nog andere bestanden dan JSP's kan bevatten, maak je in WEB-INF een submap speciaal voor de JSP's: JSP.

1. Je klikt met de rechtermuisknop op WEB-INF.
2. Je kiest New, Folder
3. Je tikt JSP bij Folder name.
4. Je kiest Finish.

Je maakt index.jsp in JSP:

1. Je klikt met de rechtermuisknop op JSP.
2. Je kiest new, JSP File.
3. Je tikt index bij File name.
4. Je kiest Finish.



Je wijzigt index.jsp:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%> ❶
<!doctype html>
<html lang='nl'>
  <head>
    <title>Pizza Luigi</title>
    <link rel='icon' href='images/pizza.ico' type='image/x-icon'> ❷
    <meta name='viewport' content='width=device-width,initial-scale=1'>
    <link rel='stylesheet' href='css/pizzaluigi.css'>
  </head>
  <body>
    <h1>Pizza Luigi</h1>
    <img src='images/pizza.jpg' alt='pizza' class='fullwidth'>
    <h2>Goede dag</h2>
  </body>
</html>
```

(1) Je stelt met deze regel drie eigenschappen in:

- a. Als contentType text/html bevat en pageEncoding UTF-8, bevat de response een header Content-type met de waarde text/html; charset=UTF-8. UTF-8 encoding kan alle tekens van alle menselijke talen korrekt voorstellen.
- b. De webserver maakt bij de JSP standaard code om session variabelen bij te houden. Een session variabele is een variabele die de webserver per gebruiker bijhoudt (zoals een winkelmandje op een shopping website). session='false' geeft aan dat deze JSP geen session variabelen nodig heeft. Dit bevordert de performantie.

(2) Je ziet hier een *relatief* pad: images/pizza.ico.

Spring zoekt dit relatief pad niet ten opzichte van de plaats van het JSP bestand.

Spring zal dit relatief pad zoeken ten opzichte van de URL van de huidige request (/).

Spring associeert deze URL met de folder webapp.

9.3 Controller bean

Je wijzigt IndexController:

```
package be.vdab.pizzaluigi.web;

@Controller
@RequestMapping("/")
class IndexController {
    @GetMapping
    String index() {
        return "/WEB-INF/JSP/index.jsp";
    }
}
```

①

②

- (1) Als een controller samenwerkt met een JSP om een browser request te verwerken, tik je voor de class `@Controller` in plaats van `@RestController`.
- (2) De returnwaarde van de method, die de browser request verwerkt, is de plaats en de naam van de JSP die de response naar de browser zal sturen.

Je kan de website terug starten en uitproberen.



Je commit de sources. Je publiceert op GitHub.

9.4 Data doorgeven van de controller naar de JSP

De JSP toont nu Goede dag aan de gebruiker. Je zal dit terug wijzigen naar Goede morgen, Goede middag of Goede avond, gebaseerd op het uur. Je doet dit met volgende stappen:

1. De controller maakt een String met Goede morgen, Goede middag of Goede avond.
2. De controller geeft deze String door aan de JSP.
3. De JSP toont de inhoud van deze String.

Je wijzigt in IndexController de method index:

```
@GetMapping
ModelAndView index() {
    String boodschap;
    int uur = LocalTime.now().getHour();
    if (uur < 12) {
        boodschap = "Goede morgen";
    } else if (uur < 18) {
        boodschap = "Goede middag";
    } else {
        boodschap = "Goede avond";
    }
    return new ModelAndView("/WEB-INF/JSP/index.jsp", "boodschap", boodschap);
}
```

①

②

- (1) Een controller die data doorgeeft aan de JSP heeft als returntype `ModelAndView`.
- (2) De 1° parameter van de `ModelAndView` constructor is de naam van de JSP waaraan de controller de request doorgeeft. De 2° parameter is de naam waaronder de controller data doorgeeft. De 3° parameter is de data zelf.

Je wijzigt in `index.jsp` de regel `<h2>Goede dag</h2>`:

```
<h2>${boodschap}</h2>
```

①

- (1) Je leest in een JSP data die de controller doorgaf met een miniprogrammeertaal: EL (Expression Language). Elke EL expressie begint met `${` en eindigt op `}`. Je leest de inhoud van de data met de naam `boodschap` met `${boodschap}`



Je commit de sources. Je publiceert op GitHub.

Je kan de website terug uitproberen.

9.5 application.properties

Je website zal per pagina

- Een method in een controller hebben.
- En een bijbehorende JSP hebben.

Het is vervelend in elke controller method aan te geven

- dat de JSP zich bevindt in /WEB-INF/JSP.
- en dat de extensie van een JSP .jsp is.

In de plaats dit iedere keer te vermelden vermeld je het 1 keer.

Je doet dit in het configuratiebestand van je website: application.properties.

Je vindt dit bestand in de map src/main/resources.

Je voegt aan dit bestand volgende regels toe:

```
spring.mvc.view.prefix:/WEB-INF/JSP/
spring.mvc.view.suffix:.jsp
```

❶
❷

(1) Elke regel in application.properties stelt één configuratieinstelling voor.

Zo'n instelling heeft een naam (voor :) en een waarde (na :).

De instelling met de naam spring.mvc.view.prefix bevat de plaats van de JSP's ten opzichte van webapp. De laatste / is verplicht.

De structuur van een properties bestand is eenvoudiger dan die van een XML bestand.

Het is daarom makkelijker voor een niet-ontwikkelaar om dit bestand aan te passen.

(2) De instelling spring.mvc.view.suffix bevat de extensie van JSP's, inclusief het punt.

Je moet nu in IndexController de plaats en de extensie van de JSP niet meer meegeven:

```
return new ModelAndView("index", "boodschap", boodschap);
```

Je kan de website terug uitproberen.



Opmerking: als je een JSP wijzigt, moet je de website niet herstarten om het resultaat te zien. Het volstaat de JSP op te slaan en daarna in je browser het beeld van de pagina te verversen.

9.6 Meerdere data doorgeven aan de JSP

Je kan meerdere data doorgeven aan de JSP met de addObject method van ModelAndView:

```
ModelAndView modelAndView = new ModelAndView("index", "geluksgetal", 7);
modelAndView.addObject("ongeluksgetal", 13);
return modelAndView;
```

❶
❷

(1) Je geeft data met de naam geluksgetal en de inhoud 7 door aan de JSP.

Je zou dit in de JSP kunnen lezen met de EL expressie \${geluksgetal}.

(2) Je geeft data met de naam ongeluksgetal en de inhoud 13 door aan de JSP.

Je zou dit in de JSP kunnen lezen met de EL expressie \${ongeluksgetal}.

Er bestaat ook een versie van addObject waarbij je geen naam voor de data meegeeft, enkel de data zelf. De naam van de data is dan gelijk aan de class van die data, waarbij de eerste letter in kleine letters staat:

```
modelAndView.addObject(BigDecimal.valueOf(3)); // naam data = bigDecimal
```

De method addObject geeft hetzelfde ModelAndView object terug waarop je addObject uitvoerde.

Dit laat korte code toe in je controllers:

```
return new ModelAndView("index", "geluksgetal", 7)
    .addObject("ongeluksgetal", 13);
```



Sluitingsdagen 2: zie takenbundel

10 \${EXPRESSION LANGUAGE}

10.1 Primitief datatype

EL neemt de tekstvoorstelling van data met een primitief datatype op in de response

Bij data met de naam kinderen met de waarde 3, maakt de JSP code

Luigi's `${kinderen}` kinderen de HTML Luigi's 3 kinderen

10.2 Object

EL voert op een object in de method `toString` uit en neemt het resultaat op in de response.

Als de controller method als volgt is:

```
return new ModelAndView("index", "geluksgetal", BigDecimal.valueOf(7));
```

voert de EL expressie `${geluksgetal}` de `toString` method uitvoeren op de `BigDecimal`

en deze String opnemen in de response.

10.3 Onbestaande data

Als je verwijst naar een onbestaande data, maakt EL een lege tekst.

10.4 Hard gecodeerde waarden

gehele getallen	<code>\${7}</code>	strings (tussen " of ')	<code>\${"james"}</code>
getallen met decimalen	<code>\${40.3399}</code>	booleans (true of false)	<code>\${false}</code>

10.5 Wiskundige operatoren

- `+` `-` `*` `/` en `%`
- **div** is synoniem voor `/` (delen), **mod** voor `%` (rest bepalen bij delen)

Als `getal` de waarde 9 bevat geeft `${getal / 2}` de waarde 4.5.

10.6 Vergelijkingsoperatoren

- `==` `!=` `>` `<` `>=` `<=`
- **eq** is synoniem voor `==`, **ne** voor `!=`, **gt** voor `>`, **ge** voor `>=`, **lt** voor `<`, **le** voor `<=`

Als `getal` de waarde 9 bevat geeft `${getal == 9}` de waarde `true`.

10.7 Logische operatoren

- `!` `&&` `||`
- **not** is synoniem voor `!`, **and** voor `&&`, **or** voor `||`

Als `getal` de waarde 9 bevat geeft `${getal > 8 && getal < 10}` de waarde `true`.

10.8 Conditionele operator ? :

Syntax: voorwaarde ? waardeAlsVoorwaardeTrue : waardeAlsVoorwaardeFalse

Als `getal` de waarde 7 bevat geeft `${getal == 7 ? "geluk" : "ongeluk"}` de waarde `geluk`.

10.9 Operator empty

Je vermeldt na **empty** een expressie. **empty** geeft `true` terug als de expressie gelijk is aan:

- **null**
- een lege String
- een lege verzameling (array, List, Set, Map)
- de naam van een onbestaande data

Als klanten een lege List bevat geeft `${empty klanten}` de waarde `true`.

10.10 Één element uit een verzameling lezen

10.10.1 Array

Je leest één array element met de syntax `${dataMetArray[indexVanHetElement]}`.

Als namen een array bevat met de elementen Joe en Averell geeft `${namen[0]}` de waarde Joe.

10.10.2 List

Als namen een List bevat met de elementen Joe en Averell geeft `${namen[0]}` de waarde Joe.

10.10.3 Map

Als eigenschappen een Map bevat met volgende entries:

key	value
Joe	driftig
Averell	hongerig

geeft `${eigenschappen["Averell"]}` de waarde hongerig.

Als de keys String zijn, bestaat een korte syntax: `${eigenschappen.Averell}`.

Opgepast, als de key een geheel getal is, moet dit in je Java code een long zijn, geen int.

10.11 Resultaat van een method oproep

Je kan met EL het resultaat van een method oproep lezen:

Als `familienaam` de waarde `dalton` bevat, geeft `${familienaam.length()}` de waarde 6.



Sluitingsdagen 3: zie takenbundel

11 JAVABEAN

Een JavaBean is een object waarvan de class voldoet aan bepaalde voorwaarden.

11.1 Getters en setters

Je maakt in de package `be.vdab.pizzaluigi.valueobjects` de class `Persoon`:

```
package be.vdab.pizzaluigi.valueobjects;
public class Persoon {
    private String voornaam;
    private String familienaam;
    private int aantalKinderen;
    private boolean gehuwd;
}
```

Persoon
-voornaam: String
-familienaam: String
-aantalKinderen: int
-gehuwd: boolean

De JavaBean standaard definieert dat je per attribuut:

- een public method maakt waarmee je de attribuut waarde kan opvragen.
 - De method naam begint met **get**. Als het attribuut boolean is, begint de naam met **is**.
 - De rest van de methodnaam is gelijk aan de naam van het bijbehorende attribuut, met de eerste letter als hoofdletter.
 - Het method returntype is gelijk aan het attribuut type.
 - De method heeft geen parameters.
 - De method returnt de attribuut waarde. Je vindt die in de bijbehorende private variabele.

Je maakt deze methods (*getters* genoemd) in de class `Persoon`:

```
public String getVoornaam() {
    return voornaam;
}
public String getFamilienaam() {
    return familienaam;
}
public int getAantalKinderen() {
    return aantalKinderen;
}
public boolean isGehuwd() {
    return gehuwd;
}
```

- een public method maakt waarmee je een waarde kan invullen in het attribuut.
 - De method naam begint met **set**.
 - De rest van de methodnaam is gelijk aan de bijbehorende attribuut naam, met de eerste letter als hoofdletter.
 - Het method returntype is `void`.
 - De method heeft één parameter: de waarde die je wilt invullen in het attribuut. Het parameter type is gelijk aan het attribuut type.

Je maakt deze methods (*setters* genoemd) in de class `Persoon`:

```
public void setVoornaam(String voornaam) {
    this.voornaam = voornaam;
}
public void setFamilienaam(String familienaam) {
    this.familienaam = familienaam;
}
public void setAantalKinderen(int aantalKinderen) {
    this.aantalKinderen = aantalKinderen;
}
public void setGehuwd(boolean gehuwd) {
    this.gehuwd = gehuwd;
}
```

11.2 ReadOnly attributen

Een attribuut dat enkel een getter (en geen setter) heeft, is een readonly attribuut: een attribuut dat je kan lezen, maar niet kan wijzigen. Voorbeeld:

```
public String getNaam() {
    return voornaam + ' ' + familienaam;
}
```

Je ziet dat een readonly attribuut niet altijd een eigen private variabele nodig heeft.

11.3 Getters en Setters maken met Eclipse

Je verwijdert eerst de getters en setters in de class Persoon.

1. Je opent de source van de class Persoon.
2. Je kiest in het menu Source de opdracht Generate Getters and Setters.
3. Je kiest de knop Select All en daarna OK.



Opmerking: Eclipse maakt het gemakkelijk om getters en setters te genereren.

Het is niet verstandig voor alle private variabelen setters te genereren.

Dit stemt niet overeen met de werkelijkheid. Een class Rekening (van een Bank) heeft bijvoorbeeld geen method setSaldo (hoe plezant deze method ook zou zijn).

Je wijzigt het saldo enkel in andere methods: storten, afhalen en overschrijven.

11.4 Constructors

Een JavaBean moet een public default constructor hebben. Dit is momenteel het geval.

Jij tikte geen constructor. De compiler maakt dan een public default constructor.

Je tikt nu een geparametriseerde constructor. De compiler maakt dan geen default constructor meer. Je moet dus ook de default constructor zelf tikken.

```
public Persoon(String voornaam, String familienaam, int aantalKinderen,
    boolean gehuwd) {
    this.voornaam = voornaam;
    this.familienaam = familienaam;
    this.aantalKinderen = aantalKinderen;
    this.gehuwd = gehuwd;
}
public Persoon() { // default constructor
}
```



Opmerking: JSP is niet zo streng als de JavaBean standaard:

er mag, maar moet geen default constructor zijn. Getters zijn verplicht, setters niet.

11.5 Constructors maken met Eclipse

Je verwijdert eerst de constructors in de class Persoon.

1. Je opent de source van de class Persoon.
2. Je kiest in het menu Source de opdracht Generate Constructor using Fields.
3. Je plaatst een vinkje bij Omit call to default constructor super().
4. Je kiest Deselect all en je kiest OK.
Je krijgt een default constructor.
5. Je herhaalt stappen 1, 2 en 3.
6. Je kiest OK.
Je krijgt een geparametriseerde constructor.

11.6 EL en JavaBeans

Als data een JavaBean object bevat, lees je met EL een attribuut waarde van dat object met de syntax `${requestAttribuut.attribuut}`.

Als de data persoon een Persoon object bevat, lees je bijvoorbeeld:

- de naam via `${persoon.naam}`
EL vertaalt naam naar een method oproep `getNaam()` op het object persoon.
- het aantal kinderen via `${persoon.aantalKinderen}`
- de gehuwde staat via `${persoon.gehuwd}`

Je vervangt de laatste ; in de method index van IndexController:

```
.addObject("zaakvoerder", new Persoon("Luigi", "Peperone", 7, true));
```

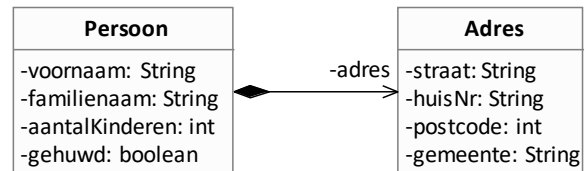

Je tikt in `index.jsp` voor `</body>`:

```
<h2>De zaakvoerder</h2>
<dl>
  <dt>Naam</dt><dd>${zaakvoerder.naam}</dd>
  <dt>Aantal kinderen</dt><dd>${zaakvoerder.aantalKinderen}</dd>
  <dt>Gehuwd</dt><dd>${zaakvoerder.gehuwd ? 'Ja' : 'Nee'}</dd>
</dl>
```

Je kan de website terug uitproberen.

11.7 Geneste attributen

Een attribuut kan een object zijn, dat zelf attributen bevat. Voorbeeld: de class `Persoon` heeft een attribuut `adres`. Het type is de class `Adres`, met de attributen `straat`, `huisNr`, `postcode` en `gemeente`.



Je spreekt een genest attribuut aan als `${requestAttribuut.attribuut.genestAttribuut}`.

Als een attribuut `persoon` een `Persoon` object bevat, spreek je met volgende expressie het genest attribuut `straat` aan: `${persoon.adres.straat}`.

Je maakt in `be.vdab.pizzaluigi.valueobjects` een class `Adres`:

```
package be.vdab.pizzaluigi.valueobjects;
public class Adres {
    private String straat;
    private String huisNr;
    private int postcode;
    private String gemeente;
}
```

Je maakt zelf getters voor de private variabelen met Eclipse.

Je maakt zelf een geparametriseerde constructor met Eclipse.

Je voegt code toe aan de class `Persoon`:

```
private Adres adres; // Je maakt zelf een bijbehorende getter en setter
```

Je voegt een parameter toe aan de geparametriseerde constructor van `Persoon`:

```
public Persoon(..., Adres adres) {
```

Je voegt een opdracht toe in deze constructor:

```
this.adres = adres;
```

Je wijzigt de code `addObject()` in de method `index` van `IndexController`:

```
.addObject("zaakvoerder", new Persoon("Luigi", "Peperone", 7, true,
    new Adres("Grote markt", "3", 9700, "Oudenaarde")));
```

Je tikt in `index.jsp` voor `</dl>`:

```
<dt>Adres</dt>
<dd>${zaakvoerder.adres.straat} ${zaakvoerder.adres.huisNr}<br>
    ${zaakvoerder.adres.postcode} ${zaakvoerder.adres.gemeente}</dd>
```

Je kan de website terug uitproberen.



Je commit de sources. Je publiceert op GitHub.



Adres: zie takenbundel

12 JSTL (JSP STANDARD TAG LIBRARY)

Je kent al HTML tags <head>, <h1>, ... De browser verwerkt die tags.

JSTL tags zijn echter server-sided tags, verwerkt door de webserver.

JSTL is één JAR bestand, met daarin enkele libraries.

12.1 Tag names en tag URI's

Elke tag heeft een naam.

- Je itereert met de tag **forEach** over een verzameling.
- Je voert met de tag **if** een JSP onderdeel conditioneel uit.
- ...

Een tag library bevat meerdere tags. Een tag library bevat een unieke identificatie. Dit is een URI.

De tags uit de meest gebruikte library hebben de URI <http://java.sun.com/jsp/jstl/core>.



De URI moet geen bestaande URL op het internet te zijn. De URI wordt ook niet als een bestaande URL op het internet opgezocht tijdens het verwerken van een tag library.

De conventie is wel dat het begin van een URI een URL is. Dit garandeert dat elke URI uniek is, wat belangrijk is: de URI is de unieke identificatie van de tag library.

Als je een tag library in een JSP gebruikt,

associeer je boven in de JSP één keer de tag library URI met een eigen gekozen prefix.

Je doet dit met de `taglib` directive. Die heeft volgende syntax:

```
<%@taglib prefix="gekozenPrefix" uri="URIVanDeTagLibrary"%>
```

Je associeert standaard de prefix `c` met de URI `http://java.sun.com/jsp/jstl/core`.

```
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
```

Je verwijst in die pagina naar een tag uit de library met de prefix, dus: `<c:forEach>`, `<c:if>`

12.2 <c:forEach>

Je itereert met `<c:forEach>` over de elementen van een verzameling:

```
<c:forEach var='eenVariabele' items='${eenVerzameling}'>
```

```
...
</c:forEach>
```

`forEach` biedt bij elke iteratie het volgend element aan in de variabele `eenVariabele`.

Je probeert dit uit: je maakt een pagina waarin de gebruiker pizza's ziet.

Je maakt in `be.vdab.pizzaluigi.web` een `PizzaController`.

Die verwerkt GET requests naar de URL `pizzas`:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Controller
@RequestMapping("pizzas")
class PizzaController {
    private static final String PIZZAS_VIEW = "pizzas";
    private final List<String> pizzas =
        Arrays.asList("Prosciutto", "Margherita", "Calzone");
    @GetMapping
    ModelAndView pizzas() {
        return new ModelAndView(PIZZAS_VIEW, "pizzas", pizzas);
    }
}
```

Je maakt pizzas.jsp in WEB-INF/JSP:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<!doctype html>
<html lang='nl'>
  <head>
    <title>Pizza's</title>
    <link rel='icon' href='images/pizza.ico' type='image/x-icon'>
    <meta name='viewport' content='width=device-width,initial-scale=1'>
    <link rel='stylesheet' href='css/pizzaluigi.css'>
  </head>
  <body>
    <h1>Pizza's</h1>
    <ul class='zebra'>
      <c:forEach var='pizza' items='${pizzas}'>
        <li>${pizza}</li>
      </c:forEach>
    </ul>
  </body>
</html>
```

①

②

③

④

- (1) Je associeert de prefix c met de URI van de core library die de tag forEach bevat.
- (2) Je itereert over de List in pizzas.
De variabele pizza wijst bij elke iteratie naar een volgend element.
- (3) Je toont het huidige element.
- (4) Je sluit de forEach af.

Je kan dit uitproberen op <http://localhost:8080/pizzas>

12.2.1 Itereren over een verzameling JavaBeans

Je maakt in be.vdab.pizzaluigi.entities een class Pizza:

```
package be.vdab.pizzaluigi.entities;
import java.math.BigDecimal;
public class Pizza {
    private long id;
    private String naam;
    private BigDecimal prijs;
    private boolean pikant;
}
```

Je maakt getters en setters voor alle private variabelen

Je maakt

1. een default constructor
2. en een geparametriseerde constructor
3. een geparametriseerde constructor zonder id parameter. Je gebruikt die later in de cursus.

Je vervangt in PizzaController de variabele pizzas:

```
private final List<Pizza> pizzas = Arrays.asList(
    new Pizza(1, "Prosciutto", BigDecimal.valueOf(4), true),
    new Pizza(2, "Margherita", BigDecimal.valueOf(5), false),
    new Pizza(3, "Calzone", BigDecimal.valueOf(4), false))
```

Je vervangt in pizzas.jsp de regel \${pizza}:

```
<li>${pizza.naam} ${pizza.prijs} &euro;</li>
```

①

- (1) Je roept met \${pizza.naam} de method getNaam op van het Pizza object.
Je roept met \${pizza.prijs} de method getPrijs op van het Pizza object.

Je kan dit terug uitproberen.

12.2.2 Itereren over een Map

Je kan met `forEach` itereren over een Map. De variabele bij `var` bevat per iteratie:

- een eigenschap `key` met de key van de huidige entry `${eenVariabele.key}`
- een eigenschap `value` met de value van de huidige entry `${eenVariabele.value}`

Je vervangt in `PizzaController` de variabele `pizzas`:

```
private final Map<Long, Pizza> pizzas = new LinkedHashMap<>(); // keys:pizza ids
PizzaController() {
    pizzas.put(1L, new Pizza(1, "Prosciutto", BigDecimal.valueOf(4), true));
    pizzas.put(2L, new Pizza(2, "Margherita", BigDecimal.valueOf(5), false));
    pizzas.put(3L, new Pizza(3, "Calzone", BigDecimal.valueOf(4), false));
    pizzas.put(4L, new Pizza(4, "Fungi & Olive", BigDecimal.valueOf(5), false));
}
```

Je vervangt in `pizzas.jsp` de regels `<forEach> ... </forEach>`:

```
<c:forEach var='entry' items='${pizzas}'>
    <li>${entry.key}: ${entry.value.naam} ${entry.value.prijs}&euro;</li>
</c:forEach>
```

Je kan dit terug uitproberen

12.2.3 begin, step en end attributen

Je gebruikt de `forEach` attributen `begin`, `step` en/of `end` om elementen over te slaan:

begin	De iteratie start met het element dat dit getal als index heeft.
end	De iteratie loopt tot en met het element dat dit getal als index heeft.
step	Bij elke iteratie selecteert <code>forEach</code> het element met een index <code>step</code> hoger dan de index van het element uit de vorige iteratie.

12.2.4 Itereren zonder verzameling

Je kan met `forEach` itereren zonder bijbehorende verzameling.

De attributen `begin` en `end` geven aan hoeveel keer `forEach` itereert.

Voorbeeld: vier keer itereren: `<c:forEach var='index' begin='1' end='4'>`

Je wijzigt in `pizzas.jsp` het element `h1`:

```
<h1>Pizza's
    <c:forEach begin='1' end='5'>
        &#9733; <!-- de HTML code van een ster -->
    </c:forEach>
</h1>
```

Je kan dit terug uitproberen.

12.2.5 varStatus attribuut

Je kan aan `forEach` een attribuut `varStatus` toevoegen, met een variabelenaam

```
<c:forEach var='eenVar' items='${eenVerzameling}' varStatus='status'>
    ...
</c:forEach>
```

`status` bevat in de `forEach` iteratie volgende eigenschappen over de iteratie:

Eigenschap	Betekenis
count	volgnummer van de huidige iteratie, de nummering begint vanaf 1.
index	volgnummer van de huidige iteratie, de nummering begint vanaf 0. Als ook het attribuut <code>begin</code> is meegegeven, wordt de inhoud van <code>begin</code> bijgeteld.
first	true bij de eerste iteratie, false bij de volgende iteraties.
last	true bij de laatste iteratie, false bij de vorige iteraties.

Bij de iteraties van de forEach

```
<c:forEach begin="4" end="6" varStatus="status">
...
</c:forEach>
```

hebben de status eigenschappen deze waarden:

<code>\${status.count}</code>	<code>\${status.index}</code>	<code>\${status.first}</code>	<code>\${status.last}</code>
1	4	true	false
2	5	false	false
3	6	false	True

12.3 <c:if>

Je voert met `<c:if>` een deel van de JSP enkel uit als een voorwaarde true is:

```
<c:if test='voorwaardeUitgedruktAlsELExpressie'>
...
</c:if>
```

Voorbeeld: je vervangt in pizzas.jsp de regel ` ... `:

```
<li>
${entry.key}: ${entry.value.naam} ${entry.value.prijs}&euro;
  <c:if test='${entry.value.pikant}'>
    pikant
  </c:if>
</li>
```

Je kan dit terug uitproberen.

`<c:if>` heeft geen else gedeelte. `<c:choose>` is daartoe een oplossing

12.4 <c:choose>

Deze tag is vergelijkbaar met het switch statement uit Java:

```
<c:choose>
  <c:when test='${voorwaarde1}'>
    Dit deel wordt enkel uitgevoerd als voorwaarde1 true is
  </c:when>
  <c:when test='${voorwaarde2}'>
    Dit deel wordt enkel uitgevoerd als voorwaarde2 true is
  </c:when>
  <c:otherwise>
    Dit deel wordt uitgevoerd als alle voorwaarden bij <c:when> false zijn.
  </c:otherwise>
</c:choose>
```

Voorbeeld: je vervangt in pizzas.jsp de regels `<c:if ...> ... </c:if>`:

```
<c:choose>
  <c:when test='${entry.value.pikant}'>
    pikant
  </c:when>
  <c:otherwise>
    niet pikant
  </c:otherwise>
</c:choose>
```

Je kan dit terug uitproberen.

Je kan soms een if ... else kort uitdrukken met de EL conditionele operator `? :`

`<c:choose ...> ... </c:choose>` wordt `${entry.value.pikant ? "pikant" : "niet pikant"}`

12.5 <c:out>

Je moet sommige tekens als een character entity code in HTML opnemen: & als &

<c:out> doet die vertaalslag: `<c:out value='tekst_of_ELExpressie' />`

Je voegt in PizzaController een pizza toe, met een naam die & bevat:

```
pizzas.put(23L, new Pizza(23, "Fungi & Olive", BigDecimal.valueOf(5), false));
```

Je vervangt in pizzas.jsp `${entry.value.naam}` door `<c:out value='${entry.value.naam}' />`

Je kan dit terug uitproberen.

Je klikt met de rechtermuisknop in de pagina in de browser en je kiest View (page) source.

Je ziet dat Fungi & Olive vervangen is door het correcte *Fungi & Olive*

12.6 <c:url>

Je maakt met <c:url> een URL.

Je kan achteraan op de URL parameters meegeven (de query string):

```
<c:url value='url' var='resultaatURL'>
  <c:param name='naamEersteParameter' value='waardeEersteParameter' />
  <c:param name='naamTweedeParameter' value='waardeTweedeParameter' />
</c:url>
```

Je maakt bij elke pizza een hyperlink naar een detailpagina over die pizza.

De URL in de hyperlink bevat een parameter met het pizza id.

Bij pizza 2 is de URL /pizzas?id=2

Je tikt voor :

```
<c:url value='/pizzas' var='url'>
  <c:param name='id' value='${entry.key}' />
</c:url>
<a href='${url}'>Detail</a>
```

Je kan dit terug uitproberen.

Als je de muisaanwijzer laat rusten op een Detail hyperlink,

zie je onder in de browser de URL die <c:url> opbouwde, inclusief de parameter id.

12.7 <c:import>

Je tikt code, die je nodig hebt in meerdere JSP's, in een kleine JSP, die je daarna met <c:import> importeert in die andere JSP's.

Syntax: `<c:import url='LocatieEnNaamVanDeTeImporterenJSP' />`

Je maakt menu.jsp in WEB-INF/JSP:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<header>
<nav>
<ul>
<li><a href="<c:url value='/' />">Welkom</a></li>
<li><a href="<c:url value='/pizzas' />">Pizza's</a></li>
<li><a href="<c:url value='/pizzas/vantotprijs' />">Van tot prijs</a></li>
<li><a href="<c:url value='/pizzas/prijzen' />">Prijzen</a></li>
<li><a href="<c:url value='/pizzas/toevoegen' />">Toevoegen</a></li>
<li><a href="<c:url value='/mandje' />">Mandje</a></li>
<li><a href="<c:url value='/identificatie' />">Identificatie</a></li>
<li><a href="<c:url value='/headers' />">Headers</a></li>
</ul>
</nav>
</header>
```

Je importeert menu.jsp in pizzas.jsp, na <body>:

```
<c:import url='/WEB-INF/JSP/menu.jsp'/>
```

De eerste slash verwijst naar de map webapp van je project.

Je tikt als tweede regel in index.jsp:

```
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
```

Je tikt na <body>:

```
<c:import url='/WEB-INF/JSP/menu.jsp'/>
```

Je kan de website uitproberen.

12.7.1 Parameters

Een import bestand kan ook parameters hebben.

Je maakt head.jsp in WEB-INF/JSP:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<title>${param.title}</title>
<link rel='icon' href='images/pizza.ico' type='image/x-icon'>
<meta name='viewport' content='width=device-width,initial-scale=1'>
<link rel='stylesheet' href='css/pizzaluigi.css'>
```

①

(1) Je verwijst naar een parameter title, die je zal meegeven bij het importeren.

Je wijzigt in index.jsp de regels tussen <head> en </head>:

```
<c:import url='/WEB-INF/JSP/head.jsp'>
  <c:param name='title' value='Pizza Luigi'>
</c:import>
```

①

(1) Je vult de parameter title met de waarde Pizza Luigi.

Je wijzigt in pizzas.jsp de regels tussen <head> en </head>:

```
<c:import url='/WEB-INF/JSP/head.jsp'>
  <c:param name='title' value="Pizza's"/>
</c:import>
```

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.

12.8 <c:set>

Je vult een variabele in met <c:set>. Je kan daarna in de rest van de JSP de variabele gebruiken.

Syntax: <c:set var='naamVandeVariabele' value='inhoudVanDeVariabele'/>

- Je mag bij value een EL expressie vermelden.
- Als de variabele nog niet bestaat, maakt <c:set> de variabele aan.
- Als de variabele al bestaat, overschrijft <c:set> de inhoud van de variabele.
- Je leest daarna de inhoud van de variabele met de EL expressie \${naamVanDeVariabele}.

Voorbeeld:

```
<c:set var='geluk' value='${7*70}'/>
```



Sauzen: zie takenbundel

13 RELATIEVE URL'S

Een relatieve URL is een URL die niet begint met `/`.

`head.jsp` bevat een relatieve URL: `css/pizzaluigi.css`. De browser interpreteert die als volgt:

- Hij neemt huidige request URL.
Dit is `http://localhost:8080/pizzas`
- Hij verwijdert de tekens na de laatste `/`.
Hij behoudt `http://localhost:8080/`.
- Hij voegt de relatieve URL toe:
`http://localhost:8080/css/pizzaluigi.css`

De browser leest op die URL de stylesheet. Dit lukt momenteel.

13.1 Probleem

Relatieve URL's kunnen problemen veroorzaken.

Je wijzigt de URL van `PizzaController` naar `/pizzas/all` om dit te zien:

```
@RequestMapping("pizzas/all")
```

Je voert de website uit op `http://localhost:8080/pizzas/all`

Het element `<h1>` is niet meer gekleurd, omdat de browser `pizzaluigi.css` niet vindt:

- Hij neemt de huidige request URL:
`http://localhost:8080/pizzas/all`
- Hij verwijdert de tekens na de laatste `/`:
`http://localhost:8080/pizzas/`
- Hij voegt de relatieve URL toe:
`http://localhost:8080/pizzas/css/pizzaluigi.css`

De browser vindt op die URL de CSS niet: `/pizzas` is overbodig in de URL

13.2 Oplossing

Je wijzigt `head.jsp`

- Je voegt na de eerste regel een regel toe:
`<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>`
- Je wijzigt `'images/pizza.ico'` naar:
`'<c:url value="/images/pizza.ico"/>'`
Opgelet, de `c:url` tag werkt enkel correct als de waarde die je meegeeft bij `value` begint met `/`.
- Je wijzigt `'css/pizzaluigi.css'` naar:
`'<c:url value="/css/pizzaluigi.css"/>'`
- Je wijzigt in `index.jsp` `'images/pizza.jpg'` naar:
`'<c:url value="/images/pizza.jpg"/>'`

Je kan de website uitproberen.

Je wijzigt de URL van `PizzaController` terug naar `pizzas`: `@RequestMapping("pizzas")`



Je leert in het hoofdstuk "Session scoped beans" dat je *elke* URL best met `<c:url>` maakt (niet enkel relatieve URL's, maar ook absolute URL's die beginnen met `/`).



Je commit de sources. Je publiceert op GitHub.

14 PARAMETERS IN DE QUERY STRING

Je opent de pagina met de pizza's in je browser.

Je laat de muisaanwijzer rusten op een Detail hyperlink.

Je ziet onder in de browser de URL waar deze hyperlink naar wijst.

Deze URL bevat achteraan een query string met één parameter: id.

Je leert hier hoe je de request verwerkt als de gebruiker deze hyperlink aanklikt en hoe je de inhoud van de parameter id leest.

Je maakt code in PizzaController:

```
private static final String PIZZA_VIEW = "pizza";
@GetMapping(params = "id")
ModelAndView pizza(long id) {
    ModelAndView modelAndView = new ModelAndView(PIZZA_VIEW);
    if (pizzas.containsKey(id)) {
        modelAndView.addObject(pizzas.get(id));
    }
    return modelAndView;
}
```

①
②

- (1) Je geeft aan dat de method pizza GET requests naar de URL pizzas verwerkt, maar enkel als de query string een parameter id bevat.
- (2) Je wil de inhoud van die parameter id kennen. Het volstaat daartoe een parameter met dezelfde naam aan je Java method toe te voegen. Spring zal de inhoud van de parameter id in de query string overbrengen naar deze method parameter.

Je maakt pizza.jsp in WEB-INF/JSP:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<!doctype html>
<html lang='nl'>
    <head>
        <c:import url='/WEB-INF/JSP/head.jsp'>
            <c:param name='title' value='${pizza.naam}'/>
        </c:import>
    </head>
    <body>
        <c:import url='/WEB-INF/JSP/menu.jsp'/>
        <c:if test='${empty pizza}'>
            <h1>Pizza niet gevonden</h1>
        </c:if>
        <c:if test='${not empty pizza}'>
            <h1>${pizza.naam}</h1>
            <dl><dt>Nummer</dt><dd>${pizza.id}</dd>
                <dt>Naam</dt><dd>${pizza.naam}</dd>
                <dt>Prijs</dt><dd>${pizza.prijs}</dd>
                <dt>Pikant</dt><dd>${pizza.pikant ? 'ja' : 'nee'}</dd>
            </dl>
        </c:if>
    </body>
</html>
```

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.

15 CLEAN URL'S MET PATH VARIABLEN

Het nummer van de pizza is de unieke identificatie van die pizza.

Een moderne website gebruikt clean URL's.

In een clean URL staat een unieke identificatie niet in de query string, maar in de URL zelf.

De URL, waarop je pizza 1 toont, wijzigt dus van `/pizzas?id=1` naar `/pizzas/1`.

Clean URL's zijn leesbaar, vlot te tikken, gemakkelijk te onthouden en scoren goed bij zoekrobots.

Een clean URL heeft een logische opbouw.

Je leest www.pizzaluigi.be/pizzas/1 (van links naar rechts) als:

van het volledige internet (www) neem je de website pizzaluigi.be. Daarin neem je de pizza's.

Daarvan neem je pizza 1. Je verfijnt dus geleidelijk van links naar rechts hetgeen je zoekt.

Een variabeel onderdeel (1) in de URL (`/pizzas/1`) heet een path variabele.

Een URI template stelt een URI met path variabelen voor.

Je geeft de path variabelen in een URI template een naam tussen accolades: `/pizzas/{id}`

Je maakt op basis van een URI template een URI door de path variabelen te vullen met waarden.

15.1 Clean URL's maken in je JSP

Je wijzigt in `pizzas.jsp` de URL die hoort bij `Detail` naar een clean URL.

De URI template is `/pizzas/{id}`.

Je voegt boven in de pagina een verwijzing toe naar de Spring tag library:

```
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
```

Je vervangt `<c:url ...> ... </c:url>` door

```
<spring:url value='/pizzas/{id}' var='url'>                                ❶
  <spring:param name='id' value='${entry.key}'/>                          ❷
</spring:url>
```

- (1) De tag `url` maakt een clean URL. `value` bevat de bijbehorende URI template.
- (2) Je vult de path variabele in de URI template (1) in. `name` bevat naam van de path variabele, `value` bevat de waarde voor de path variabele.
Als de pizza de id 1 heeft, bevat de variabele `url` de URL `/pizzas/1`.
Als de URI template meerdere path variabelen bevat, schrijf je meerdere regels zoals (2).

15.2 Clean URL verwerken in een @GetMapping method

Je wijzigt in `PizzaController` de declaratie van de method `pizza`:

```
@GetMapping("/{id}")                                                    ❶
ModelAndView pizza(@PathVariable long id) {                             ❷
  ...
}
```

- (1) Bij de class staat `@RequestMapping("pizzas")`.
De class verwerkt dus requests naar URL's die beginnen met `/pizzas`.
Je voegt hier `{id}` aan toe. Je bekomt zo de URI template `/pizzas/{id}`.
De method `pizza` verwerkt zo requests naar URL's die passen bij `/pizzas/{id}`.
- (2) Je tikt voor een method parameter `@PathVariable`.
Spring vult die method parameter met de waarde van de path variabele met dezelfde naam (`id`) in de URL van de binnengekomen request.

Je kan de website terug uitproberen.



Je commit de sources. Je publiceert op GitHub.

15.3 URL met meerdere path variabelen

Een URL kan meerdere path variabelen bevatten.

Volgende fictieve URL geeft de verkoopstatistieken van een bepaalde maand in een bepaald jaar: /verkoopstatistieken/2015/10.

De bijbehorende URI template is /verkoopstatistieken/{jaar}/{maand}

De bijbehorende @GetMapping method:

```
@GetMapping(path = "{jaar}/{maand}")
ModelAndView read(@PathVariable int jaar, @PathVariable int maand) {
    ...
}
```

16 UNIT TEST VAN EEN CONTROLLER

Spring is zo gemaakt dat je elke bean (dus ook een controller) gemakkelijk kan unit testen.

Je schrijft als voorbeeld een unit test van `PizzaController`.

Je maakt in `src/test/java` een class `PizzaControllerTest`:

```
package be.vdab.pizzaluigi.web;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
// enkele imports
public class PizzaControllerTest {
    private PizzaController controller;
    @Before
    public void before() {
        controller = new PizzaController();
    }
    @Test
    public void pizzaWerktSamenMetDeJspPizza() {
        ModelAndView modelAndView = controller.pizza(1);
        assertEquals("pizza", modelAndView.getViewName());
    }
    @Test
    public void pizzaGeeftPizzaDoor() {
        ModelAndView modelAndView = controller.pizza(1);
        assertTrue(modelAndView.getModel().get("pizza") instanceof Pizza);
    }
    @Test
    public void onbestaandePizza() {
        ModelAndView modelAndView = pizzasController.pizza(-1);
        assertFalse(modelAndView.getModel().containsKey("pizza"));
    }
    @Test
    public void pizzasWerktSamenMetDeJspPizzas() {
        ModelAndView modelAndView = controller.pizzas();
        assertEquals("pizzas", modelAndView.getViewName());
    }
    @Test
    public void pizzasGeeftPizzasDoor() {
        ModelAndView modelAndView = controller.pizzas();
        assertTrue(modelAndView.getModel().get("pizzas") instanceof List);
    }
}
```

- (1) De method `getViewName` geeft je de naam van de JSP die je doorgaf in het `ModelAndView` object dat je teruggaf in de method `pizza`.
- (2) De method `getModel` geeft je de data die je doorgaf in het `ModelAndView` object dat je teruggaf in de method `pizza`. De method `getModel` geeft je die data als een `Map`. Je controleert of die `Map` een key met de naam `pizza` en met als waarde een `Pizza` object bevat.
- (3) Je controleer of de data in het `ModelAndView` object een key met de naam `pizzas` en met als waarde een `List` bevat. Door beperkingen in Java kan je hier niet `List<Pizza>` schrijven.

Je voert de unit test uit. Hij lukt.



Je commit de sources. Je publiceert op GitHub.



Unit test controller: zie takenbundel

17 REQUEST HEADERS

Je leerde in het begin van de cursus dat de browser in elke request ook request headers meegeeft.

Een voorbeeld is de request header user-agent. Deze bevat een String met daarin

- het browsermerk
- en het besturingssysteem waarop de browser draait

Een voorbeeld van zo'n request header: user-agent: Mozilla/5.0 (Windows NT 10.0; ...) Gecko/20100101 Firefox/56.0.

Je leert hier hoe je in een controller method de inhoud van een request header leest.

Je leest als voorbeeld de inhoud van de request header user-agent.

Je maakt een controller:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Controller
@RequestMapping("headers")
class HeaderController {
    private static final String VIEW = "headers";
    @GetMapping
    ModelAndView opWindows(@RequestHeader("user-agent") String userAgent) { ❶
        return new ModelAndView(VIEW,
            "opWindows", userAgent.toLowerCase().contains("windows")); ❷
    }
}
```

- (1) Spring zal de inhoud van de request header user-agent automatisch invullen in de method parameter die volgt op @RequestHeader: userAgent.
- (2) Je geeft aan de JSP data met de naam opWindows door.
Dit zal true bevatten als de request header user-agent het woord windows bevat.

Je maakt headers.jsp:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core' %>
<!doctype html>
<html lang='nl'>
    <head>
        <c:import url='/WEB-INF/JSP/head.jsp'>
            <c:param name='title' value="Headers"/>
        </c:import>
    </head>
    <body>
        <c:import url='/WEB-INF/JSP/menu.jsp' />
        Je browser wordt uitgevoerd op
        ${opWindows ? "Windows" : "een niet-Windows besturingssysteem"}.
    </body>
</html>
```

Je kan de website terug uitproberen.



Je commit de sources. Je publiceert op GitHub.



Voorkeurtaal: zie takenbundel

18 COOKIES

Een cookie is data die de browser bijhoudt ten dienste van een website.

- Een browser kan maximaal 20 cookies bijhouden per website.
- Één cookie is maximaal 4KB groot.
- Voorbeelden van cookies: de gebruikersnaam, voorkeur achtergrondkleur.
- Elke cookie heeft een naam en een String waarde.
- Er bestaan twee soorten cookies:
 - Tijdelijke cookies.
De browser onthoudt tijdelijke cookies in het interne geheugen, tot je de browser sluit.
 - Permanente cookies.
De browser onthoudt permanente cookies op de harde schijf.
Als je de browser sluit, blijven permanente cookies dus bestaan.
Een permanente cookie heeft een vervaltijdstip, die de website programmeur bepaalt.
De browser verwijdert de permanente cookie na dit vervaltijdstip.
- De browser stuurt bij elke request alle website cookies mee in de request header Cookie.
- Als een website een cookie wil toevoegen, wijzigen of verwijderen, geeft hij dit aan in de response header Set-Cookie.
- Bewaar in cookies geen confidentiële data (paswoorden, betaalkaartnummers, ...).
Je kan in de meeste browsers gemakkelijk de inhoud van cookies zien.
- De gebruiker kan in de browser cookies uitschakelen.
Als de website cookies gebruikt, zal hij bij die gebruiker niet goed functioneren.

18.1 Cookie

De class Cookie stelt een cookie voor:

Cookie
-name: String
-value: String
-maxAge: int
+Cookie(name: String, value: String)

Alle attributen (name, value, maxAge) hebben getters en setters.
Een Cookie met een maxAge -1 is een tijdelijke cookie.
Een cookie met een positieve waarde in maxAge is een permanente cookie.
maxAge geeft dan aan na hoeveel seconden de cookie verdwijnt.

Je wijzigt in IndexController de declaratie van de method index:

```
@GetMapping
ModelAndView index(@CookieValue(name = "laatstBezocht", required = false) ❶
String laatstBezocht, HttpServletResponse response) { ❷
...
}
```

- (1) Spring zal de inhoud van de cookie met de naam `laatstBezocht` automatisch invullen in de method parameter die volgt op `@CookieValue`: `laatstBezocht`. Je krijgt standaard een fout als deze cookie niet bestaat. Door `required` op `false` te plaatsen los je dit op: als de cookie niet bestaat vult Spring de method parameter `laatstBezocht` met `null`.
- (2) Om een cookie te schrijven heb je een object nodig van het type `HttpServletResponse`. Dit low-level object laat toe de response verfijnd in te stellen.

Je wijzigt het return statement:

```
Cookie cookie = new Cookie("laatstBezocht", LocalDateTime.now().toString()); ❶
cookie.setMaxAge(31_536_000); ❷
response.addCookie(cookie); ❸
ModelAndView modelAndView = new ModelAndView("index", "boodschap", boodschap)
    .addObject("zaakvoerder", new Persoon("Luigi", "Peperone", 7, true,
        new Adres("Grote markt", "3", 9700, "Oudenaarde")));
if (laatstBezocht != null) { ❹
    modelAndView.addObject("laatstBezocht", laatstBezocht);
}
return modelAndView;
```

- (1) Je maakt een cookie met de naam `laatstBezocht` en als inhoud de systeemtijd.
- (2) Je laat de cookie na 365 dagen vervallen.
- (3) Je voegt deze cookie toe aan de response die je naar de browser stuurt.
- (4) Als je de cookie `laatstBezocht` (ingesteld bij een vorig bezoek aan deze pagina) kon lezen, geef je hem door aan de JSP als data met de naam `laatstBezocht`.

Je voegt in `index.jsp` regels toe, na `<dl/>`:

```
<c:if test='${not empty laatstBezocht}'>
  <p>Je bezocht onze website laatst op ${laatstBezocht}.</p>
</c:if>
```

Je kan de website terug uitproberen.

Je zal verder in de cursus zien hoe je de datum en tijd op een gebruikersvriendelijker manier toont.



Je commit de sources. Je publiceert op GitHub.

Je kan een permanente cookie verwijderen door de `maxAge` eigenschap op `-1` te plaatsen.

Het volgend codefragment zou de cookie `laatstBezocht` verwijderen:

```
Cookie cookie = new Cookie("laatstBezocht", "");
cookie.setMaxAge(-1);
response.addCookie(cookie);
```



Bezocht: zie takenbundel

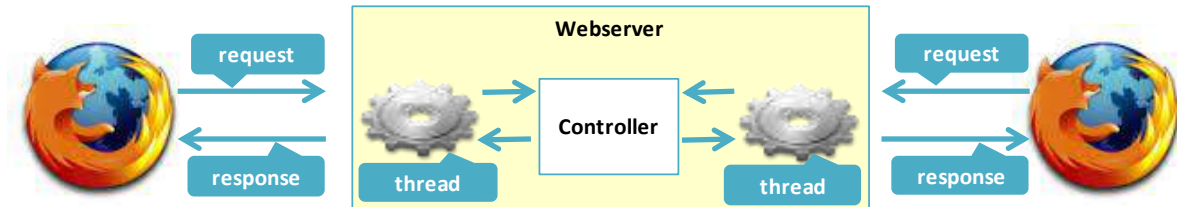
19 MULTITHREADING

Spring maakt van een controller class één bean (één object) bij het starten van de website.

Men spreekt ook over een singleton bean (naar het singleton design pattern).

Spring houdt deze bean bij in het interne geheugen zolang de website draait.

Deze ene bean verwerkt alle browser requests. Als meerdere browsers tegelijk requests naar de website sturen, verwerkt de website deze requests met één thread per request.



Als meerdere browsers tegelijk een request sturen naar de welkompagina, voeren meerdere threads dus gelijktijdig de code van IndexController uit.

Als je in IndexController een private variabele wijzigt gedurende het verwerken van een browser request, moet die variabele thread safe zijn.

Dit betekent dat het wijzigen van de variabele door meerdere gelijktijdige thread correct verloopt.

- Primitieve variabelen (int, long, double, ...) zijn niet thread safe.
Als je dit soort private variabelen wijzigt in een controller, zijn volgende classes ter beschikking als alternatief voor de primitieve types:
 - AtomicBoolean een thread safe boolean
 - AtomicInteger een thread safe int
 - AtomicLong een thread safe long
 - Sommige classes uit de Java libraries zijn thread safe. Een voorbeeld daarvan is StringBuffer. Veel classes uit de Java libraries zijn niet thread safe.
JDBC objecten (Connection, Statement, ResultSet, ...) zijn niet thread safe.
Collection objecten (ArrayList, HashSet, HashMap, ...) zijn niet thread safe.
De package java.util.concurrent bevat alternatieve classes die wél thread safe (maar wat minder performant) zijn:
 - CopyOnWriteArrayList een thread safe List
 - CopyOnWriteArraySet een thread safe Set
 - ConcurrentHashMap een thread safe Map

19.1 AtomicInteger voorbeeld

Je maakt in IndexController een teller.

Je houdt daarin bij hoeveel requests de controller kreeg over alle browsers heen.

```
private final AtomicInteger aantalKeerBekeken = new AtomicInteger(); ❶
```

- (1) De constructor van AtomicInteger initialiseert de getalwaarde in die AtomicInteger op 0.

Je voegt in de method index een opdracht toe voor de return opdracht:

```
modelAndView.addObject("aantalKeerBekeken",  
    aantalKeerBekeken.incrementAndGet()); ❶
```

- (1) De method incrementAndGet verhoogt de teller in de AtomicInteger op een thread-safe manier. De method geeft deze verhoogde deze teller terug als returnwaarde.

Je tikt in index.jsp voor </body>

```
<p>Deze pagina werd ${aantalKeerBekeken} keer bekeken.</p>
```

Je kan de website uitproberen.

Je simuleert meerdere gebruikers door de website in verschillende browsers te openen.



Je commit de sources. Je publiceert op GitHub.

20 STATELESS

Het HTTP protocol is een stateless protocol.

Dit betekent dat de webserver, na het verwerken van een request, in het interne geheugen alle data vergeet die hij opbouwde tijdens het verwerken van die request. Zo kunnen veel gebruikers tegelijk een website bezoeken en heeft de website toch geen geheugentekort.

Het houdt wel in dat je bij iedere request *alle* data moet opbouwen om de pagina te tonen.

Een beginnende ontwikkelaar kan dit wel eens over het hoofd zien. Je ziet dit met een voorbeeld: je maakt een pagina Prijzen. Je toont in die pagina de unieke pizza prijzen als hyperlinks.

Als de gebruiker zo'n hyperlink kiest, toont je onder die hyperlinks de pizza's met de gekozen prijs.

Als de gebruiker in het menu Prijzen kiest, toon je de pagina met unieke pizzaprijzen.

Je toont nog geen pizza's. Je doet dit pas als de gebruiker een hyperlink met een prijs aanklikt.

Je maakt een method in PizzaController:

```
private static final String PRIJZEN_VIEW = "prijzen";
@GetMapping("prijzen")
ModelAndView prijzen() {
    return new ModelAndView(PRIJZEN_VIEW,
        "prijzen", pizzas.values().stream().map(pizza ->
            pizza.getPrijs()).distinct().collect(Collectors.toSet()));
}
```

- (1) De method die hoort bij deze @GetMapping verwerkt GET requests naar de URL vermeld bij @RequestMapping("pizzas"), gevolgd door /, gevolgd door de URL hier vermeld bij @GetMapping("prijzen"). De method verwerkt dus GET requests naar de URL pizzas/prijzen.

Je maakt prijzen.jsp:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
<%@taglib prefix='form' uri='http://www.springframework.org/tags/form'%>
<!doctype html>
<html lang='nl'>
<head>
    <c:import url='/WEB-INF/JSP/head.jsp'>
        <c:param name='title' value='Prijzen'/>
    </c:import>
</head>
<body>
<c:import url='/WEB-INF/JSP/menu.jsp' />
<h1>Prijzen</h1>
<ul>
<c:forEach items='${prijzen}' var='prijs'>
    <c:url value='/pizzas' var='url'>
        <c:param name='prijs' value='${prijs}' />
    </c:url>
    <li><a href='${url}'>${prijs}</a></li>
</c:forEach>
</ul>
<c:if test='${not empty pizzas}'>
    <h2>${prijs}</h2>
    <ul>
    <c:forEach items='${pizzas}' var='pizza'>
        <spring:url var='url' value='/pizzas/{id}'>
            <spring:param name='id' value='${pizza.id}' />
        </spring:url>
        <li><a href='${url}'><c:out value='${pizza.naam}' /></a></li>
    </c:forEach>
</c:if>
```

```

    </ul>
  </c:if>
</body>
</html>

```

Je kan de website uitproberen (maar je kan nog niet op de hyperlinks met prijzen klikken).

Je maakt in `PizzaController` een method . Die verwerkt de request als de gebruiker een hyperlink met een prijs aanklikt. Als beginnend ontwikkelaar zou je kunnen denken dat je enkel de pizza's moet zoeken met de geselecteerde prijs. Je zou kunnen denken dat je de unieke prijzen niet moet zoeken, gezien deze "al op de pagina staan":

```

@GetMapping(params="prijs")
ModelAndView pizzasVanPrijs(BigDecimal prijs) {
    return new ModelAndView(PRIJZEN_VIEW, "pizzas",
        pizzas.values().stream().filter(pizza ->
            pizza.getPrijs().equals(prijs)).collect(Collectors.toList()))
        .addObject("prijs", prijs);
}

```

Je bezoekt de website. Je klikt op een hyperlink met een prijs. Je ziet wel de pizza's met die prijs, maar niet meer de hyperlinks met de prijzen. De browser hertekent bij elke request (hier een klik op een hyperlink) de pagina vanaf nul met de HTML die de webserver hem toestuurt.

Bij het verwerken van deze request lees je de unieke prijzen niet. Je geeft ze dus ook niet door aan de JSP. De JSP maakt daarom geen HTML met die unieke prijzen.

Je lost dit op. Je leest in de method `pizzasVanPrijs` de unieke prijzen en geeft ze door aan de JSP:

```

@GetMapping(params = "prijs")
ModelAndView pizzasVanPrijs(BigDecimal prijs) {
    return new ModelAndView(PRIJZEN_VIEW, "pizzas",
        pizzas.values().stream().filter(pizza ->
            pizza.getPrijs().equals(prijs)).collect(Collectors.toList()))
        .addObject("prijs", prijs)
        .addObject("prijzen", pizzas.values().stream().map(pizza ->
            pizza.getPrijs()).distinct().collect(Collectors.toSet()));
}

```

Je kan deze pagina opnieuw uitproberen.






Je commit de sources. Je publiceert op GitHub.

Voordelen van stateless:

- ⊕ De webserver houdt tussen de requests geen data bij in het interne geheugen. Veel gelijktijdige gebruikers kunnen dus de website bezoeken zonder dat de webserver geheugen te kort komt.
- ⊕ In een webfarm mogen *verschillende* webserver de requests van een gebruiker verwerken. Voorbeeld: de gebruiker kiest Prijzen in het menu. Webserver 1 verwerkt deze request. De gebruiker kiest daarna een prijs. Webserver 2 verwerkt deze request. De gebruiker ziet een correct pagina: de code op Webserver 2 leest de unieke prijzen en de pizza's met de gekozen prijs uit de database. Hij probeert die data niet uit het interne geheugen te halen in de hoop dat hij ze daar bij een vorige request onthouden heeft. Dit zou niet lukken: de vorige request werd verwerkt door een andere webserver: Webserver 1.
- ⊕ De gebruiker krijgt altijd een correct beeld van de prijzen. Voorbeeld: gebruiker 1 ziet de prijzen. Gebruiker 2 voegt in de database een pizza met een nieuwe prijs toe. Gebruiker 1 "refresh" daarna de pagina. Je code leest bij deze request ook de prijs van die nieuwe pizza uit de database. Gebruiker 1 ziet dus ook die nieuwe prijs.

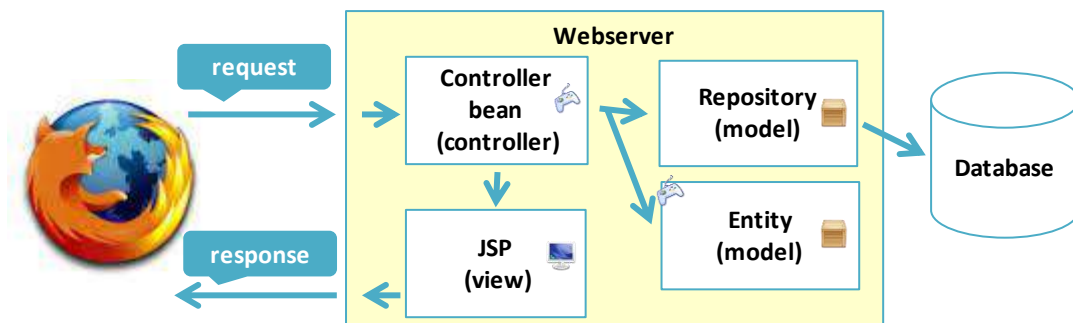
21 MODEL-VIEW-CONTROLLER

Bij het MVC (Model-View-Controller) design pattern bevat je website drie soorten onderdelen:

	Onderdeel	Taak van dit onderdeel
	Model	De werkelijkheid voorstellen en databasebewerkingen doen. <ul style="list-style-type: none"> • Entitiy classes stellen de werkelijkheid voor. • Repository classes doen databasebewerkingen.
	View	Data mooi opgemaakt tonen aan de gebruiker. Bij ons is dit onderdeel een JSP.
	Controller	Requests binnenkrijgen. Bij ons is dit onderdeel een controller bean.

Model, view en controller werken samen bij het verwerken van een request:

1. De controller  (PizzaController) krijgt een browser request om pizza's te zien.
2. Hij spreekt classes aan uit het model : hij roept de class PizzaRepository op. Die leest pizza's uit de database en geeft ze aan PizzaController als een `List<Pizza>`. Pizza behoort ook tot het model .
3. De controller  geeft de pizza's door aan de view  (JSP). Die maakt HTML en stuurt die als response naar de browser.



22 ARCHITECTUUR VAN EEN ENTERPRISE APPLICATION

In een grote (“enterprise”) applicatie spreek je, bij het verwerken van browser requests, ook de database aan of haal je data op uit andere websites.

Als je de code die dit doet schrijft in de controller, worden de methods in die controller lang en onoverzichtelijk, omdat één method meerdere uiteenlopende taken uitvoert.

Je zou in één method bijvoorbeeld volgende code vinden:

- producten uit de database lezen.
- de koers van de dollar opzoeken op de website van de european central bank.
- de prijs van ieder gelezen product omzetten naar dollar.

Een ander probleem is dat je in verschillende controllers dezelfde code zou terug vinden.

Dit kan bijvoorbeeld de code zijn om alle producten uit de database te lezen.

Je lost deze problemen op:

je schrijft de verschillende verantwoordelijkheden in verschillende classes.

Je project bevat nu al twee soorten classes, die elk een eigen verantwoordelijkheid hebben:

- controllers verwerken browser requests.
- entities stellen dingen of personen uit de werkelijkheid voor.

Men spreekt ook over softwarelagen (layers). Je project bevat nu 2 lagen:

- presentation
Deze laag is verantwoordelijk voor
 - het tonen van gegevens aan de gebruiker
 - en het vragen van gegevens aan de gebruikerDe controllers en JSP's behoren tot deze laag.
- model
Deze laag bevat de entities.

Daarnaast bestaan ook volgende lagen: repositories, services, restclients en restservices.

22.1 Repositories

Deze laag bevat code waarmee je (met JDBC of een soortgelijke library) de database aanspreekt. Deze laag bevat geen transactie code. Deze bevindt zich in een andere laag: services (zie verder).

Bij iedere entity class hoort één class in de repositories laag. Als je bij Pizza Luigi de entity classes Klant en Product hebt, heb je de bijbehorende repository classes KlantenRepository en ProductenRepository. KlantenRepository doet databasebewerkingen op de table klanten. ProductenRepository doet databasebewerkingen op de table producten.

In een Repository class vind je CRUD methods en find methods.

CRUD is een afkorting van Create, Read, Update en Delete.

- De create method voegt één record toe:
`insert into producten(naam, prijs) values (?, ?)`
- De read method leest één record waarvan de primary key gekend is:
`select naam, prijs from producten where id=?`
- De update method wijzigt één record:
`update producten set naam=?, prijs=? where id=?`
- De delete method verwijdert één record.:
`delete from producten where id=?`
- De find methods lezen ook records. Deze keer is niet de primary key van het te zoeken record gekend, maar zoek je records op andere criteria.
Voorbeeld: Je zoekt de producten met een prijs tussen twee grenzen:
`select id, naam, prijs from producten where prijs between ? and ?`

22.2 RestClients

De classes in deze layer communiceren met andere applicaties over het internet of intranet.

De Fixer website (<https://fixer.io>) biedt de koers van de dollar ten opzichte van de euro aan.

Je RestClient class FixerKoersClient leest deze koers en geeft ze als een BigDecimal aan de andere applicatie lagen.

FixerKoersClient
+getDollarKoers(): BigDecimal

22.3 RestServices

De classes in deze layer bieden data aan andere applicaties aan, over het internet of over het intranet. Je biedt bijvoorbeeld de producten van Pizza Luigi in XML formaat aan aan andere applicaties op het internet.

22.4 Services

Een method van een service class stelt één use case (programmaonderdeel) voor, behalve de user-interface van die use case. De user interface van die use case zit in de presentation layer.

Een service class method roept methods op uit de repositories layer, om de databasebewerkingen uit te voeren die bij de use case horen. Een service class method kan ook een method oproepen uit de restClient layer, om data op te halen van een andere applicatie.

Er is één service class per entity class. Als je in een bank de entity classes Klant en Rekening hebt, heb je bijbehorende service classes KlantenService en RekeningenService.

- methods uit KlantenService stellen use cases voor die vooral met klanten werken.
- methods uit RekeningenService stellen use cases voor die vooral met rekeningen werken.

Een voorbeeld van een method uit RekeningenService is de method
void overschrijven(String vanRekeningNr, String naarRekeningNr, BigDecimal bedrag)

Deze method zoekt in de database de rekening met het nummer vanRekeningNr. Hij vermindert in die rekening het saldo met het bedrag. Hij zoekt de rekening met het nummer naarRekeningNr. Hij vermeerderd het saldo van die rekening met het bedrag. Het is belangrijk dat de wijzigingen behoren tot één transactie. De database wijzigt dan ofwel *beide* rekeningen of doet (bijvoorbeeld bij een stroompanne midden in de wijzigingen) ale wijzigingen ongedaan.

De binnenkant van de method zal volgende stappen doen:

1. Een transactie starten.
2. De method read van RekeningRepository oproepen om de Rekening entity te zoeken die hoort bij vanRekeningNr.
3. De method read van RekeningRepository oproepen om de Rekening entity te zoeken die hoort bij naarRekeningNr.
4. De method overschrijven oproepen van de 1° Rekening entity en daarbij de 2° Rekening entity en het bedrag als parameter meegeven. Deze method vermindert het saldo in de 1° entity (als er voldoende saldo is) en vermeerderd het saldo in de 2° entity.

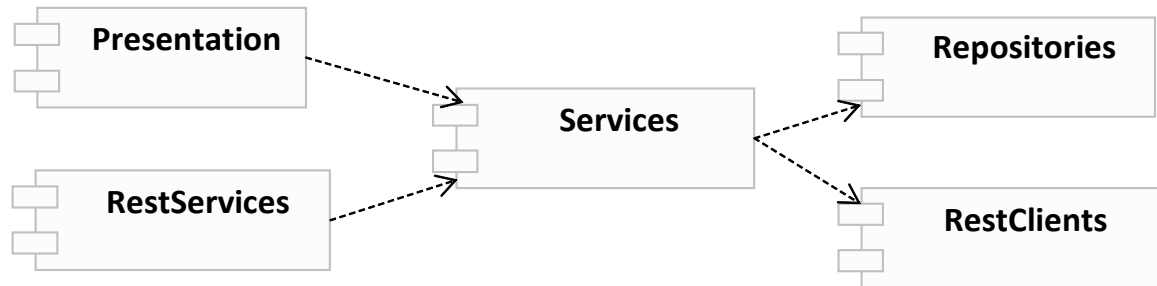
Beide wijzigingen gebeuren in het interne geheugen.

```
public void overschrijven(BigDecimal bedrag, Rekening naarRekening) {
    if (this.saldo.compareTo(bedrag) < 0) {
        throw new IllegalArgumentException("saldo ontoereikend");
    }
    this.saldo = this.saldo.subtract(bedrag);
    naarRekening.saldo = naarRekening.saldo.add(bedrag)
}
```

5. De method update van RekeningRepository oproepen om het record te wijzigen dat hoort bij de eerste entity.
6. De method update van RekeningRepository oproepen om het record te wijzigen dat hoort bij de tweede entity.
7. Een commit op de transactie.

22.4.1 Samenwerking tussen de layers

De layers werken samen om een use case uit te voeren. De presentation layer roept de services layer op die op zijn beurt de repositories layer en/of de restclients layer oproept:

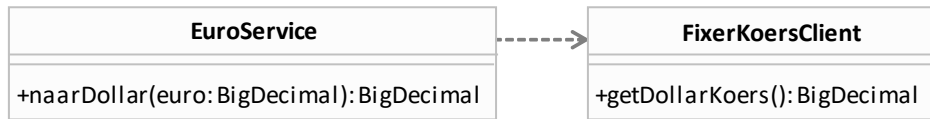


De pijlen drukken een dependency (afhankelijkheid) uit.

De layer waaruit de pijl vertrekt gebruikt (is afhankelijk van) de layer waarin de pijl aankomt.

23 DEPENDENCY INJECTION

Je leerde in het vorige hoofdstuk dat een class uit een layer classes uit andere layers kan oproepen. De servie class EuroService roept bijvoorbeeld de restclient class FixerKoersClient op:



De method `naarDollar` converteert een bedrag in € naar \$. Je hebt daarbij de koers van de \$ nodig. Je roept daarom de method `getDollarKoers` van `FixerKoersClient` op. Die zoekt de koers op de Fixer website en geeft die terug.

`EuroService` heeft dus een dependency op `FixerKoersClient`.

Je zou deze dependency kunnen hard coderen:

```

public class EuroService {
    private final FixerKoersClient koersClient = new FixerKoersClient();
    // vorige regel= hard gecodeerde dependency
    public BigDecimal naarDollar(BigDecimal euro) {
        return euro.multiply(koersClient.getDollarKoers())
            .setScale(2, RoundingMode.HALF_UP);
    }
}
  
```

Een dependency hard coderen is een slecht idee. Het laat niet toe

- bij unit testen de dependency te vervangen door een dummy object.
- meerdere implementaties van de dependency te hebben (er zijn nog andere websites dan Fixer die de koers van de dollar ten opzichte van de euro aanbieden).

23.1 Dummy objecten gebruiken bij unit testen

Je schrijft voor elke class een bijbehorende unit-test.

Je wil in de test enkel de werking van de class testen, niet de werking van zijn dependencies.

- Je test in `EuroServiceTest` `EuroService`, niet zijn dependency (`FixerKoersClient`).
- Als `EuroService` een hard gecodeerde dependency heeft op `FixerKoersClient`, voer je bij een `EuroService` test ook `FixerKoersClient` code uit en test je ook die class.

Als oplossing vervang je in de test de echte dependency (`FixerKoersClient`) door een dummy object (`dummyKoersClient`). De echte dependency en de dummy zijn slechts inwisselbaar als ze beiden eenzelfde interface (`KoersClient`) implementeren.

23.1.1 Interface

Je maakt een package `be.vdab.pizzaluigi.restclients`.

Je maakt daarin de interface `KoersClient`:

```

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

23.1.2 Service

Je maakt een package `be.vdab.pizzaluigi.services` en daarin de class `EuroService`:

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

- (1) Als je `@Service` tikt voor een class, maakt Spring bij het starten van de website een bean (object) van deze class. Spring houdt deze bean bij zolang de website in uitvoering is. Dit is vergelijkbaar met `@Controller` voor een class.
- (2) De variabele kan verwijzen naar elk object dat de interface `KoersClient` implementeert. Dit kan een `FixerKoersClient` object zijn als je de code uitvoert op de website, of een dummy object aangemaakt met Mockito als je de code uitvoert in een test. Je maakt de variabele `final`. Je garandeert zo dat je deze variabele in een andere method dan de constructor niet kan wijzigen en per ongeluk op `null` zou plaatsen.
- (3) De constructor krijg een object geïnjecteerd dat de interface `KoersClient` implementeert. In een unit testen roep je deze constructor op en je injecteer een dummy die je maakt met Mockito. Als je de code uitvoert in de website roept Spring deze constructor op. Spring ziet dat hij een object moet meegeven dat de interface `KoersClient` implementeert. Spring zoekt tussen alle beans een bean die deze interface implementeert. Als Spring zo'n bean vindt, geeft hij deze mee aan de constructor. Dit heet dependency injection. Als Spring zo'n bean niet vindt, krijg je een exception. We zorgen er straks voor dat Spring een bean ter beschikking heeft gebaseerd op de class `FixerKoersClient`.

23.1.3 Test van de service

Je maakt in `src/test/java` een package `be.vdab.pizzaluigi.services` en daarin de test class:

```
package be.vdab.pizzaluigi.services;
import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;
// enkele andere imports
@RunWith(MockitoJUnitRunner.class)
public class EuroServiceTest {
    @Mock
    private KoersClient dummyKoersClient;
    private EuroService euroService;
    @Before
    public void before() {
        when(dummyKoersClient.getDollarKoers()).thenReturn(BigDecimal.valueOf(1.5));
        euroService = new EuroService(dummyKoersClient);
    }
    @Test
    public void naarDollar() {
        assertEquals(0, BigDecimal.valueOf(3).compareTo(
            euroService.naarDollar(BigDecimal.valueOf(2))));
        verify(dummyKoersClient).getDollarKoers();
    }
}
```


Je kan de unit test uitvoeren.

Andere voordelen van het gebruik van een dummy object bij het testen:

- De test loopt snel. Het dummy object maakt de koers in het geheugen. Dit gaat veel sneller dan deze koers te lezen van de Fixer website.
- Je kan EuroService schrijven en testen terwijl FixerKoersClient nog niet bestaat. Dit gebeurt als verschillende personen de classes schrijven.
- Je kan testen hoe EuroService reageert als FixerKoersClient een fout werpt omdat de Fixer website niet werkt. Als je FixerKoersClient als dependency zou gebruiken, moet je, om de fout na te bootsen, Fixer vragen hun website stil te leggen. Dit is onmogelijk. Als je dummyKoersClient als dependency gebruikt, simuleer je de fout door vanuit dummyKoersClient exceptions te werpen.

23.2 Eerste implementatie van de dependency

Één dependency kan meerdere echte implementaties hebben:

je kan vanuit meerdere websites de koersen van munten lezen.

- De ene website geeft de koersen gratis, bij de andere website moet je betalen.
- De ene website geeft de koersen in real-time, de andere website met vertraging.
- De ene website heeft een limiet hoeveel koersen je per maand kan opvragen, de andere website heeft geen limiet.
- De ene website garandeert 99,99% van de dag beschikbaarheid, de andere website minder.

Je maakt per website één restclient class:

- FixerKoersClient leest de koersen op de Fixer website.
- ECBKoersClient leest de koersen op de ECB website.

De class EuroService heeft een dependency op één van deze classes.

Je wil dat je een andere dependency kan gebruiken zonder veel code te wijzigen.

Dit kan terug als al deze dependencies de interface KoersClient implementeren.

Je maakt eerst een exception class, die je straks gebruikt:

```
package be.vdab.pizzaluigi.exceptions;
public class KoersClientException extends RuntimeException {
    private static final long serialVersionUID = 1L;
    public KoersClientException(String message) {
        super(message);
    }
}
```

Je begint met een eerste implementatie: de class FixerKoersClient.

Je vraagt op <https://fixer.io> een free api key. Dit is een waarde die je bij elke request meegeeft.

Je kan met een “free” key 1000 requests per maand naar de fixer website sturen.

Je zal vanuit je code een request doen naar (je kan dit nu ook eens in je browser proberen)

http://data.fixer.io/api/latest?access_key=TikHierJeFreeKey&symbols=USD

Je krijgt volgende response:

```
{ "success": true, "timestamp": 1526453227, "base": "EUR",
  "date": "2018-05-16", "rates": { "USD": 1.183715 } }
```

Je maakt de class FixerKoersClient:

```
package be.vdab.pizzaluigi.restclients;
// enkele imports
@Component
class FixerKoersClient implements KoersClient {
    private static final Logger LOGGER =
        LoggerFactory.getLogger(FixerKoersClient.class);
    // importeer Logger en LoggerFactory uit org.slf4j
    private final URL url;
```

①
②
③

```

FixerKoersClient() {
    try {
        url = new URL(
            "http://data.fixer.io/api/latest?access_key=TikHierJeFreeKey&symbols=USD");
    } catch (MalformedURLException ex) {
        String fout = "Fixer URL is verkeerd.";
        LOGGER.error(fout, ex);
        throw new KoersClientException(fout);
    }
}
@Override
public BigDecimal getDollarKoers() {
    try (Scanner scanner = new Scanner(url.openStream())) {
        String lijn = scanner.nextLine();
        int beginPositieKoers = lijn.indexOf("USD") + 5;
        int accoladePositie = lijn.indexOf('}', beginPositieKoers);
        return new BigDecimal(lijn.substring(beginPositieKoers, accoladePositie));
    } catch (IOException | NumberFormatException ex) {
        String fout = "kan koers niet lezen via Fixer";
        LOGGER.error(fout, ex);
        throw new KoersClientException(fout);
    }
}
}

```

- (1) Als je `@Component` voor een class tikt, maakt Spring bij het starten van de website een bean (object) van deze class. Spring houdt deze bean bij zolang de website draait.
Het is deze bean die Spring via de constructor zal injecteren in de `EuroService` bean.
- (2) Als een fout optreedt, wil je info over de fout schrijven naar de console, een bestand, ...
Hiervoor bestaat de class `Logger`. Standaard schrijft de `Logger` de info naar de console.
Als je `logging.file=/logging/log.txt` toevoegt aan `application.properties`, schrijft Spring de foutinformatie ook naar het bestand `log.txt` in de map `logging` in de root van je hard disk.
- (3) De static method `getLogger` geeft je een `Logger` object.
Je geeft aan `getLogger` de huidige class mee.
- (4) Je schrijft foutinformatie weg. De 2^o parameter is de opgetreden exception.
De `Logger` schrijft dan ook weg op welk lijnnummer van welke source de exception optrad.

23.3 Dependency injection in de controller

Je wil de class `EuroService` oproepen vanuit een controller.

Ook in die controller zou het verkeerd zijn deze dependency hard te coderen:

```
private final EuroService euroService = new EuroService();
```

Het laat niet toe dat je bij een unit test van de controller een dummy object gebruikt in plaats van de echte `EuroService`.

Je zal daarom ook hier dependency injection toepassen.

Je hernoemt met de hulp van Eclipse `EuroService` naar `DefaultEuroService`:

1. Je selecteert `EuroService` in de `Project Explorer`.
2. Je drukt F2.
3. Je tikt `DefaultEuroService`.
4. Je drukt Enter.

Je laat Eclipse een interface `EuroService` maken die `DefaultEuroService` implementeert:

1. Je selecteert `DefaultEuroService` in de `Project Explorer`.
2. Je kiest in het menu `Refactor` de opdracht `Extract Interface`.
3. Je tikt `EuroService` bij `Interface name`.
4. Je kiest `Select All`.
5. Je kiest OK.

Je tikt public voor deze interface.

Je wijzigt PizzaController:

Je voegt een variabele toe:

```
private final EuroService euroService;
```

Je voegt een parameter toe aan de constructor:

```
PizzaController(EuroService euroService) {
```

❶

- (1) In een unit test van de controller roep je de constructor op en geef je een dummyEuroService object mee. Bij het uitvoeren van de website roept Spring deze constructor op en geeft de bean mee die deze interface implementeert: de bean gebaseerd op de class DefaultEuroService.

Je voegt een opdracht toe aan de constructor:

```
this.euroService = euroService;
```

Je wijzigt de code in de method pizza:

```
ModelAndView modelAndView = new ModelAndView(PIZZA_VIEW);
```

❷

```
Pizza pizza = pizzas.get(id);
```

```
modelAndView.addObject(pizza);
```

```
modelAndView.addObject("inDollar", euroService.naarDollar(pizza.getPrijs()));
```

```
return modelAndView;
```

- (1) Je gebruikt hier voor het eerst de ModelAndView constructor met één parameter: de naam van de JSP. Verder in de code geef je met de addObject method ook data door aan die JSP.

Je wijzigt PizzaControllerTest:

Je voegt een regel toe voor de class:

```
@RunWith(MockitoJUnitRunner.class)
```

Je voegt een private variabele toe:

```
@Mock
```

```
private EuroService dummyEuroService;
```

Je wijzigt de method before:

```
@Before
```

```
public void before() {
```

```
    pizzaController = new PizzaController(dummyEuroService);
```

```
}
```

Je voegt in pizza.jsp code toe, voor de laatste <dt>:

```
<dt>In dollar</dt><dd>${inDollar}</dd>
```

Je kan de website uitproberen.

Je schrijft ook een integration test (in src/test/java) voor FixerKoersClient.

Je test bij een integration test een class in samenwerking met zijn omgeving (hier de Fixer website).

```
package be.vdab.pizzaluigi.restclients;
```

```
// enkele imports
```

```
public class FixerKoersClientTest {
```

```
    private FixerKoersClient client;
```

```
@Before
```

```
    public void before() {
```

```
        client = new FixerKoersClient();
```

```
    }
```

```
@Test
```

```
    public void deKoersMoetPositiefZijn() {
```

```
        assertTrue(client.getDollarKoers().compareTo(BigDecimal.ZERO) > 0);
```

```
    }
```

```
}
```



Je commit de sources. Je publiceert op GitHub.

23.4 Als een dependency ontbreekt

Als je vergeet `@Component` bij de class `FixerKoersClient` te tikken, maakt Spring geen bean van deze class. Spring heeft dan ook geen bean ter beschikking om te injecteren in de constructor van `DefaultEuroService`. Spring zal dan een exception werpen bij de start van de website.

Gelijkaardig: als je vergeet `@Service` bij de class `DefaultEuroService` te tikken, maakt Spring geen bean van deze class. Spring heeft dan ook geen bean ter beschikking om te injecteren in de constructor van `PizzaController`. Spring zal een exception werpen bij de start van de website.

Je probeert dit uit. Je plaatst `@Service` in commentaar bij de class `DefaultEuroService`.

Je start de website. Je krijgt een exception met als omschrijving:

```
Parameter 0 of constructor in be.vdab.pizzaluigi.web.PizzaController required  
a bean of type 'be.vdab.services.EuroService' that could not be found.
```

Action:

Consider defining a bean of type 'be.vdab.pizzaluigi.services.EuroService' in your configuration.

Je haalt `@Service` terug uit commentaar in de class `DefaultEuroService`.



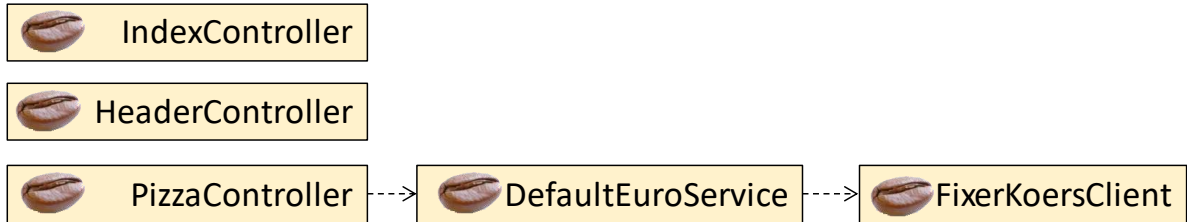
Sausen.csv: zie takenbundel

24 DE IOC CONTAINER

IOC is een afkorting van Inversion of Control. Het is een synoniem voor dependency injection.

De IOC container is de verzameling van alle Spring beans in je project.

Je ziet hier onder een overzicht van de beans in de IOC container, met hun dependencies:



25 MEERDERE IMPLEMENTATIES VAN EEN DEPENDENCY

Je hebt soms van één dependency meerdere implementaties. Je leert hier hoe je hiermee omgaat.

Je maakt een tweede class die ook de interface KoersClient implementeert.

Deze class leest de koers van de dollar op de website van de ECB (European Central Bank).

Je voegt eerst een regel toe voor het eerste return statement in FixerKoersClient:

```
LOGGER.info("koers gelezen via Fixer");
```

❶

Je gebruikt de info method als je in het venster Console van Eclipse informatief wil zien dat je deze method uitvoert.

Je maakt in be.vdab.pizzaluigi.restclients de class ECBKoersClient.

Die leest regels op de URL <https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml>.

Één van die regels bevat de koers van de dollar: `<Cube currency="USD" rate="1.3594"/>`

```
package be.vdab.pizzaluigi.restclients;
```

```
// enkele imports
```

```
@Component
```

```
class ECBKoersClient implements KoersClient {
```

```
    private static final Logger LOGGER =
```

```
        LoggerFactory.getLogger(ECBKoersClient.class);
```

```
    private final URL url;
```

```
    public ECBKoersClient() {
```

```
        try {
```

```
            this.url = new URL(
```

```
                "https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml");
```

```
        } catch (MalformedURLException ex) {
```

```
            String fout = "ECB URL is verkeerd.";
```

```
            LOGGER.error(fout, ex);
```

```
            throw new KoersClientException(fout);
```

```
        }
```

```
    }
```

```
@Override
```

```
public BigDecimal getDollarKoers() {
```

```
    XMLInputFactory factory = XMLInputFactory.newInstance();
```

```
    try (InputStream stream = url.openStream()) {
```

```
        XMLStreamReader reader = factory.createXMLStreamReader(stream);
```

```
        try {
```

```
            while (reader.hasNext()) {
```

```
                if (reader.next() == XMLStreamConstants.START_ELEMENT
```

```
                    && "USD".equals(reader.getAttributeValue(null, "currency"))) {
```

```
                    LOGGER.info("koers gelezen via ECB");
```

```
                    return new BigDecimal(reader.getAttributeValue(null, "rate"));
```

```
                }
```

```
            }
```

```
            String fout = "XML van ECB bevat geen USD";
```

```
            LOGGER.error(fout);
```

```
            throw new KoersClientException(fout);
```

```
        } finally {
```

```
            reader.close();
```

```
        }
```

```
    } catch (IOException | NumberFormatException | XMLStreamException ex) {
```

```
        String fout = "kan koers niet lezen via ECB";
```

```
        LOGGER.error(fout, ex);
```

```
        throw new KoersClientException(fout);
```

```
    }
```

```
}
```

```
}
```

❶

❷

❸

❹

❺

❻

❼

❽

- (1) Je gebruikt bij ❷ een XMLStreamReader object om XML te lezen.
Je hebt een XMLInputFactory nodig om dit object te maken.
Dit is een toepassing van het factory design pattern.
Je maakt het XMLInputFactory object op zijn beurt met de static method newInstance.
- (2) Je maakt een XMLStreamReader object. Je leest met zo'n object sequentieel (één per één) de onderdelen van een XML bron: begintags, eindtags, commentaar, ...
- (3) De method hasNext geeft true terug zolang je nog onderdelen uit het XML bestand kan lezen.
Deze method geeft false terug bij het einde van het bestand.
- (4) Je leest met de method next een volgend onderdeel uit de XML bron.
Deze method geeft je een int terug met het soort onderdeel.
Als deze int gelijk is aan de constante START_ELEMENT heb je een begintag gelezen.
- (5) Je controleert of de tag een attribuut heeft met de naam USD.
Je zou de parameterwaarde null vervangen door een namespace als de XML bron met een namespace geassocieerd is. Dit is hier niet het geval.
- (6) Je leest de koers van de dollar in het attribuut rate.
- (7) XmlStreamReader implementeert AutoCloseable niet. Je moet hem daarom zelf sluiten.
- (8) Als je hier komt
 - a. is er een exception opgetreden
 - b. of heb je de XML volledig gelezen en de koers van de dollar niet tegengekomen.

Je maakt in src/test/java een integration test voor deze class:

```
package be.vdab.pizzaluigi.restclients;
// enkele imports
public class ECBKoersClientTest {
    private ECBKoersClient client;
    @Before
    public void before() {
        client = new ECBKoersClient();
    }
    @Test
    public void deKoersMoetPositiefZijn() {
        assertTrue(client.getDollarKoers().compareTo(BigDecimal.ZERO) > 0);
    }
}
```

Je voert deze test uit.

Je start de website. Je krijgt daarbij een exception met volgende omschrijving:

Parameter 0 of constructor in be.vdab.pizzaluigi.services.DefaultEuroService required a single bean, but 2 were found:

- ECBKoersClient
- fixerKoersClient

Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed

In het nederlands: Spring kan zelf niet kiezen of hij de FixerKoersClient bean of de ECBKoersClient bean injecteert in de constructor van DefaultEuroService.

Je kan dit probleem oplossen met @Primary of met @Qualifier.

25.1 @Primary

Als *meerdere* implementaties een dependency implementeren, en je tikt bij één van die implementaties @Primary, doet Spring dependency injection met die implementatie.

Je probeert dit uit: Je tikt @Primary voor de class ECBKoersClient. Je start de website.

Je bekijkt de detail van een pizza. Je ziet daarna in het Console venster van Eclipse:

koers gelezen via ECB

25.2 @Qualifier

@Qualifier is een alternatieve oplossing dan @Primary.

Je verwijdert daarom @Primary in de class ECBKoersClient.

@Qualifier lost het probleem in 2 stappen op:

25.2.1 @Qualifier bij de bean classes

Je tikt @Qualifier voor elke bean class, die dezelfde interface implementeert.

Je tikt bij @Qualifier een unieke string met een betekenisvolle omschrijving van de class.

- Je tikt @Qualifier("Fixer") voor de class FixerKoersClient.
- Je tikt @Qualifier("ECB") voor de class ECBKoersClient.

25.2.2 @Qualifier bij constructor injection

Je tikt @Qualifier ook bij een constructor waarin je deze beans injecteert. Je tikt bij @Qualifier een string. Spring injecteert de bean van de class met een @Qualifier met dezelfde string.

Je doet dit in de DefaultEuroService constructor:

```
DefaultEuroService(@Qualifier("Fixer") KoersClient koersClient) { ❶
    this.koersClient = koersClient;
}
```

- (1) Je tikt @Qualifier vóór de te injecteren parameter. Je geeft de string Fixer mee.
Spring injecteert de bean van de class die de interface KoersClient implementeert én voorzien is van @Qualifier("Fixer"). Dit is de bean van de class FixerKoersClient.

Je kan de applicatie uitproberen.

25.3 Meerdere dependencies injecteren

Je injecteert tot nu in DefaultEuroService één dependency die KoersClient implementeert, terwijl er meerdere dependencies bestaan. Als je een andere dependency wil injecteren (bijvoorbeeld omdat de Fixer website niet functioneert), moet je de applicatie wijzigen.

Een andere strategie is alle mogelijke dependencies te injecteren in DefaultEuroService.

Je wijzigt daartoe de private variabele en de constructor van DefaultEuroService:

```
private final KoersClient[] koersClients; ❶
DefaultEuroService(KoersClient[] koersClients) { ❷
    this.koersClients = koersClients;
}
```

- (1) Je stelt de dependencies die KoersClient implementeren voor als een array.
(2) Spring injecteert alle dependencies die KoersClient implementeren als een KoersClient array met 2 elementen: een FixerKoersClient object en een ECBKoersClient object.

Je kan de volgorde waarmee Spring de array opvult beïnvloeden:

Je tikt @Order(1) voor de class FixerKoersClient. Je mag @Qualifier verwijderen.

Je tikt @Order(2) voor de class ECBKoersClient. Je mag @Qualifier verwijderen.

Gezien het getal bij @Order bij FixerKoersClient kleiner is dan dat bij ECBKoersClient, vult Spring de array met een FixerKoersClient object, gevolgd door een ECBKoersClient object.

Je hebt nu meerdere dependencies ter beschikking in DefaultEuroService.

Je gebruikt die in de method naarDollar. Je voegt eerst een variabele toe:

```
private static final Logger LOGGER =
    LoggerFactory.getLogger(DefaultEuroService.class);
```


Je wijzigt de method naarDollar:

```
@Override
public BigDecimal naarDollar(BigDecimal euro) {
    for (KoersClient koersClient : koersClients) {
        try {
            return euro.multiply(koersClient.getDollarKoers())
                .setScale(2, RoundingMode.HALF_UP);
        } catch (KoersClientException ex) {
            LOGGER.error("kan dollar koers niet lezen", ex);
        }
    }
    LOGGER.error("kan dollar koers van geen enkele bron lezen");
    return null;
}
```

- (1) Je itereert over de dependencies.
- (2) Je probeert de method `getDollarKoers` op te roepen.
Als dit lukt, gebruik je de waarde die je van deze method gekregen hebt.
Gezien op deze regel een `return` gebeurt, zal ook de iteratie bij ❶ stoppen.
- (3) Alle dependencies hebben een fout geworpen.
- (4) Je moet "iets" teruggeven, anders kan je de method niet compileren.

Je hebt nu nog een compilerfout in `EuroServiceTest`: je roept in de method `before` de `DefaultEuroService` constructor op en je geeft één `KoersClient` object mee.

De constructor verwacht nu echter een *array* van `KoersClient` objecten.

Je wijzigt de laatste opdracht in de method `before`:

```
euroService = new DefaultEuroService(new KoersClient[] {dummyKoersenClient});
```

Je voert de test uit. Hij lukt.

Je probeert de applicatie uit.

Als je de prijs in dollar vraagt van een pizza, zie je in het Console venster van Eclipse de tekst `koers gelezen via Fixer`.

Je maakt een tikfout in de Fixer api key in de URL in `FixerKoersClient` en je slaat het bestand op.

Als je de prijs in dollar vraagt van een pizza, zie je in het Console venster van Eclipse eerst een fout `kan koers niet lezen via Fixer`, gevolgd door de de tekst `koers gelezen via ECB`.

Je corrigeert de Fixer api key in `FixerKoersClient`.



Je commit de sources . Je publiceert op GitHub.



Sauzen.properties: zie takenbundel

26 APPLICATION.PROPERTIES

De url's van de Fixer website en van de ECB website zijn momenteel getikt in Java sources. Men zegt dan ook dat deze url's hard gecodeerd zijn. Dit is onhandig: zo'n URL kan wijzigen nadat de website af is. De persoon die de applicatie beheert moet dan de Java source openen, deze source compileren en terug een WAR bestand maken.

Het is mogelijk dat deze persoon geen Java kent, of geen Java editor bij de hand heeft.

Een betere oplossing is deze urls te tikken in `application.properties` en deze waarden te lezen in de classes `FixerDollarKoers` en `ECBDollarKoers`.

26.1 application.properties

Je voegt regels toe aan `application.properties`:

```
ecbKoersURL=https://www.ecb.europa.eu/stats/eurofxref/eurofxref-daily.xml
fixerKoersURL=http://data.fixer.io/api/latest?access_key=TikHierJeFreeKey&symbols=USD
```

26.2 Beans

Je wijzigt de constructor van `FixerKoersClient`:

```
FixerKoersClient(@Value("${fixerKoersURL}") URL url) {
    this.url = url;
}
```

❶

- (1) Je voorziet de constructor van een parameter. Je tikt voor die parameter `@Value`. Spring injecteert dan een instelling uit `application.properties` in die parameter. Je geeft aan `@Value` de naam van de instelling mee die Spring moet injecteren. Je geeft deze naam mee tussen `${}` en `}`. Dit heet een SpEL expressie (Spring expression language). Spring maakt een URL object gebaseerd op de waarde van de instelling, omdat je de parameter `url` beschreven hebt als een URL object. De constructor is veel eenvoudiger dan de vorige versie. Het is niet meer jij, maar Spring die een URL object maakt op basis van een stukje tekst (uit `application.properties`). Het is ook niet jij, maar Spring die een `MalformedURLException` opvangt als Spring het stukje tekst niet naar een URL object kan omzetten. Als die fout optreedt start je applicatie niet. Je ziet dit als je in `application.properties` de waarde van `fixerKoersURL` even wijzigt naar `hahaha`. Je ziet fout informatie in het venster Console. Deze fout informatie bevat onder andere `Could not retrieve URL for ServletContext resource [/hahah]: ServletContext resource [/hahah] cannot be resolved to URL because it does not exist.` Je herstelt de waarde van `fixerKoersURL` in `application.properties`.

Je wijzigt ook de constructor van `ECBKoersClient`:

```
public ECBKoersClient(@Value("${ecbKoersURL}") URL url) {
    this.url = url;
}
```

De unit tests `FixerKoersClientTest` en `ECBKoersClientTest` bevatten nu fouten.

Je lost deze op in het volgende hoofdstuk.

Je kan de applicatie uitproberen.



Je commit de sources. Je publiceert op GitHub.



Application.properties: zie takenbundel

27 INTEGRATION TEST VAN EEN SPRING BEAN

FixerKoersClientTest geeft nu een fout bij de oproep van de FixerKoersClient constructor. Deze constructor verwacht een parameterwaarde terwijl je in de test geen meegeeft. Het zou jammer zijn de Fixer URL hard te coderen in deze unit test, terwijl hij al is opgenomen in application.properties.

Een betere oplossing is Spring te laten samenwerken met JUnit. Spring maakt dan de beans van de applicatie en past er dependency injection op toe. Spring zal daarbij met @Value ook de Fixer URL injecteren in de constructor van de FixerKoersClient bean.

Je injecteert daarna deze bean in de unit test zodat je er testen kan op doen.

Je tikt volgende regels voor FixerKoersClientTest:

```
@RunWith(SpringRunner.class)           ❶
@Import(FixerKoersClient.class)         ❷
@PropertySource("application.properties") ❸
```

- (1) Je laat JUnit en Spring samenwerken.
- (2) Je laat Spring zijn IOC container maken met daarin enkel de bean van de class ECBKoersClient. Zo loopt de test snel.
- (3) De class ECBKoersClient leest een instelling uit application.properties via @Value. Spring gebruikt een interne bean om application.properties te verwerken. Je laadt met @PropertySource ook deze bean in de IOC container. De parameter is de naam van het te verwerken properties bestand.

Je verwijdert de method before.

Je tikt volgende regel voor de variabele client:

```
@Autowired                               ❶
```

- (1) Als je @Autowired tikt voor een private variabele, zoekt Spring een bean waarvan de class dezelfde is als deze van die variabele. Als Spring zo'n bean vindt, vult Spring de variabele met een reference naar deze bean.

Je tikt volgende regels voor ECBKoersClientTest:

```
@RunWith(SpringRunner.class)
@Import(ECBKoersClient.class)
@PropertySource("application.properties")
```

Je verwijdert de method before.

Je tikt volgende regel voor de variabele client:

```
@Autowired
```

Je kan de unit tests uitproberen.



Je commit de sources . Je publiceert op GitHub.

28 DATABASE – DATASOURCE

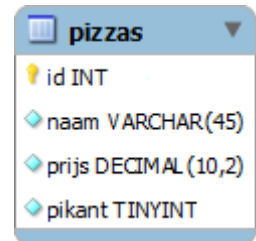
28.1 Database

Je voert in de MySQL Workbench het script `pizzaluigi.sql` uit.

Dit script maakt een database `pizzaluigi` en daarin een table `pizzas`.

- `id` is een auto increment kolom.
- `tinyint` is een synoniem voor boolean.

De gebruiker `cursist` heeft select en insert rechten in deze table.



28.2 DataSource

Je hebt een JDBC Connection nodig om een databasebewerking uit te voeren.

Een Connection openen vraagt tijd. Bij eenvoudige SQL statements kan het openen

van een Connection meer tijd vragen dan het uitvoeren van het SQL statement zelf.

Het is dus interessant om niet bij elke browser request een nieuwe Connection te maken, maar een bestaande Connection te herbruiken.

Java biedt hiertoe een object van het type DataSource aan. Synoniem is connection pool.

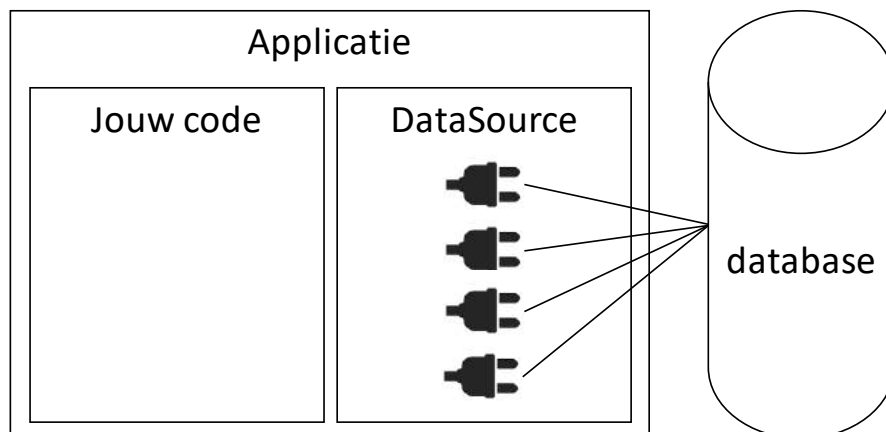
Dit is een verzameling Connection objecten die de applicatie continu openhoudt.

Standaard bevat een Spring Boot DataSource 10 connecties.

Je kan hiermee duizenden gelijktijdige gebruikers vlot bedienen.

Als je in je code een Connection nodig hebt, vraag je de DataSource één van zijn Connections.

Spring maakt van de DataSource een Spring bean bij het starten van de applicatie.



De werking van de DataSource:

1. Spring maakt de DataSource wanneer de website start. De DataSource opent enkele Connections naar de database en houdt ze bij in het RAM geheugen.
2. Een browser request komt binnen. De webserver voert met een thread je code uit.
Je vraagt in je Java code een Connection aan de DataSource.
Je krijgt de Connection zeer snel: ze stond al open naar de database.
De DataSource onthoudt dat die Connection in gebruik is.
3. Terwijl de webserver die request verwerkt, komt een andere request binnen.
De webserver verwerkt die request met een andere thread.
Je vraagt in de Java code een Connection aan de DataSource.
Die zoekt een Connection die niet in gebruik is en geeft die aan je.
Hij geeft je dus niet de Connection die de eerste thread nog in gebruik heeft.
Hij onthoudt ook nu dat de Connection, die hij je geeft, in gebruik is.
4. Je sluit in de code, uitgevoerd door de eerste thread, de Connection.
De DataSource onderschept dit sluiten echter en laat de Connection naar de database open.
Hij onthoudt wel dat die Connection niet meer in gebruik is.

Het is essentieel dat je een Connection sluit na gebruik. Anders zijn binnen de kortste keren alle Connection objecten van de DataSource in gebruik. Dan blokkeert de website.

De DataSource moet weten naar welke database hij connecties moet openen en met welke gebruiker. Je definieert dit met volgende regels in `application.properties`:

```
spring.datasource.url=jdbc:mysql://localhost/pizzaluigi?useSSL=false ❶
spring.datasource.username=cursist ❷
spring.datasource.password=cursist ❸
```

- (1) De JDBC URL die het soort database, de locatie en de naam van de database aangeeft.
- (2) De gebruikersnaam waarmee je connecties maakt.
- (3) Het bijbehorende paswoord.

Zodra `application.properties` deze instellingen bevat, maakt Spring bij het starten van je applicatie een bean van het type DataSource.

28.3 JDBC driver

Je voegt dependencies toe aan `pom.xml`:

- de dependency voor de MySQL JDBC driver
- de JDBC dependency waarmee je met de hulp van Spring gemakkelijk JDBC aanspreekt

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope> ❶
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

- (1) Je geeft aan dat je zelf in je code de classes uit de MySQL JDBC driver niet zal aanspreken. Deze classes moeten tijdens de uitvoering van het programma wel aanwezig zijn: de DataSource gebruikt deze classes om connecties te maken met de database.



Opmerking: als je een nieuwe website maakt kan je vinkjes plaatsen bij MySQL en bij JDBC in de website `start.spring.io` (waarmee je het project aanmaakt). Je moet dan deze dependencies niet meer met de hand toevoegen aan `pom.xml`. Zorg er wel voor dat `application.properties` de database instellingen bevat. Zoniet geeft je website fouten tijdens het starten.

28.4 Integration test

Je maakt in de map `src/test/java` een package `be.vdab.pizzaluigi.repositories` en daarin een class `DataSourceTest`:

```
package be.vdab.pizzaluigi.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@JdbcTest ❶
@AutoConfigureTestDatabase(replace = Replace.NONE) ❷
public class DataSourceTest {
  @Test
  public void goedGeïnitieerd() { ❸
  }
}
```

- (1) `@JdbcTest` maakt de test zo performant mogelijk: hij laadt enkel een DataSource bean en een JdbcTemplate bean. Je leert deze bean kennen in het volgende hoofdstuk.
- `@JdbcTest` laadt geen andere beans (zoals Controllers) die zinloos zijn in repositories.

- (2) `@JdbcTest` voert de tests standaard niet uit met de MySQL database (beschreven in `application.properties`).
`@JdbcTest` voert de tests standaard uit met een in-memory database.
Een in-memory database houdt de data bij in het interne geheugen, niet op de harde schijf.
Dit houdt in dat als je programma crasht of stopt je alle data verliest.
Dit is ontoelaatbaar in productie, gedurende een test kan het wel.
Het gebruik van een in-memory database versnelt je test.
Je zal in de cursus “Spring advanded” leren testen met een in-memory database.
Je test hier nog met je MySQL database.
Je geeft hier aan dat Spring in je test de MySQL database niet moet vervangen door een in-memory database.
- (3) JUnit wil een test class enkel uitvoeren als die minstens één method bevat voorzien van `@Test`.
De method moet in deze test geen code bevatten. `@JdbcTest` initialiseert de `DataSource`.
Deze maakt in zijn initialisatie database verbindingen.
De `DataSource` werpt exceptions als dit mislukt. Dan mislukt ook je test.
- Je probeert dit uit. Je tikt een fout in `application.properties`: je vervangt `jdbc` door `jbc`.
Je slaat het bestand op.
Je voert de test uit. Hij mislukt.
- Je corrigeert `application.properties`: je vervangt `jbc` terug door `jdbc`.
Je slaat het bestand op.
Je voert de test uit. Hij lukt.



`DataSource`: zie takenbundel

29 REPOSITORIES LAYER

Spring kan in de repositories layer samenwerken met meerdere database libraries: JDBC, JPA, ... Je leert hier dat Spring het gebruik van JDBC gemakkelijker maakt.



29.1 Exceptions

Je kan bij het aanspreken van de database twee soorten exceptions krijgen:

- Exceptions die je kan verhelpen zonder de applicatie te stoppen.
Voorbeeld: je probeert een record te wijzigen, maar het record is gelocked.
Je kan een aantal milliseconden wachten en het record opnieuw proberen te wijzigen.
Waarschijnlijk is het record ondertussen niet meer gelocked.
Je kan dit “opnieuw proberen” een aantal keer na mekaar doen.
- Fatale exceptions
Voorbeeld: je SQL statement bevat een syntaxfout.

JDBC maakt jammer genoeg geen verschil tussen verhelpbare en fatale exceptions: alle exceptions zijn van het type `SQLException`.

`SQLException` bevat een `int` property `errorCode` met meer detail over de exception. Jammer genoeg heeft dit getal niet bij alle merken databases dezelfde betekenis.

Spring vertaalt, aan de hand van die `errorCode` én het merk van de database, `SQLExceptions` naar specifiekere exceptions.

Je kan daarmee wel onderscheid maken tussen verhelpbare en fatale exceptions.

De belangrijkste Spring verhelpbare database exceptions:

- `DuplicateKeyException`
Een insert of een update veroorzaakte een fout, omdat je probeerde een dubbele waarde te maken op een index die geen dubbele waarden toelaat.
- `DataIntegrityViolationException`
Een insert of een update veroorzaakte een fout op een database constraint.
Je vulde bvb. een verplicht in te vullen kolom niet met een waarde.
- `CannotAcquireLockException`
Een record is gelocked door een andere applicatie.

De belangrijkste Spring fatale database exceptions:

- `BadSqlGrammarException`
Je SQL statement bevat een syntaxfout.
- `InvalidResultSetAccessException`
Je probeert uit een `ResultSet` een kolom te lezen die niet bestaat.
- `TypeMismatchDataException`
Een Java type komt niet overeen met een database type.
Je vult bvb. een date kolom met een double.
- `PermissionDeniedDataAccessException`
Je hebt niet genoeg rechten in de database om de handeling uit te voeren.
- `CannotGetJdbcConnection`
De verbinding met de database mislukt.

29.2 PizzaNietGevondenException

Je maakt een package `be.vdab.pizzaluigi.exceptions`.

Je maakt daarin `PizzaNietGevondenException`:

```
public class PizzaNietGevondenException extends RuntimeException {  
    private static final long serialVersionUID = 1L;  
}
```

29.3 Interface

Je beschrijft de functionaliteit van de repositories laag in een interface.:

```
package be.vdab.pizzaluigi.repositories;
// enkele imports
public interface PizzaRepository {
    void create(Pizza pizza);
    Optional<Pizza> read(long id);
    void update(Pizza pizza);
    void delete(long id);
    List<Pizza> findAll();
    List<Pizza> findByPrijsBetween(BigDecimal van, BigDecimal tot);
    long findAantalPizzas();
    List<BigDecimal> findUniekePrijzen();
    List<Pizza> findByPrijs(BigDecimal prijs);
}
```

- (1) Je voert deze method uit als de gebruiker een pizza wil toevoegen.
De parameter pizza is een Pizza object met de naam, ... van de toe te voegen pizza.
- (2) Je voert deze method uit als de gebruiker de gegevens van één pizza wil zien.
De parameter id is het nummer van de pizza die die de gebruiker wil zien.
- (3) Je voert deze method uit als de gebruiker een bestaande pizza wil wijzigen.
De parameter pizza is een Pizza object met de te wijzigen naam,
- (4) Je voert deze method uit als de gebruiker een pizza wil verwijderen.
De parameter id is het nummer van de pizza die de gebruiker wil verwijderen.
- (5) Je voert deze method uit als de gebruiker alle pizza's wil zien.
- (6) Je voert deze method uit als de gebruiker de pizza's wil zien waarvan de prijs ligt tussen een benedengrens (parameter van) en een bovengrens (parameter tot).
- (7) Je voert deze method uit als de gebruiker het aantal pizza's wil weten.
- (8) Je voert deze method uit als de gebruiker de unieke prijzen van de piza's wil weten.
- (9) Je voert deze method uit als de gebruikers de pizza's met één bepaalde prijs willen weten.

29.4 JdbcTemplate

Je JDBC code is eenvoudig, omdat de Spring class JdbcTemplate volgend werk voor je doet:

- Connections, Statements en ResultSets maken.
- itereren over ResultSets.
- Connections, Statements en ResultSets sluiten.

Als de applicatie start, maakt Spring een DataSource bean en daarna een JdbcTemplate bean.

Spring doet dit automatisch zodra je de JDBC dependency toevoegt aan pom.xml.

Spring injecteert daarbij de DataSource in de JdbcTemplate, zodat deze tijdens zijn werk een database connectie kan vragen aan de DataSource.

29.5 Repository bean

Je maakt een class JdbcPizzaRepository. Een object van die class zal een Spring bean zijn.

Je injecteert in die bean de JdbcTemplate bean (zelf geïnjecteerd met de DataSource bean):



```
package be.vdab.pizzaluigi.repositories;
// enkele imports
@Repository
class JdbcPizzaRepository implements PizzaRepository {
    private final JdbcTemplate template;
    JdbcPizzaRepository(JdbcTemplate template) {
        this.template = template;
    }
}
```


- (1) Je tikt `@Repository` voor de class. Spring maakt bij de start van de website een bean van die class. Spring houdt die bij bean zolang de website draait.
- (2) Je injecteert de `JdbcTemplate` bean die Spring maakt.

Je krijgt fouten: de class bevat de methods beschreven in de interface `PizzaRepository` nog niet. Je implementeert de methods één per één. Je leert daarbij de werking van `JdbcTemplate`.

29.5.1 Scalar value lezen

Het resultaat van een select statement met één rij met één kolom is een scalar value.

Je leest een scalar value met de `JdbcTemplate` method `queryForObject`:

```
private static final String SELECT_AANTAL_PIZZAS =
    "select count(*) from pizzas";
@Override
public long findAantalPizzas() {
    return template.queryForObject(SELECT_AANTAL_PIZZAS, Long.class); ❶
}
```

- (1) De 1° parameter is het select statement.
- (2) De 2° parameter is de gedaante waaronder je de scalar value wenst terug te krijgen als returnwaarde van de method `queryForObject`.

29.5.2 Update of delete SQL statement met één parameter

Je voert een update of delete SQL statement uit met de `JdbcTemplate` method `update`:

```
private static final String DELETE_PIZZA = "delete from pizzas where id=?"; ❶
@Override
public void delete(long id) {
    template.update(DELETE_PIZZA, id); ❷
}
```

- (1) ? duidt een parameter (veranderlijke waarde) aan in het SQL statement.
- (2) De 1° parameter is het SQL statement.
De 2° parameter is de waarde voor het ? in het SQL statement.
Als het SQL statement nog meer ? tekens bevat, geeft je nog meer parameters mee.

29.5.3 Update of delete SQL statement met meerdere parameters

```
private static final String UPDATE_PIZZA =
    "update pizzas set naam=?, prijs=?, pikant=? where id=?";
@Override
public void update(Pizza pizza) {
    if (template.update(UPDATE_PIZZA, pizza.getNaam(), pizza.getPrijs(),
        pizza.isPikant(), pizza.getId()) == 0) {
        throw new PizzaNietGevondenException(); ❶
    }
}
```

- (1) De method `update` geeft het aantal aangepaste records terug.
De method `update` geeft in het vorige voorbeeld het aantal verwijderde records terug.

29.5.4 Record toevoegen

Je voegt een record toe met de class `SimpleJdbcInsert`.

Je voegt een `SimpleJdbcInsert` member variabele toe:

```
private final SimpleJdbcInsert insert;
```

Je wijzigt de constructor:

```
JdbcPizzaRepository(JdbcTemplate template) {
    this.template = template;
    this.insert = new SimpleJdbcInsert(template); ❶
    insert.withTableName("pizzas"); ❷
    insert.usingGeneratedKeyColumns("id"); ❸
}
```

- (1) Je geeft de `JdbcTemplate` mee aan de `SimpleJdbcInsert` constructor.
Deze `JdbcTemplate` is zelf geïnjecteerd met een `DataSource`. De `SimpleJdbcInsert` gebruikt een connectie uit die `DataSource` tijdens het toevoegen van een record.
- (2) Je definieert de naam van de table waarin je records wil toevoegen.
- (3) Als de table een automatisch gegenereerde primary key kolom bevat, vermeld je met de method `usingGeneratedKeyColumns` de naam van die kolom.

Je maakt de method `create`:

```
@Override
public void create(Pizza pizza) {
    Map<String, Object> kolomWaarden = new HashMap<>();
    kolomWaarden.put("naam", pizza.getNaam());
    kolomWaarden.put("prijs", pizza.getPrijs());
    kolomWaarden.put("pikant", pizza.isPikant());
    Number id = insert.executeAndReturnKey(kolomWaarden);
    pizza.setId(id.longValue());
}
```

- (1) Deze Map bevat één entry per in te vullen kolom in het nieuwe record.
- (2) De key is de naam van een in te vullen kolom. De value is de waarde die je invult.
- (3) De method `executeAndReturnKey` voegt een record toe.
De parameter is een Map met kolomnamen en bijbehorende kolomwaarden.
De method maakt zelf een SQL insert statement en voert dit uit.
De method geeft de automatisch gegenereerde primary key waarde als een `Number`.
- (4) Je vult de property `id` van de `Pizza` entity met de gegenereerde primary key.
De controller kan deze waarde bijvoorbeeld lezen om ze te tonen aan de gebruiker.

29.5.5 RowMapper

Een `RowMapper`

- implementeert de functionele interface `RowMapper`.
- de method `mapRow` converteert één `ResultSet` rij naar één entity.
- `JdbcTemplate` gebruikt een `RowMapper` object bij het lezen van records.

De method `mapRow` heeft volgende eigenschappen:

- De 1° parameter is de `ResultSet` waarvan jij de huidige rij omzet naar één entity.
- De 2° parameter is het volgnummer van de huidige rij in de `ResultSet`.
- De returnwaarde is de entity waarnaar je het huidig record van de `ResultSet` omzet.
Het entity type is in de interface beschreven als een generisch type `T`.
Als je de interface implementeert, vervang je `T` door het entity type waarnaar je het huidig record van de `ResultSet` omzet. In dit voorbeeld is dit `Pizza`.

Je implementeert deze interface met een lambda in de class `JdbcPizzaRepository`:

```
private final RowMapper<Pizza> pizzaRowMapper = (resultSet, rowNum) ->
    new Pizza(resultSet.getLong("id"), resultSet.getString("naam"),
        resultSet.getBigDecimal("prijs"),
        resultSet.getBoolean("pikant"));
```

Je gebruikt die variabele als parameter van de leesoperaties van `JdbcTemplate`.

Die zetten gelezen records om naar entities met deze `RowMapper` en bieden je die entities aan.

De `JdbcTemplate` method `query` leest records uit een select statement.

Deze method heeft 2 parameters: een `String` met het select statement en een `RowMapper`.

De method voert het select statement uit en maakt op basis van de gelezen records, met de hulp van de `RowMapper`, een `List` van entities.

```
private static final String SELECT_ALL =
    "select id, naam, prijs, pikant from pizzas order by id";
@Override
public List<Pizza> findAll() {
    return template.query(SELECT_ALL, pizzaRowMapper);
}
```

- (1) De method query voert volgende stappen uit:
 - a. een Connection vragen aan de DataSource bean.
 - b. een Statement met het select statement maken en uitvoeren.
 - c. een List<Pizza> maken.
 - d. itereren over de ResultSet van het resultaat van het select statement.
 - e. per rij de rowMapper method mapRow uitvoeren.
 - f. de Pizza, die de mapRow method teruggeeft, toevoegen aan de List.
 - g. de ResultSet, het Statement en de Connection sluiten.
 - h. de List<Pizza> teruggeven.

De query method heeft ook een versie waarbij je een SQL statement met parameters uitvoert:

```
private static final String SELECT_BY_PRIJS_BETWEEN =
    "select id, naam, prijs, pikant from pizzas"
    + " where prijs between ? and ? order by prijs";

@Override
public List<Pizza> findByPrijsBetween(BigDecimal van, BigDecimal tot) {
    return template.query(SELECT_BY_PRIJS_BETWEEN, pizzaRowMapper, van, tot); ❶
}
```

- (1) De 1° parameter is een select statement. De 2° parameter is een RowMapper. De volgende parameters zijn waarden voor de parameters in het select statement (aangegeven met ?).

Je kan met de JdbcTemplate method queryForObject (die je al gebruikte in de method findAantalPizzas) één record lezen in de gedaante van een entity.

Als het select statement geen record vindt, of meer dan één record, werpt queryForObject een IncorrectResultSetSizeDataAccessException.

```
private static final String READ =
    "select id, naam, prijs, pikant from pizzas where id=?";

@Override
public Optional<Pizza> read(long id) {
    try {
        return Optional.of(template.queryForObject(READ, pizzaRowMapper, id));
    } catch (IncorrectResultSetSizeDataAccessException ex) {
        return Optional.empty();
    }
}
```

Je kan nu ook de resterende methods maken:

```
private final RowMapper<BigDecimal> prijsRowMapper =
    (resultSet, rowNum) -> resultSet.getBigDecimal("prijs");
private static final String SELECT_UNIEKE_PRIJZEN =
    "select distinct prijs from pizzas order by prijs";

@Override
public List<BigDecimal> findUniekePrijzen() {
    return template.query(SELECT_UNIEKE_PRIJZEN, prijsRowMapper);
}

private static final String SELECT_BY_PRIJS =
    "select id, naam, prijs, pikant from pizzas where prijs=? order by naam";

@Override
public List<Pizza> findByPrijs(BigDecimal prijs) {
    return template.query(SELECT_BY_PRIJS, pizzaRowMapper, prijs);
}
```

29.6 Integration test

Een *unit* test van een repository bean is zinloos: je kan de correcte werking van zo'n bean niet testen als hij geen verbinding maakt naar de database en er SQL statements naar stuurt.

Je doet een *integration* test: je test de repository bean met zijn samenwerking met de database.

Je hebt hierbij een probleem: de database onthoudt toevoegingen, wijzigingen en verwijderingen die je in de integration test doet. Dit houdt in dat als je de integration test een tweede keer doet, de eerste uitvoering de tweede uitvoering negatief beïnvloedt. Een voorbeeld:

- Je voegt in een test een pizza toe aan de database.
- De test slaagt wanneer je hem de eerste keer uitvoert.
- De tweede keer faalt de test, omdat de pizza al bestaat in de database.

De oplossing is als volgt:

1. Je start voor elke test een transactie.
2. Je voert de test uit.
3. Je doet na de test een rollback van de transactie. Je doet zo de bewerkingen die je in de test op de database deed ongedaan. Je kan op die manier toch je tests meerdere keren uitvoeren.

Je erft hiertoe de test class van `AbstractTransactionalJUnit4SpringContextTests`. Spring voert dan elke test uit in een transactie. Spring doet na de test een rollback op die transactie.

De integration test is zo geschreven dat het niet uitmaakt of de database reeds records bevat en hoeveel records de database bevat. Om het verwijderen of het wijzigen van een pizza te testen moet de database wel minstens 1 pizza bevat.

Je doet hiertoe voorbereidend werk:

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`.
Dit is de standaard Maven folder voor bestanden
 - a. die je enkel tijdens het testen nodig hebt.
 - b. en geen Java sources zijn.
4. Je kiest Finish.
5. Je klikt met de rechtermuisknop op die nieuwe folder `src/test/resources`.
6. Je kiest New, Other, SQL Development, SQL File en je kiest Next.
7. Je tikt `insertPizza` en je kiest Finish.
8. Je tikt volgende SQL opdracht in dit bestand:
`insert into pizzas(naam,prijs,pikant) values('test', 10, false);`
9. Je slaat het bestand op.

De integration test zelf:

```
package be.vdab.pizzaluigi.repositories;
// enkele imports
@RunWith(SpringRunner.class)
@JdbcTest
@AutoConfigureTestDatabase(replace = Replace.NONE)
@Import(JdbcPizzaRepository.class)
@Sql("/insertPizza.sql")
public class JdbcPizzaRepositoryTest
    extends AbstractTransactionalJUnit4SpringContextTests {
    private static final String PIZZAS = "pizzas";
    @Autowired
    private JdbcPizzaRepository repository;
    @Test
    public void findAantal() {
        assertEquals(super.countRowsInTable(PIZZAS),
            repository.findAantalPizzas());
    }
}
```

①
②
③
④
⑤
⑥

```

@Test
public void findAll() {
    List<Pizza> pizzas = repository.findAll();
    assertEquals(super.countRowsInTable(PIZZAS), pizzas.size());
    // zijn ze wel oplopend gesorteerd :
    long vorigeId = 0;
    for (Pizza pizza : pizzas) {
        assertTrue(pizza.getId() > vorigeId);
        vorigeId = pizza.getId();
    }
}

@Test
public void create() {
    int aantalPizzas = super.countRowsInTable(PIZZAS);
    Pizza pizza = new Pizza("test2", BigDecimal.TEN, false);
    repository.create(pizza);
    assertEquals(0, pizza.getId());
    assertEquals(aantalPizzas + 1, this.countRowsInTable(PIZZAS));
    assertEquals(1, super.countRowsInTableWhere(PIZZAS,
        "id=" + pizza.getId()));
}

private long idVanTestPizza() {
    return super.jdbcTemplate.queryForObject(
        "select id from pizzas where naam='test'", Long.class);
}

@Test
public void delete() {
    long id = idVanTestPizza();
    int aantalPizzas = super.countRowsInTable(PIZZAS);
    repository.delete(id);
    assertEquals(aantalPizzas - 1, super.countRowsInTable(PIZZAS));
    assertEquals(0, super.countRowsInTableWhere(PIZZAS, "id=" + id));
}

@Test
public void read() {
    assertEquals("test", repository.read(idVanTestPizza()).get().getNaam());
}

@Test
public void readOnbestaandePizza() {
    assertFalse(repository.read(-1).isPresent());
}

@Test
public void update() {
    long id = idVanTestPizza();
    Pizza pizza = new Pizza(id, "test", BigDecimal.ONE, false);
    repository.update(pizza);
    assertEquals(0, BigDecimal.ONE.compareTo(super.jdbcTemplate.queryForObject(
        "select prijs from pizzas where id=?", BigDecimal.class, id)));
}

@Test(expected = PizzaNietGevondenException.class)
public void updateOnbestaandePizza() {
    repository.update(new Pizza(-1, "test", BigDecimal.ONE, false));
}

@Test
public void findByPrijsBetween() {
    List<Pizza> pizzas =
        repository.findByPrijsBetween(BigDecimal.ONE, BigDecimal.TEN);
    BigDecimal vorigePrijs = BigDecimal.ZERO;

```

7
8
9

10
11

```

    for (Pizza pizza : pizzas) {
        assertTrue(pizza.getPrijs().compareTo(BigDecimal.ONE) >= 0);
        assertTrue(pizza.getPrijs().compareTo(BigDecimal.TEN) <= 0);
        assertTrue(vorigePrijs.compareTo(pizza.getPrijs()) <= 0);
        vorigePrijs = pizza.getPrijs();
    }
    assertEquals(super.countRowsInTableWhere(PIZZAS,
        "prijs between 1 and 10"), pizzas.size());
}
@Test
public void findUniekePrijzenGeeftPrijzenOplopend() {
    List<BigDecimal> prijzen = repository.findUniekePrijzen();
    long aantalPrijzen = super.jdbcTemplate.queryForObject(
        "select count(distinct prijs) from pizzas", Long.class);
    assertEquals(aantalPrijzen, prijzen.size());
    BigDecimal vorigePrijs = BigDecimal.valueOf(-1);
    for (BigDecimal prijs : prijzen) {
        assertTrue(prijs.compareTo(vorigePrijs) > 0);
        vorigePrijs = prijs;
    }
}
@Test
public void findByPrijs() {
    List<Pizza> pizzas = repository.findByPrijs(BigDecimal.TEN);
    String vorigeNaam = "";
    for (Pizza pizza : pizzas) {
        assertEquals(0, BigDecimal.TEN.compareTo(pizza.getPrijs()));
        assertTrue(vorigeNaam.compareTo(pizza.getNaam()) <= 0);
        vorigeNaam = pizza.getNaam();
    }
    assertEquals(super.countRowsInTableWhere(PIZZAS, "prijs=10"), pizzas.size());
}
}

```

- (1) @JdbcTest voert elke test uit in een eigen transactie.
@JdbcTest doet een rollback van die transactie op het einde van de test.
- (2) @JdbcTest voert de tests standaard niet uit met de database beschreven in application.properties : een MySQL database.
@JdbcTest voert de tests standaard uit met een in-memory database.
Een in-memory database houdt de data bij in het interne geheugen, niet op de harde schijf.
Dit houdt in dat als je programma crasht of stopt je alle data verliest.
Dit is ontoelaatbaar in productie, gedurende een test kan het wel.
Het gebruik van een in-memory database versnelt je test.
Je zal in de cursus "Spring advanced" leren testen met een in-memory database.
Je test hier nog met je MySQL database. Je vraagt hier aan Spring in je test de MySQL database niet te vervangen door een in-memory database.
- (3) Je maakt een bean van de te testen class: JdbcPizzaRepository.
- (4) Je verwijst naar het SQL bestand dat je vroeger in dit hoofdstuk maakte.
Spring voert de SQL opdracht(en) in dit bestand uit voor elke test.
Spring voert die opdrachten uit in dezelfde transactie waarin Spring ook de test zelf uitvoert.
Spring doe na de test een rollbakc van die transactie.
De SQL opdrachten uit het SQL bestand worden dus ook ongedaan gemaakt.
- (5) Je test erft van AbstractTransactionalJUnit4SpringContextTests.
Deze class bevat functionaliteit waarmee je gemakkelijk een repository test.
Je leert deze functionaliteit kennen in de code die volgt.
- (6) Je base class evat een method countRowsInTable. Je geeft de naam van een table mee.
Je krijgt het aantal records in die table als returnwaarde.
Je test met deze method of je eigen method findAantalPizzas correct werkt:

De method `countRowsInTable` moet hetzelfde aantal records teruggeven als je method `findAantalPizzas`.

- (7) Je controleert of de autonummering van de database het id van de nieuwe pizza invulde.
- (8) Het aantal records in de table `pizzas` moet met 1 verhogen na het toevoegen van een pizza.
- (9) Je base class bevat een method `countRowsInTableWhere`. Je geeft de naam van een table mee. Je geeft ook een voorwaarde mee waaraan de records moeten voldoen. Je krijgt het aantal records als returnwaarde.

Je test met deze method of de table `pizzas` een record bevat met dezelfde id als de id van de nieuw toegevoegde pizza.

- (10) Je gebruikt deze method verder in andere test methods.
- (11) Je base class bevat een variabele `jdbcTemplate`, van het type `JdbcTemplate`.
Je gebruikt die om vanuit je tests SQL statements uit te voeren.

Je kan de integration test uitvoeren.

Het is als ontwikkelaar interessant de SQL opdrachten te zien die Spring naar de database stuurt. Je voegt daartoe volgende regels toe aan `application.properties`:

```
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG ❶
logging.level.org.springframework.jdbc.core.simple.SimpleJdbcInsert=DEBUG ❷
logging.level.org.springframework.jdbc.core.StatementCreatorUtils=TRACE ❸
```

- (1) `JdbcTemplate` toont hiermee de SQL opdrachten die hij naar de database stuurt.
- (2) `SimpleJdbcInsert` toont hiermee de SQL opdrachten die hij naar de database stuurt.
- (3) Spring toont zo de waarden die hij invult in de ? tekens in SQL opdrachten met parameters.

Je slaat het bestand op.

Je voert de test `findByPrijsBetween` uit.

Je ziet in het Eclipse venster Console onder andere volgende informatie:

```
Executing prepared SQL statement [select id,naam,prijs,pikant from pizzas
where prijs between ? and ? order by prijs] ❶
Setting SQL statement parameter value: column index 1, parameter value [1],
value class [java.math.BigDecimal], SQL type unknown ❷
Setting SQL statement parameter value: column index 2, parameter value [10],
value class [java.math.BigDecimal], SQL type unknown ❸
```

- (1) Spring voert volgend SQL statement uit:
`select id,naam,prijs,pikant from pizzas where prijs between ? and ?
order by prijs`
- (2) Spring vult het eerste ? in deze SQL opdracht in met de waarde 1.
- (3) Spring vult het tweede ? in deze SQL opdracht in met de waarde 10.



Opmerking: je kan `@Sql(...)` meerdere keren tikken, telkens verwijzend naar een ander `.sql` bestand. Spring voert voor elke test alle SQL opdrachten in deze bestanden uit binnen dezelfde transactie als de test zelf.



Repositories: zie takenbundel

30 SERVICES EN TRANSACTIES

30.1 Interface

```
package be.vdab.pizzaluigi.services;
// enkele imports
public interface PizzaService {
    void create(Pizza pizza);
    Optional<Pizza> read(long id);
    void update(Pizza pizza);
    void delete(long id);
    List<Pizza> findAll();
    List<Pizza> findByPrijsBetween(BigDecimal van, BigDecimal tot);
    long findAantalPizzas();
    List<BigDecimal> findUniekePrijzen();
    List<Pizza> findByPrijs(BigDecimal prijs);
}
```

30.2 Implementatie

```
package be.vdab.pizzaluigi.services;
// enkele imports
@Service
class DefaultPizzaService implements PizzaService {
    private final PizzaRepository pizzaRepository;
    DefaultPizzaService(PizzaRepository pizzaRepository) {
        this.pizzaRepository = pizzaRepository;
    }
    @Override
    public void create(Pizza pizza) {
        pizzaRepository.create(pizza);
    }
    @Override
    public Optional<Pizza> read(long id) {
        return pizzaRepository.read(id);
    }
    @Override
    public void update(Pizza pizza) {
        pizzaRepository.update(pizza);
    }
    @Override
    public void delete(long id) {
        pizzaRepository.delete(id);
    }
    @Override
    public List<Pizza> findAll() {
        return pizzaRepository.findAll();
    }
    @Override
    public List<Pizza> findByPrijsBetween(BigDecimal van, BigDecimal tot) {
        return pizzaRepository.findByPrijsBetween(van, tot);
    }
    @Override
    public long findAantalPizzas() {
        return pizzaRepository.findAantalPizzas();
    }
    @Override
    public List<BigDecimal> findUniekePrijzen() {
        return pizzaRepository.findUniekePrijzen();
    }
}
```

❶

❷


```

@Override
public List<Pizza> findByPrijs(BigDecimal prijs) {
    return pizzaRepository.findByPrijs(prijs);
}
}

```

- (1) Je tikt `@Service` voor een class. Spring maakt bij het starten van de website een bean van deze class. Spring houdt die bean bij zolang de website draait.
- (2) Je injecteert de bean die de interface `PizzaRepository` implementeert: de bean gebaseerd op de class `JdbcPizzaRepository`.

30.3 Transactie eigenschappen

Je beheert transacties in de services laag. Je leert eerst de transactie eigenschappen kennen. De belangrijkste zijn isolation level, read-only, timeout, en propagation (later in dit hoofdstuk).

30.3.1 Isolation level

Het isolation level definieert hoe andere gelijktijdige transacties (van andere gebruikers) de huidige transactie beïnvloeden. Volgende problemen kunnen optreden bij gelijktijdige transacties:

- ⊖ Dirty read
Een transactie leest data die een tweede transactie wijzigde, maar nog niet committe. Als de tweede transactie rollbackt, heeft de eerste transactie verkeerde data gelezen.
- ⊖ Nonrepeatable read
Een transactie leest dezelfde data meerdere keren en krijgt per leesopdracht andere data. De oorzaak zijn andere transacties die tussen de leesoperaties van de eerste transactie dezelfde data wijzigen. De eerste transactie krijgt geen stabiel beeld van de gelezen data.
- ⊖ Phantom read
Een transactie leest dezelfde data meerdere keren en krijgt per leesopdracht meer records. De oorzaak zijn andere transacties die tussen de leesoperaties van de eerste transactie records toevoegen. De eerste transactie krijgt geen stabiel beeld van de gelezen data.

Je verhindert één of meerdere van deze problemen door het transaction isolation level in te stellen

↓ Isolation level ↓ van snel naar traag	Dirty read kan optreden	Nonrepeatable read kan optreden	Phantom read kan optreden
Read uncommitted	Ja	Ja	Ja
Read committed	Nee	Ja	Ja
Repeatable read	Nee	Nee	Ja
Serializable	Nee	Nee	Nee

Serializable lost alle problemen op maar is het traagste isolation level.

Je analyseert dus per use case welke problemen (dirty read, ...) die use case benadelen.

Je kiest daarna het isolation level dat juist gepast is om die problemen op te lossen.

Je gebruikt zelden read uncommitted, omdat het geen enkel probleem oplost.

Als je het isolation level niet instelt, krijgt de transactie het default isolation level van de database. Dit verschilt per merk database. Bij MySQL is het bijvoorbeeld Repeatable read.

30.3.2 Read-only

Als een transactie enkel records leest (niet toevoegt, wijzigt of verwijdert) maak je die transactie read-only. Spring voert dan op de JDBC connectie de method `setReadOnly(true)` uit. Sommige merken database optimaliseren dan de uitvoeringssnelheid van de transactie.

Als je in een read-only transactie toch records toevoegt, wijzigt of verwijdert, krijg je een exception.

Als je read-only niet instelt, is een transactie niet read-only.

30.3.3 Timeout

De database vergrendelt records tijdens het uitvoeren van bepaalde transacties. Als andere transacties die records aanspreken, wachten ze tot die eerste transacties de records ontgrendelen.

Je stelt met de timeout eigenschap in hoelang je transactie mag lopen.

Als de transactie langer loopt (omdat de transactie records aanspreekt die door andere transacties lang vergrendeld blijven), voert Spring automatisch een rollback uit op de transactie.

Als je de timeout niet instelt, krijg je de timeout van de achterliggende database.

Deze verschilt per merk database.

30.4 @Transactional

Je moet in je service layer geen code schrijven om een transactie te starten.

Als je voor een method `@Transactional` tikt, verzamelt Spring alle databasebewerkingen van die method in één transactie. Als je zo'n method oproept, start Spring eerst een transactie.

Op het einde van die method doet Spring automatisch

- een commit als de method geen exception werpte
- een rollback als de method een exception werpte

Je kan `@Transactional` tikken bij een class en/of bij methods van een class.

- Als je `@Transactional` tikt bij een class, is elke method in die class één transactie.
- Als je `@Transactional` tikt bij een method, is die method één transactie.
- Als je `@Transactional` tikt bij één class én bij een method van die class, overschrijven de transactie eigenschappen, beschreven bij de method, de transactie eigenschappen beschreven bij de class.

Je kan bij `@Transactional` de transactie eigenschappen instellen:

- `isolation` `READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`
- `readOnly` `true` of `false`
- `timeout` een aantal seconden

Je wijzigt de class `DefaultPizzaService`:

- Je schrijft voor de class
`@Transactional(readOnly = true , isolation = Isolation.READ_COMMITTED)`
- Je schrijft voor de methods `create`, `update` en `delete` volgende regel.
 Bemerk dat je niet enkel de parameter meegeeft die wijzigt ten opzichte van de class (`readOnly`), maar dat je ook de parameter moet meegeven die gelijk blijft (`isolation`)
`@Transactional(readOnly = false, isolation = Isolation.READ_COMMITTED)`

30.5 Propagation

Propagation is ook een transactie eigenschap die je kan meegeven aan `@Transactional`.

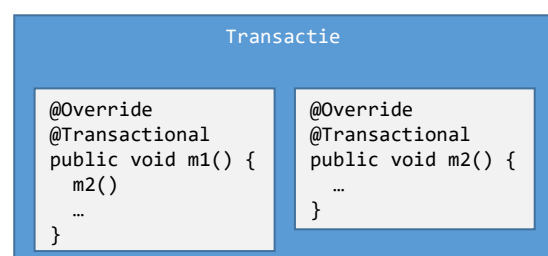
Je bepaalt met de propagation wat er gebeurt met een `@Transactional` method `m2`, als je die oproept vanuit een `@Transactional` method `m1`.

De meest gebruikte propagations:

REQUIRED (dit is de standaard propagation)

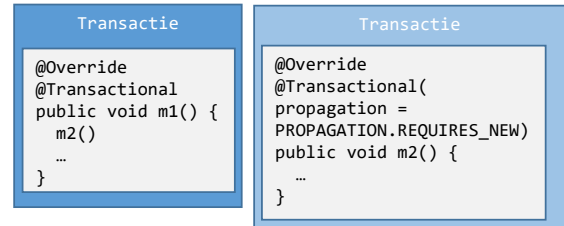
Spring voert de databasebewerkingen in method `m2` uit in de transactie die al loopt in method `m1`. De databasebewerkingen van beide methods behoren tot dezelfde transactie.

Als één van die methods een exception werpen, doet Spring een rollback op de transactie.



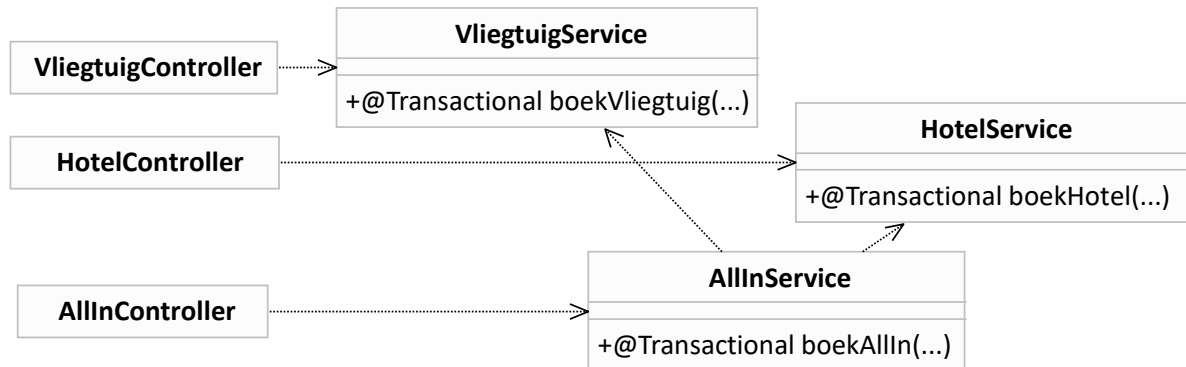
REQUIRES_NEW

Spring voert de method m2 uit in een nieuwe, aparte transactie. Tijdens het uitvoeren van method2 pauzeert Spring de transactie van method m1. Als m2 een exception werpt, doet Spring enkel een rollback op de transactie van m2.



Je leert in onderstaand voorbeeld (een reisbureau) dat REQUIRED handig is om een method uit de services laag

- apart te kunnen oproepen
- te kunnen combineren met andere methods uit de services laag tot één transactie



- **Gebruiker boekt een vliegtuigreis.**

VliegtuigController roept boekVliegtuig op in VliegtuigService.

```

@Override
@Transactional
public void boekVliegtuig(VliegtuigBoeking boeking) {
    // alle repository oproepen behoren tot een transactie
    if ( ! vliegtuigRepository.heeftPlaatsenVrijVoor(boeking) ) {
        throw new OnvoldoendeVliegtuigPlaatsenVrijException();
    }
    vliegtuigRepository.create(boeking);
}
  
```

- **Gebruiker boekt een hotelverblijf.**

HotelController roept boekHotel op in HotelService.

```

@Override
@Transactional
public void boekHotel(HotelBoeking boeking) {
    // alle repository oproepen behoren tot een transactie
    if ( ! hotelRepository.heeftBeddenVrijVoor(boeking) ) {
        throw new OnvoldoendeHotelBeddenVrijException();
    }
    hotelRepository.create(boeking);
}
  
```

- **Gebruiker boekt een all-in.**

```

@Override
@Transactional
// methods boekVliegtuig en boekHotel combineren tot één transactie:
public void boekAllIn(AllInBoeking boeking) {
    vliegtuigService.boekVliegtuig(boeking.getVliegtuigBoeking());
    hotelService.boekHotel(boeking.getHotelBoeking());
}
  
```

①
②
③

- (1) Spring start een transactie.
- (2) Spring start geen nieuwe transactie, maar voert de repository methods in boekVliegtuig uit in de transactie die gestart is bij de start (1) van boekAllIn.

- (3) Spring start ook hier geen nieuwe transactie, maar voert de repository methods in boekHotel uit in de transactie die al gestart is bij de start (1) van boekAllIn.

Als boekHotel een exception werpt, doet Spring een transactie rollback en doet ook de handelingen die boekVliegtuig uitvoerde ongedaan. Dit is goed: als je het hotel van een all-in niet kan boeken, mag ook het vliegtuig niet geboekt zijn.

30.6 Services laag oproepen in de controller

Nu rest nog service te injecteren in de controller en daar de service op te roepen:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Controller
@RequestMapping("pizzas")
class PizzaController {
    private static final String PIZZAS_VIEW = "pizzas";
    private final EuroService euroService;
    private final PizzaService pizzaService;
    PizzaController(EuroService euroService, PizzaService pizzaService) {
        this.euroService = euroService;
        this.pizzaService = pizzaService;
    }
    @GetMapping
    ModelAndView pizzas() {
        return new ModelAndView(PIZZAS_VIEW, "pizzas", pizzaService.findAll());
    }
    private static final String PIZZA_VIEW = "pizza";
    @GetMapping("{id}")
    ModelAndView pizza(@PathVariable long id) {
        ModelAndView modelAndView = new ModelAndView(PIZZA_VIEW);
        pizzaService.read(id).ifPresent(pizza -> {
            modelAndView.addObject(pizza);
            modelAndView.addObject("inDollar",
                euroService.naarDollar(pizza.getPrijs()));
        });
        return modelAndView;
    }
    @GetMapping("prijzen")
    ModelAndView prijzen() {
        return new ModelAndView(PRIJZEN_VIEW, "prijzen",
            pizzaService.findUniekePrijzen());
    }
    @GetMapping(params = "prijs")
    ModelAndView pizzasVanPrijs(BigDecimal prijs) {
        return new ModelAndView(PRIJZEN_VIEW, "pizzas",
            pizzaService.findByPrijs(prijs))
            .addObject("prijs", prijs)
            .addObject("prijzen", pizzaService.findUniekePrijzen());
    }
}
```

Deze wijzigingen vereisen ook enkele wijzigingen in `PizzaControllerTest`.

Je voegt een private variabele toe:

```
@Mock
```

```
private PizzaService dummyPizzaService;
```

Je wijzigt de before method:

```
@Before
```

```
public void before() {
    when(dummyPizzaService.read(1))
        .thenReturn(Optional.of(new Pizza(1, "Test", BigDecimal.ONE, true)));
    pizzaController = new PizzaController(dummyEuroService, dummyPizzaService);
}
```

Je wijzigt de `<c:forEach> ... </c:forEach>` in `pizzas.jsp`:

```
<c:forEach var='pizza' items='${pizzas}'>
<li>
${pizza.id}: <c:out value='${pizza.naam}' /> ${pizza.prijs}&euro;
<c:choose>
<c:when test='${pizza.pikant}'>pikant</c:when>
<c:otherwise>niet pikant</c:otherwise>
</c:choose>
<spring:url value='/pizzas/{id}' var='url'>
    <spring:param name='id' value='${pizza.id}' />
</spring:url>
<a href='${url}'>Detail</a>
</li>
</c:forEach>
```

Je kan de website uitproberen.



Spring maakt het gemakkelijker om de database aan te spreken.

Het blijft echter jouw verantwoordelijkheid om de database optimaal aan te spreken (zie cursus JDBC) en alles in goede banen te leiden als meerdere gebruikers tegelijk dezelfde data willen wijzigen.



Je commit de sources en je publiceert op GitHub.

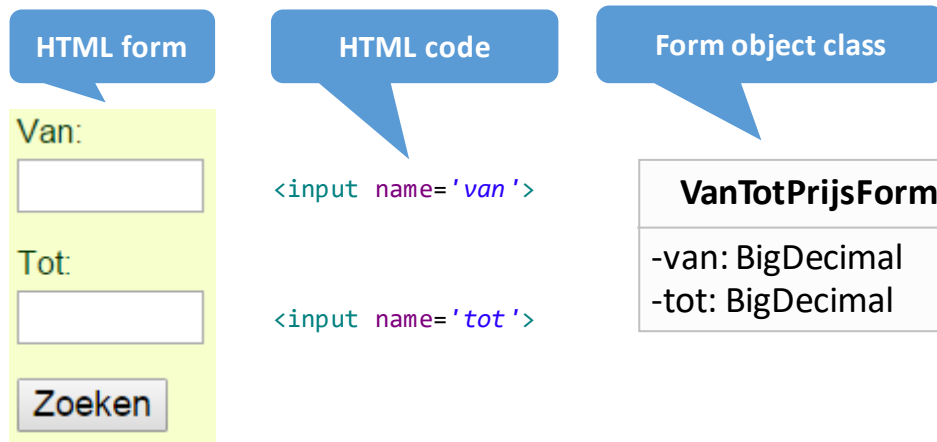


Services: zie takenbundel

31 HTML FORMS

31.1 Form object

Een form object is een Java object. De bijbehorende class stelt een HTML form voor.



De class bevat een private variabele per invoervak in de HTML form.

31.1.1 Form tonen aan de gebruiker

Je kan met een form object een HTML form tonen aan de gebruiker, waarbij je de invoervakken een beginwaarde geeft (bijvoorbeeld 10 en 20).

The diagram shows the same HTML form as before, but with pre-filled values in the input fields: '10' for 'Van:' and '20' for 'Tot:'. The 'Zoeken' button remains at the bottom.

Je doet dit met volgende stappen:

1. Je maakt een `VanTotPrijsForm` object.
2. Je vult de private variabele `van` met 10 en `tot` met 20.
3. Je geeft dit form object door aan Spring.
4. Spring maakt een HTML form, vult het vak `van` met de waarde in de variabele `van` en vult het vak `tot` met de waarde in de variabele `tot`. Dit heet data binding. De Spring class `DataBinder` doet dit werk.

31.1.2 Ingetikte waarden lezen

De gebruiker tikt bij `van` 3 en bij `tot` 4. Hij submit daarna de form.

Jij wil in je code de inhoud van de ingetikte invoervakken lezen. Dit gaat als volgt:

1. Spring verwerkt de browser request van de form submit en maakt een `VanTotPrijsForm` object.
2. Spring vult de private variabele `van` met 3 en `tot` met 4.
3. Spring biedt je dit object aan. Jij leest de waarden van de invoervakken, door in het `VanTotPrijsForm` object de variabelen `van` en `tot` te lezen.

31.1.3 Voorwaarden voor de command object class

- De naam van de class is vrij te kiezen.
- Er is een default constructor.
- Er is een private variabele (met getter en setter) per invoervak van de HTML form.

31.1.4 Package voor de command object class

- Als deze class enkel nuttig is als voorstelling van een HTML form, maak je deze class met package visibility in de package `be.vdab.pizzaluigi.web`.

- Als deze class nuttig is in andere applicatie lagen, maak je deze class met public visibility in de package `be.vdab.pizzaluigi.entities`.

31.1.5 Command object voorbeeld

Je maakt in `be.vdab.pizzaluigi.web` een class `VanTotPrijsForm`:

```
package be.vdab.pizzaluigi.web;
import java.math.BigDecimal;
class VanTotPrijsForm {
    private BigDecimal van;
    private BigDecimal tot;
}
```

Je maakt getters en setters voor de private variabelen.

31.2 Form tonen aan de gebruiker

Je toont de form bij een GET request naar de URL `/pizzas/vantotprijs`.

Je verwerkt die request in een nieuwe `PizzaController` method:

```
private static final String VAN_TOT_PRIJS_VIEW = "vantotprijs";
@GetMapping("vantotprijs")
ModelAndView findVanTotPrijs() {
    VanTotPrijsForm form = new VanTotPrijsForm();
    form.setVan(BigDecimal.ZERO);
    form.setTot(BigDecimal.ZERO);
    return new ModelAndView(VAN_TOT_PRIJS_VIEW).addObject(form);
}
```

- (1) Je geeft het form object aan de JSP onder de naam `vanTotPrijsForm`.

31.3 Spring form tag library

Je definieert in de JSP de form elementen met de Spring form tag library.

Die initialiseert de invoervakken van de form aan de hand van het formobject.

Je maakt `vantotprijs.jsp`:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
<%@taglib prefix='form' uri='http://www.springframework.org/tags/form'%>
<!doctype html>
<html lang='nl'>
<head>
<c:import url='/WEB-INF/JSP/head.jsp'>
    <c:param name='title' value='Van tot prijs'/>
</c:import>
</head>
<body>
<c:import url='/WEB-INF/JSP/menu.jsp'>
<h1>Van tot prijs</h1>
<c:url value='/pizzas' var='url'/>
<form:form action='${url}' modelAttribute='vanTotPrijsForm' method='get'>
    <form:label path='van'>Van:</form:label>
    <form:input path='van' autofocus='autofocus'/>
    <form:label path='tot'>Tot:</form:label>
    <form:input path='tot'/>
    <input type='submit' value='Zoeken'>
</form:form>
<c:if test='${not empty pizzas}'>
    <ul>
    <c:forEach items='${pizzas}' var='pizza'>
    <spring:url var='url' value='/pizzas/{id}'>
```

```

        <spring:param name='id' value='${pizza.id}'/>
    </spring:url>
    <li><a href='${url}'><c:out value='${pizza.naam}'/></a>(<out value='${pizza.prijs}'/></li>
</c:forEach>
</ul>
</c:if>
</body>
</html>

```

- (1) Je associeert de prefix form met de URL van de Spring form tag library.
- (2) Deze tag maakt een HTML tag `<form...>`.
 modelAttribute bevat de naam van het form object.
 Je tikt bij method get, omdat je met deze form enkel data zoekt, geen data wijzigt.
 Je geeft bij action de url aan waar de browser een request verstuurt als de gebruiker de form submit. De naam en inhoud van de tekstvakken komen dan achter op die url.
 Voorbeeld: /pizzas?van=3&tot=4
- (3) Deze tag maakt een HTML tag `<label for='van'>`.
- (4) Deze tag maakt een HTML tag
`<input name='van' id='van' value='0' autofocus='autofocus'>`
 De tag heeft 0 gelezen uit het form object via de method getVan.
 De tag verwacht dat je bij elk attribuut een waarde meegeeft met =,
 ook bij het attribuut autofocus. Je geeft dan de waarde 'autofocus' mee.
 Er bestaat naast `<form:input ...>` ook `<form:password ...>`, ...
- (5) De tag library bevat geen tag om een submit knop te maken.
 Je gebruikt de klassieke HTML tag `<input type='submit' ...>`.
- (6) Als de gebruiker de form submit, verwerk je de bijbehorende request.
 pizzas zal dan de opgevraagde pizza's bevatten.

31.4 Form verwerken na de submit

Als de gebruiker de form submit, verwerkt Spring deze browser request als volgt:

1. Spring maakt met de default constructor een form object.
2. Spring roept hierop setVan op en geeft de inhoud van het vak van mee.
3. Spring roept hierop setTot op en geeft de inhoud van het vak tot mee.

Je maakt in de class PizzaController een @GetMapping method die deze request verwerkt:

```

@GetMapping(params = {"van", "tot"})
 ModelAndView findVanTotPrijs(VanTotPrijsForm form) {
    return new ModelAndView(VAN_TOT_PRIJS_VIEW,
        "pizzas", pizzaService.findByPrijsBetween(
            form.getVan(), form.getTot());
}

```

- (1) De method verwerkt GET requests naar /pizzas,
 op voorwaarde dat die de request parameters van en tot bevat.
- (2) De method heeft een VanTotPrijsForm parameter.
 Spring ziet dit en maakt een VanTotPrijsForm object met de default constructor.
 Spring ziet dat de request een parameter van bevat.
 Spring ziet dat de class VanTotPrijsForm een method setVan bevat.
 Spring roept setVan op en geeft de waarde van de request parameter van mee.
 Spring ziet dat de request ook een parameter tot bevat.
 Spring ziet dat de class VanTotPrijsForm een method setTot bevat.
 Spring roept setTot op en geeft de waarde van de request parameter tot mee.
 Daarna roept Spring de huidige method findVanTotprijs op
 en geeft zijn opgevuld VanTotPrijsForm object mee.
- (3) Je zoekt de pizza's met de PizzaService method findByPrijsBetween.
 Je geeft deze pizza's onder de naam pizzas door aan de JSP.

Je kan de website uitproberen.

31.5 Lege invoervakken tonen

Je initialiseert in de method `findVanTotPrijs()` de properties van het form object:

```
form.setVan(BigDecimal.ZERO);
form.setTot(BigDecimal.ZERO);
```

Het resultaat hiervan is dat als de gebruiker in de pagina binnenkomt, de twee invoervakken de waarde nul bevatten. Als je liever hebt dat deze invoervakken leeg zijn, verwijder je deze regels. De variabelen van en tot zullen nu null bevatten in het form object. Spring toont dan lege bijbehorende invoervakken.

31.6 Invoer valideren

Het `DataBinder` object (dat de data binding doet) helpt je invoervakken te valideren.

Je zal in het volgende hoofdstuk volgende validaties leren:

- De invoervakken zijn verplicht in te vullen.
- De invoervakken mogen geen negatieve waarden bevatten.

31.6.1 Verkeerd type data

De gebruiker moet in sommige vakken geen tekst tikken, maar een getal, een datum, ...

De `DataBinder` kan de tekst van het invoervak niet doorgeven aan de form object setter, als de conversie naar het setter parameter type mislukt.

Als de gebruiker blabla tikt in het vak van, kan de `DataBinder` de tekst niet converteren naar `BigDecimal` (het parametertype van de bijbehorende method `setVan`).

Als de conversie mislukt, kom je dit te weten in de method die de request (veroorzaakt door de submit van de form) verwerkt. Je doet dit in de tweede method `findVanTotprijs`:

```
@GetMapping(params = {"van", "tot"})
ModelAndView findVanTotPrijs(VanTotPrijsForm form,
    BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return new ModelAndView(VAN_TOT_PRIJS_VIEW);
    }
    return new ModelAndView(VAN_TOT_PRIJS_VIEW, "pizzas",
        pizzaService.findByPrijsBetween(form.getVan(), form.getTot()));
}
```

- (1) Je kan het mislukken van de conversie nazien in een object van het type `BindingResult`. Deze `BindingResult` parameter moet volgen op de parameter met het form object.
- (2) De method `hasErrors` geeft true als de `DataBinder` fouten ontdekte bij de form validatie.
- (3) Je zoekt enkel pizza's als er geen fouten waren.

31.6.2 Foutboodschappen

Je toont in `vantotprijs.jsp` per validatiefout een boodschap aan de gebruiker.

Je tik na de tekst Van: `<form:errors path='van'/>`

- (1) `form:errors` toont een boodschap die hoort bij van, als van foutief is.

Je tikt na de tekst Tot: `<form:errors path='tot'/>`

Je kan de website uitproberen. Als je blabla tikt in het vak van, zie je de standaard foutboodschap van Spring: Failed to convert property value of type java.lang.String to required type java.math.BigDecimal for property van; nested exception is java.lang.NumberFormatException

Je overschrijft deze foutboodschap in een bestand `messages.properties`.

Je maakt dit bestand aan:

1. Je klikt met de rechtermuisknop op de map `src/main/resources`.
2. Je kiest New, Other, General, File.
3. Je tikt `messages.properties` bij File name en je kiest Finish.

4. Standaard kan een properties bestand geen geaccentueerde tekens (ë, ...) bevatten. `application.properties` bevat technische instellingen. Deze bevatten geen geaccentueerde tekens. `messages.properties` bevat menselijke teksten. Deze kunnen wel geaccentueerde tekens bevatten. Je doet daarom volgende stappen:
5. Je klikt met de rechtermuisknop op `messages.properties`.
6. Je kiest Properties.
7. Je kiest Other.
8. Je kiest daarnaast UTF-8.
9. Je kiest Apply and Close.
10. Je kiest Yes.

Zoals elk properties bestand bevat dit bestand regels met een key, een = teken en een waarde. Spring zoekt in `messages.properties` één per één volgende keys, tot hij er één vindt.

1. `typeMismatch.naamCommandObject.naamVak` (`typeMismatch.vanTotPrijsForm.van`)
2. `typeMismatch.naamVak` (`typeMismatch.van`).
3. `typeMismatch.typeVariabeleDieBijVakHoort` (`typeMismatch.java.math.BigDecimal`)
4. `typeMismatch`

Je voegt een regel toe aan `messages.properties`:

`typeMismatch.java.math.BigDecimal=tik een bedrag`

Je kan de website uitproberen.

31.6.3 Foutboodschappen maken in de controller bean

Je kan foutboodschappen maken in de controller method die de form submit verwerkt.

`BindingResult` bevat daarvoor enkele methods.

- De methods hebben een parameter `errorCode` met een key uit `messages.properties`.
- Sommige methods hebben een parameter `field` met de naam van het foute vak.
- Sommige methods hebben een array parameter `errorArgs`. Spring vervangt in de foutboodschap `{0}` door het 1^o array element, `{1}` door het tweede array element, ...

De methods zijn:

- `reject(String errorCode)`
Je maakt een algemene boodschap die niet met één vak verbonden is.
Je toont in de JSP dit soort foutboodschappen met de tag `<form:errors/>`.
- `reject(String errorCode, String defaultMessage)`
Zie vorige method. Als Spring de `errorCode` niet vindt in `teksten.properties`, gebruikt Spring `defaultMessage` als tekst.
- `reject(String errorCode, Object[] errorArgs, String defaultMessage)`
Zie vorige method. Spring vult `{0}`, `{1}`, ... in de tekst met waarden uit `errorArgs`.
- `rejectValue(String field, String errorCode)`
Je maakt een boodschap die verbonden is met het vak vermeld bij `field`.
Je toont deze boodschap met de tag `<form:errors path='naamPrivateVarBijVak' />`
- `rejectValue(String field, String errorCode, String defaultMessage)`
Zie vorige method. Als Spring de `errorCode` niet vindt in `teksten.properties`, gebruikt Spring `defaultMessage` als tekst.
- `rejectValue(String field, String errorCode, Object[] errorArgs, String defaultMessage)`
Zie vorige method. Spring vult `{0}`, `{1}`, ... in de tekst met waarden uit `errorArgs`.

Je toont als voorbeeld een foutmelding als je geen pizza's vindt tussen van en tot.

Je voegt een regel toe aan `messages.properties`:

`geenPizzas=geen pizza's gevonden`

Je wijzigt de 2° method findByVanTotPrijs:

```
@GetMapping(params = { "van", "tot" })
ModelAndView findVanTotPrijs(VanTotPrijsForm form, BindingResult bindingResult){
    ModelAndView modelAndView = new ModelAndView(VAN_TOT_PRIJS_VIEW);
    if (bindingResult.hasErrors()) {
        return modelAndView;
    }
    List<Pizza> pizzas =
        pizzaService.findByPrijsBetween(form.getVan(), form.getTot());
    if (pizzas.isEmpty()) {
        bindingResult.reject("geenPizzas");
    } else {
        modelAndView.addObject("pizzas", pizzas);
    }
    return modelAndView;
}
```

- (1) Je maakt een algemene boodschap die niet aan één vak verbonden is.
De key van de foutboodschap is geenPizzas.

Je voegt code toe in vantotprijs.jsp, na `<input type='submit' value='Zoeken'>`
`<form:errors cssClass='fout'>`

- (1) Je toont met `<form:errors ...>` (zonder path attribuut) foutmeldingen die niet aan één vak verbonden zijn. Je geeft met `cssClass='fout'` aan dat Spring de CSS class fout moet toepassen op de span waarin Spring deze foutmeldingen toont.

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.

32 BEAN VALIDATION

Bean validation is een officiële Java specificatie waarmee je objecten valideert.



32.1 Annotations

Je beschrijft de validatie met een annotation voor de te valideren private variabele. Voorbeeld:

```
public class Gemeente {
    @Min(1000)
    private short postcode;
}
```

1

(1) Je beschrijft een validatie voor de variabele `postcode`: de inhoud is ≥ 1000 .

Je kan bij één variabele meerdere validation annotations schrijven:

```
@Min(1000)
@Max(99999)
private short postcode; // moet dus een waarde tussen 1000 en 9999 bevatten
```

Bean validation bevat naast `@Min` en `@Max` volgende standaard annotations:

Annotation	Betekenis
<code>@DecimalMin(minimumWaarde)</code>	Zoals <code>@Min</code> , maar je geeft <code>minimumWaarde</code> als een String. Je gebruikt <code>@DecimalMin</code> bij een <code>BigDecimal</code> variabele, als de <code>minimumWaarde</code> cijfers na de komma bevat.
<code>@DecimalMax(maximumWaarde)</code>	Zoals <code>@Max</code> , maar je geeft <code>maximumWaarde</code> als een String.
<code>@Digits(integer=voorKomma, fraction=naKomma)</code>	De variabele heeft max. <code>voorKomma</code> cijfers voor de komma en max. <code>naKomma</code> cijfers na de komma.
<code>@Email</code>	De variabele moet de structuur van een e-mailadres hebben.
<code>@NotBlank</code>	De String variabele mag niet null zijn en moet meer dan enkel spaties bevatten.
<code>@NotEmpty</code>	De String variabele mag niet null zijn en mag niet leeg zijn. De variabele (array, List, Set, Map) mag niet null zijn en mag niet leeg zijn.
<code>@Negative</code>	De variabele moet een negatief getal zijn.
<code>@NegativeOrZero</code>	De variabele moet een negatief getal of 0 zijn.
<code>@Positive</code>	De variabele moet een positief getal zijn.
<code>@PositiveOrZero</code>	De variabele moet een positief getal of 0 zijn.
<code>@NotNull</code>	De variabele mag niet null bevatten.
<code>@Null</code>	De variabele moet null bevatten.
<code>@Future</code>	De variabele moet in de toekomst liggen.
<code>@FutureOrPresent</code>	De variabele moet vandaag zijn of in de toekomst liggen.
<code>@Past</code>	De variabele moet in het verleden liggen.
<code>@PastOrPresent</code>	De variabele moet vandaag zijn of in het verleden liggen.
<code>@Pattern(regex=regularExpression)</code>	De String variabele moet passen bij <code>regularExpression</code> .
<code>@Size(min=min, max=max)</code>	<ul style="list-style-type: none"> Het aantal tekens in de String variabele moet liggen tussen <code>min</code> en <code>max</code> of. Het aantal elementen in de variabele (array, List, Set, Map) moet liggen tussen <code>min</code> en <code>max</code>. <p>Als je de param. <code>min</code> weglaat, krijgt die de waarde 0. Als je de param. <code>max</code> weglaat, krijgt die de waarde $2^{31}-1$.</p>

De annotations (behalve `@NotBlank` en `@NotEmpty`) valideren een variabele enkel als die verschilt van null.

Er bestaan meerdere implementaties van de Bean validation specificatie.

De Hibernate implementatie (Hibernate validator) bevat extra annotations. De interessantste:

Annotation	Betekenis
<code>@CreditCardNumber</code>	De String variabele moet de structuur en het controlegetal hebben van een betaalkaartnummer.
<code>@EAN</code>	De String variabele moet de structuur en het controlegetal hebben van een European Article Number (zoals je ziet op een barcode).
<code>@Length(min=min, max=max)</code>	De String variabele bevat minstens min en maximaal max tekens.
<code>@Range(min=min, max=max)</code>	De getal variabele heeft een waarde tussen min en max.

32.2 @Valid

Je schrijft `@Valid` bij een variabele die verwijst naar een object dat zelf ook properties met validation annotations bevat.

Bean validation valideert dan ook die geneste validation annotations.

Voorbeeld:

```
public class Persoon {
    @Range(min = 0, max = 69)
    private int aantalKinderen;
    @Valid
    private Adres adres;
}

public class Adres {
    @Range(min = 1000, max = 9999)
    private int postcode;
}
```

m 1

- (1) Bean validation valideert bij de validatie van een Persoon object ook zijn Adres object en controleert dus of postcode ligt tussen 1000 en 9999.

Je kan `@Valid` ook toepassen op een verzameling objecten.

Bean validation valideert dan alle objecten in die verzameling. Voorbeeld:

```
@Valid private Set<Adres> adressen; // valideer alle Adres objecten in de Set
```

32.3 Foutboodschappen

Bean validation zoekt de foutboodschappen in `ValidationMessages.properties`.

Je maakt dit bestand:

1. Je klikt met de rechtermuisknop op de map `src/main/resources`.
2. Je kiest New, Other, General, File.
3. Je tikt `ValidationMessages.properties` bij File name en je kiest Finish.
4. Je klikt met de rechtermuisknop op `ValidationMessages.properties`.
5. Je kiest Properties.
6. Je kiest Other.
7. Je kiest daarnaast UTF-8.
8. Je kiest Apply and Close.
9. Je kiest Yes.

Je tikt volgende regels in dit bestand:

```
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 {regex}
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}

```

Bean validation vervangt de woorden {value}, {min}, {max}, ... in de teksten door de bijbehorende parameters die je bij de validation annotation tikte.

32.4 Valideren met @Valid

De validation annotations beschrijven wel de validaties, maar voeren zelf geen validaties uit. Je kan bijvoorbeeld een Pizza object maken en prijs op -7 plaatsen.

Je valideert een form object in een @GetMapping of @PostMapping method van een controller bean. Je schrijft @Valid bij dit form object. Het valideren gaat als volgt:

1. De gebruiker vult de vakken van de HTML form in.
2. Hij submit de form. Spring verwerkt de bijbehorende request. Spring ziet dat hij daarbij een @GetMapping of @PostMapping method van een controller bean moet oproepen. Deze method heeft een parameter met als type het form object. Voor deze parameter staat @Valid.
3. Spring maakt een form object met de default constructor.
4. Spring vult de form object properties met de inhoud van de vakken. Sommige properties bevatten dan een waarde die verkeerd is ten opzichte van hun validation annotations.
5. Spring roept de @GetMapping of @PostMapping method op die hoort bij de URL waarnaar de form submit.
6. De method heeft een parameter die het form object voorstelt. Je tikt @Valid vóór deze parameter. Spring valideert dan de properties van dit form object ten opzichte van hun validation annotation.
7. De BindingResult method hasErrors geeft true terug als er validatiefouten waren.

32.5 Form object class

Je gebruikt bean validation in de class VanTotPrijsForm.

Je tikt voor de variabele van én voor de variabele tot: @NotNull @PositiveOrZero.

32.6 Controller class wijzigingen

Je wijzigt de class PizzaController.

Je wijzigt de declaratie van de 2° versie van de method findVanTotPrijs: ModelAndView findVanTotPrijs(@Valid VanTotPrijsForm form, BindingResult bindingResult)

@Valid doet bean validation op deze parameter en vult de parameter bindingResult met het resultaat van deze validatie.

Je kan de website uitproberen.

32.7 Unit test

Een ontwikkelaar automatiseert taken. Het is logisch dat hij ook zijn eigen taken automatiseert. Je hebt zopas handmatig gecontroleerd of de class VanTotPrijsForm de juiste validation annotations bevat: je startte de website en je tikte verkeerde waarden in de vakken van en tot. Je vervangt hier deze handmatige controle door een geautomatiseerde controle.

Je voegt hiertoe een unit test toe aan src/test/java:

```
package be.vdab.pizzaluigi.web;
// enkele imports
public class VanTotPrijsFormTest {
    private Validator validator;
    @Before
    public void before() {
        ValidatorFactory factory = Validation.buildDefaultValidatorFactory();
        validator = factory.getValidator();
    }
    @Test
    public void vanOk() {
        assertTrue(validator.validateValue(
            VanTotPrijsForm.class, "van", BigDecimal.ONE).isEmpty());
    }
    @Test
    public void vanMoetIngeVuldZijn() {
        assertFalse(validator.validateValue(
            VanTotPrijsForm.class, "van", null).isEmpty());
    }
    @Test
    public void vanMoetMinstensNulZijn() {
        assertFalse(validator.validateValue(
            VanTotPrijsForm.class, "van", BigDecimal.valueOf(-1)).isEmpty());
    }
    @Test
    public void totOk() {
        assertTrue(validator.validateValue(
            VanTotPrijsForm.class, "tot", BigDecimal.ONE).isEmpty());
    }
    @Test
    public void totMoetIngeVuldZijn() {
        assertFalse(validator.validateValue(
            VanTotPrijsForm.class, "tot", null).isEmpty());
    }
    @Test
    public void totnMoetMinstensNulZijn() {
        assertFalse(validator.validateValue(
            VanTotPrijsForm.class, "tot", BigDecimal.valueOf(-1)).isEmpty());
    }
}
```

- (1) Je valideert met een Validator een object voorzien van validation annotations. Je zal deze Validator in deze test gebruiken om een VanTotPrijsForm object te valideren.
- (2) De static method buildDefaultValidatorFactory van de class Validation geeft je een ValidatorFactory object.
- (3) De method getValidator van ValidatorFactory geeft je een Validator object. Dit is een voorbeeld van het factory design pattern.

- (4) Je valideert met de method `validateValue` één eigenschap van een `VanTotPrijsForm` object. De 1° parameter is de class `VanTotPrijsForm`. De method `validateValue` maakt een object van die class. De 2° parameter is "van". Dit duidt aan dat je enkel de eigenschap van in het `VanTotPrijsForm` object valideert. De 3° parameter is de waarde die je wil invullen in die van eigenschap vooraleer te valideren. Na het valideren geeft de method `validateValue` een verzameling terug met de fouten die het valideren detecteerde. Gezien je van correct invulde moet deze verzameling leeg zijn.
- (5) Je geeft een verkeerde waarde mee voor de eigenschap van. De verzameling met fouten na het valideren mag dus zeker niet leeg zijn.

Je voert de test uit. Hij lukt.



Je commit de sources. Je publiceert op GitHub.



Begin naam: zie takenbundel

33 CLIENT SIDED VALIDATIE

Je doet de validatie tot nu aan de kant van de server.

Dit soort validatie is essentieel omdat ze door een hacker ze niet kan manipuleren.

Je kan daarnaast ook client sided validatie doen: validatie op de browser. Deze heeft als voordeel dat ze zeer interactief is: client sided validation gebeurt voor het submitten van de form.

De browser moet voor deze validaties dus niets naar de website doorsturen.

Het is wel gemakkelijk voor een hacker om client sided validatie te verwijderen.

Je doet daarom zowel server sided validatie als client sided validatie.

Je voegt met HTML 5 attributen client sided validation toe in `vantotprijs.jsp`:

```
<form:input path='van' autofocus='autofocus' type='number' required='required'
min='0'/>
<form:input path='tot' required='required' type='number' min='0'/>
```

Je kan de website uitproberen.

Als je een verkeerde waarde tikt, toont de browser een foutmelding.

De tekst in deze foutmelding is hard gecodeerd. Je kan deze tekst niet wijzigen.



Je commit de sources. Je publiceert op GitHub.

Omdat je client sided validatie hebt toegevoegd, is het moeilijk de server sided validatie te testen.

Je kan dit toch, door tijdelijk (voor de huidige request) de client sided validatie te verwijderen:

1. Je voert de website uit met FireFox.
2. Je klikt met de rechtermuisknop in in invoervak van.
3. Je kiest Inspect Element.
4. Je dubbelklikt `required="required"` (onder in het scherm) en je verwijdert dit onderdeel.
5. Je doet hetzelfde bij `min="0"`
6. Je vult de invoervakken in.
7. Je submit de form.



Client sided validatie: zie takenbundel

34 FORM MET METHOD = POST

Als de gebruiker een form met method='post' submit, verstuurt de browser een POST request.

Je gebruikt een form met method='post' als

- je bij het submitten van die form data toevoegt, wijzigt of verwijdt.
- je een paswoord opvraagt in die form.
- je een bestand uploadt in die form.

Je zal als voorbeeld een pagina maken waarmee de gebruiker een pizza toevoegt aan de database.

Je voegt eerst validation annotations toe aan de class Pizza:

- Je tikt @NotBlank voor de private variabele naam (uit javax.validation.constraints)
- Je tikt @NotNull en @PositiveOrZero voor de private variabele prijs.

Als de gebruiker in het menu Toevoegen kiest, doet hij een GET request naar /pizzas/toevoegen.

Je toont dan de toevoegpagina.

Je maakt daartoe een method in PizzaController:

```
private static final String TOEVOEGEN_VIEW = "toevoegen";
@GetMapping("toevoegen")
ModelAndView toevoegen() {
    return new ModelAndView(TOEVOEGEN_VIEW).addObject(new Pizza());
}
```

(1) Je geeft een Pizza object door aan de JSP. Je zal dit object daar gebruiken als form object.

Je maakt toevoegen.jsp:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
<%@taglib prefix='form' uri='http://www.springframework.org/tags/form'%>
<!doctype html>
<html lang='nl'>
<head>
    <c:import url='/WEB-INF/JSP/head.jsp'>
        <c:param name='title' value='Toevoegen'/>
    </c:import>
</head>
<body>
<c:import url='/WEB-INF/JSP/menu.jsp'/>
<h1>Toevoegen</h1>
<c:url value='/pizzas/toevoegen' var='url'/>
<form:form action='${url}' modelAttribute='pizza' method='post' id='pizzaform'>
    <form:label path='naam'>Naam: <form:errors path='naam'/></form:label>
    <form:input path='naam' autofocus='autofocus' required='required'/>
    <form:label path='prijs'>Prijs: <form:errors path='prijs'/></form:label>
    <form:input path='prijs' type='number' required='required' min='0'/>
    <form:checkbox path='pikant'/><form:label path='pikant'>Pikant</form:label>
    <input type='submit' value='Toevoegen' id='toevoegknop'>
</form:form></body></html>
```

Wanneer de gebruiker de form submit, stuurt de browser een POST request naar de webserver.

Je maakt een method die deze request verwerkt in PizzaController:

```
@PostMapping("toevoegen")
ModelAndView toevoegen(@Valid Pizza pizza, BindingResult bindingResult) {
    if (bindingResult.hasErrors()) {
        return new ModelAndView(TOEVOEGEN_VIEW);
    }
    pizzaService.create(pizza);
    return new ModelAndView(PIZZAS_VIEW, "pizzas", pizzaService.findAll());
}
```

- (1) Je geeft aan dat de method die op deze regel volgt POST requests verwerkt. De method verwerkt POST requests naar de URL vermeld bij @RequestMapping (pizzas), gecombineerd met de URL vermeld bij @PostMapping (toevoegen): pizzas/toevoegen.
- (2) Als er validatiefouten zijn toon je de de pagina met de form opnieuw.
- (3) Je voegt de pizza toe aan de database.
- (4) Je toont de pagina met alle pizzas.


Je kan de website uitproberen.

34.1 Het refresh probleem en POST-REDIRECT-GET

Je ziet het refresh probleem met volgende stappen:

1. Je voegt een pizza toe in de website. Je ziet daarna een lijst met alle pizza's.
De browser adresbalk bevat nog de URL van de POST request:
`http://localhost:8080/pizzas/toevoegen`.
2. Je doet onmiddellijk een pagina 'refresh' in de browser (bvb. met de toets F5).
3. De browser toont een waarschuwing. Je vraagt de request toch uit te voeren.
4. De browser voert de laatste request opnieuw uit. De website voegt de pizza nog eens toe: de laatste request was de POST request waarmee je een pizza toevoegde.
Dit had de gebruiker niet verwacht: hij zag een pagina met alle pizza's en dacht bij een 'refresh' alle pizza's opnieuw op te vragen.

Je kan de volgorde van requests ook zien met Firefox:

1. Je surft naar de pagina Toevoegen.
2. Je kiest rechts boven in Firefox .
3. Je kiest Web Developer.
4. Je kiest Network.
5. Je vult de naam en de prijs van een pizza in en je kiest Toevoegen.
6. Je ziet onder in het venster dat je met die keuze een POST request verstuurd.
De toevoegen method die deze request verwerkt, stuurt HTML naar de browser.
Deze HTML is aangemaakt door pizzas.jsp.

● 200 POST ☐ toevoegen

7. Je drukt F5 om de pagina te verversen en je kiest Resend bij de waarschuwing.
8. Je ziet onder in het venster dat je met die keuze dezelfde POST request terug verstuurd (en zo de pizza nog eens toevoegt).

● 200 POST ☐ toevoegen

Je lost dit probleem als volgt op:

1. Als je in de method toevoegen de request correct verwerkte, stuur je een redirect response naar de browser.
Zo'n response bevat de status code 302 (Found), een lege body en een Location header met een URL.
In ons voorbeeld is dit de URL waarop je alle pizza's ziet: /pizzas.
2. Wanneer de browser een redirect response ontvangt, doet de browser onmiddellijk een GET request naar de URL vermeld in de response header Location.
Bij ons is dit een GET request naar /pizzas die je verwerkt in de method pizzas.
Deze method leest alle pizza's uit de database en toont die aan de gebruiker. Als de gebruiker een 'refresh' doet, herhaalt de browser die GET request. De gebruiker ziet de pizza's opnieuw.



Je maakt een constante in PizzaController:

```
private static final String REDIRECT_URL_NA_TOEVOEGEN="redirect:/pizzas"; ❶
```

1. Als je een URL zal gebruiken als redirect URL, tik je voor die URL redirect:

Je gebruikt deze constante in de tweede method toevoegen.

Je wijzigt het laatste return statement:

```
return new ModelAndView(REDIRECT_URL_NA_TOEVOEGEN); ❶
```

(1) Wanneer je aan de ModelAndView constructor een String meegeeft die begint met redirect:/, gebruikt Spring deze String niet om een JSP te zoeken die HTML afbeeldt, maar gebruikt Spring deze String om een redirect response naar de browser te sturen.

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.

Je kan de volgorde van de requests ook zien met Firefox:

1. Je surft naar de pagina Toevoegen.
2. Je vult de naam en de prijs van een pizza in en je kiest Toevoegen.
3. Je ziet onder in het venster dat je met die keuze een POST request verstuurd:

▲ 302 POST ☐ toevoegen

De toevoegen method die deze request verwerkt, stuurt nu geen HTML naar de browser, maar een response met status code 302 en een response header Location.

Je ziet de response headers als je de regel 302 POST toevoegen aanklikt:

▼ Response headers (128 B)

- Content-Language: nl-BE
- Content-Length: 0
- Date: Mon, 20 Nov 2017 10:29:43 GMT
- Location: /pizzaluigi/pizzas

Je ziet dat de browser, bij de ontvangst van de response, direct een GET request doet naar de URL in die response header Location:

▲ 302 POST ☐ toevoegen

● 200 GET ☐ pizzas

De pizzas method stuurt, in samenwerking met pizzas.jsp HTML naar de browser.

4. Je drukt F5 om de pagina te verversen.
 5. Je ziet onder in het venster dat je met die keuze de GET request terug verstuurd.
- Deze GET request toont terug alle pizza's (en voegt geen pizza toe).

● 200 GET ☐ pizzas

34.1.1 Request parameters meegeven bij een redirect

Je leert hier hoe je request parameters kan toevoegen aan de URL waarnaar je redirect.

Je redirect in het voorbeeld naar /pizzas. Je voegt aan deze URL een parameter met de naam boodschap en met de tekst Pizza toegevoegd toe.

De URL wordt dan /pizzas?boodschap=Pizza+toegevoegd. Een URL stelt bepaalde tekens op een speciale manier voor. Een URL stelt een spatie bijvoorbeeld voor met +.

Je wijzigt de declaratie van de tweede method toevoegen:

```
ModelAndView toevoegen(@Valid Pizza pizza, BindingResult bindingResult,  
    RedirectAttributes redirectAttributes) ❶
```

(1) Je kan met een RedirectAttributes parameter verder in de code request parameters toevoegen aan de URL waarnaar je redirect.

Je voegt volgende regel toe voor de return opdracht

```
redirectAttributes.addAttribute("boodschap", "Pizza toegevoegd"); ❶
```

- (1) Spring zal aan de redirect URL een request parameter boodschap toevoegen met de inhoud Pizza toegevoegd. Spring zal daarbij de spatie vervangen door het plus teken.

Je voegt volgende regels toe aan pizzas.jsp, voor <h1>:

```
<c:if test='${not empty param.boodschap}'>  
  <div class='boodschap'>${param.boodschap}</div>  
</c:if>
```

❶

- (1) De expressie param.boodschap verwijst naar de request parameter boodschap.

Je kan de website uitproberen.

34.2 Dubbele submit vermijden

Als de gebruiker de submit knop twee keer snel na mekaar aanklikt, submit hij dezelfde HTML form twee keer. Hij zou dezelfde pizza twee keer toevoegen.

Je vermijdt dit door de submit knop te disablen bij de eerste submit.

Je tikt JavaScript code voor </body>:

```
<script>  
  document.getElementById('pizzaform').onsubmit = function() {  
    document.getElementById('toevoegknop').disabled = true;  
  };  
</script>
```



Je commit de sources. Je publiceert op GitHub.



Snack wijzgen: zie takenbundel

35 CROSS-SITE SCRIPTING (XSS)



Cross-site scripting is een hacker techniek.

De hacker tikt in een invoervak van een HTML form een stuk JavaScript, bijvoorbeeld: `<script>alert('hacked')</script>`.

Als je deze tekst daarna afbeeldt in een webpagina, voert de browser het script uit. In dit voorbeeld toont JavaScript een popup venster met de tekst hacked.

Je kan dit uitproberen door een pizza toe te voegen. Je tikt het volgende bij de naam:

```
<script>alert('hacked')</script>
```

Nadat je de pizza hebt toegevoegd, zie je een lijst met alle pizza's. Bij de nieuw toegevoegd pizza staat als naam letterlijk `<script>alert('hacked')</script>`. Hier is de hacker er niet in geslaagd een popup venster te tonen. Dit komt omdat je de naam toont met de tag `<c:out value='${pizza.naam}' />`. Deze tag helpt dus tegen cross-site scripting.

Als je echter Detail kiest bij de pizza, zie je een pagina waar de hacker er wel in slaagt het popup scherm twee keer te doen verschijnen. Dit komt omdat je in deze pagina de pizza naam twee keer gebruikt zonder daarbij de tag `<c:out ...>` toe te passen.

Een eerste oplossing tegen cross-site scripting is dus overal de tag `<c:out ...>` te gebruiken.

Dit vergt veel aandacht. Ook als andere applicaties de database lezen, moeten de programmeurs van die applicaties maatregelen treffen tegen cross-site scripting.

Een betere oplossing is de leuze “keep the garbage out of your database” te volgen.

Hierbij valideer je het invoervak: het mag geen `<script>` ... bevatten.

Deze validatie gebruikt intern de jsoup library.

Je voegt daartoe een dependency toe aan pom.xml:

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

Je doet de validatie (die cross-site scripting verhindert) met `@SafeHtml`.

Je tikt in de class Pizza `@SafeHtml` voor de private variabele naam.

Je voegt de foutmelding die bij `@SafeHtml` hoort toe aan `ValidationMessages.properties`:
`org.hibernate.validator.constraints.SafeHtml.message=mag geen script bevatten`

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



Opmerking: je tikt `@SafeHtml` niet voor numerieke en datum variabelen. Deze kunnen geen letters bevatten. Ze zijn daarom niet gevoelig voor een XSS hack.

36 SESSION SCOPED BEANS

Je hebt reeds gezien dat HTTP een stateless protocol is: elke request is een zelfstandige handeling. Zijn verwerking hangt niet af van de verwerking van vorige requests. Variabelen die je aanmaakt gedurende het verwerken van een request, verdwijnen uit het interne geheugen van de webserver nadat de request verwerkt is. Ze zijn niet meer beschikbaar bij een volgende request.

36.1 Session

Sommige data moet je toch onthouden over requests heen, zoals gebruikersnaam of winkelmandje



Één controller instance verwerkt de requests van *alle* gebruikers.

Je onthoudt een winkelmandje niet zomaar in een *private* variabele van die controller. Alle gebruikers zouden hetzelfde winkelmandje delen!

Je gebruikt voor zulke data een session: een stuk webserver geheugen waarin je data voor één gebruiker onthoudt. Het is aan te raden de hoeveelheid data in session te minimaliseren:

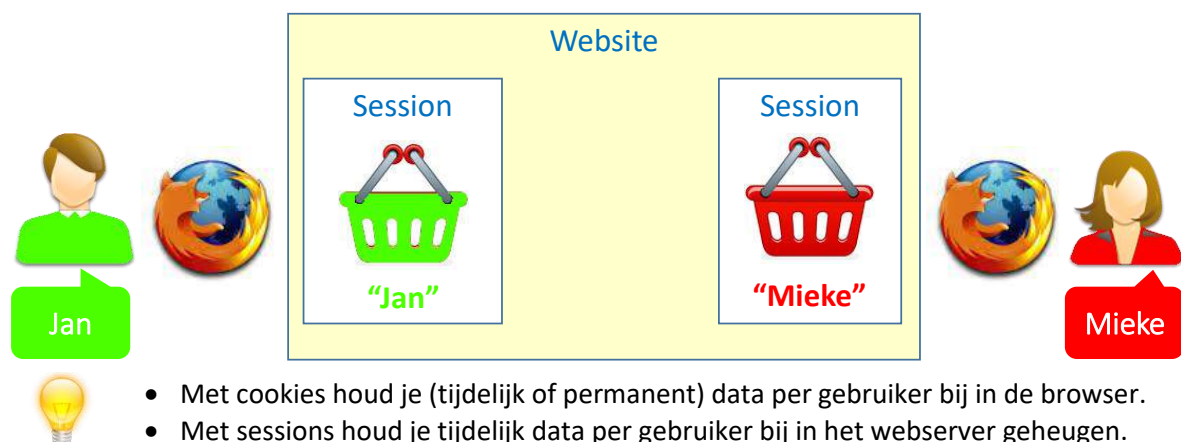
- Bij veel gelijktijdige gebruikers bevat het interne geheugen van de webserver veel sessions. Als elk van die sessions weinig data bevat, kom je geen geheugen te kort.
- Als de website draait op meerdere webserver in een webfarm, moeten die webserver de sessions niet enkel onthouden in hun eigen interne geheugen. Ze moeten de sessions ook uitwisselen met de andere webserver wanneer de data in de sessions wijzigt. Dit uitwisselen heet *session replication*. Dit uitwisselen gebeurt over het interne netwerk tussen de webserver. Dit uitwisselen is essentieel zodat alle webserver dezelfde data per gebruiker bevatten. Als één van de webserver uitvalt, kennen de overgebleven webserver nog de data per gebruiker. Als de sessions weinig data bevat, gebeurt dit uitwisselen performant.

Voorbeelden van weinig data in session:

- Als de gebruiker per artikel maar één stuk kan bestellen, houd je in het winkelmandje artikelnummers bij, niet volledige artikel objecten.
- Als de gebruiker per artikel meerdere stuks kan bestellen, houd je in het winkelmandje artikelnummers en -aantallen bij te houden.

Één session kan meerdere data bevatten. Elk data heet een session attribuut.

Een session kan bijvoorbeeld twee attributen bevatten: gebruikersnaam en winkelmandje.



36.2 Serializable

Als de waarde in een session attriboot een object is (geen primitief data type), moet de bijbehorende class `Serializable` implementeren. De redenen hiervoor zijn:

- Session persistence
- Session replication

36.2.1 Session persistence

De webserver onthoud sessions in zijn RAM geheugen. Om deze sessions niet te verliezen bij een herstart van de webserver, doet de webserver volgende stappen bij een herstart:

1. De webserver schrijft de sessions via serialization naar een tijdelijk bestand. Dit serialization proces werkt enkel als de objecten `Serializable` zijn.
2. De webserver herstart.
3. De webserver leest de sessions terug uit het tijdelijk bestand.

36.2.2 Session replication

In een web cluster(webfam) stuurt een webserver elke session wijziging via het netwerk naar de andere webserver uit de cluster. Dit heet session replication.

Alle webserver moeten dezelfde sessions bevatten: webserver A verwerkt een request, maar een andere webserver B verwerkt een volgende request van dezelfde gebruiker:

- Mieke legt in haar mandje appels.
- Webserver A verwerkt die request en onthoudt het mandje als een session attriboot.
- Webserver A stuurt die session (inclusief de session ID) naar webserver B.
- Webserver B onthoudt die session in zijn geheugen.
- Mieke vraagt haar mandje te zien.
- Webserver B verwerkt die request: hij haalt het mandje uit de session van Mieke.

Je kan objecten enkel over het netwerk versturen als ze `Serializable` zijn.

36.3 Session identificatie

Wanneer een gebruiker een eerste request doet naar een website, maakt de webserver voor hem een session met een unieke identificatie: de session ID.

De session van Jan krijgt bijvoorbeeld een session ID `0AAB9C8DE666`.

De session van Mieke krijgt bijvoorbeeld een session ID `DD12EE78A4B5`.

Het interne geheugen van de webserver bevat dus één session per actieve gebruiker.

De webserver identificeert elke session met een uniek session ID.

De webserver stuurt deze sessionID in de response naar de browser.

- Als tijdelijke cookies ingeschakeld zijn in de browser, stuurt de webserver de session ID als een tijdelijke cookie. Die heeft de naam `JSESSIONID`. Bij elke volgende request leest de de webserver deze tijdelijke cookie. Aan de hand van die waarde kan de webserver de sessie ophalen die bij de huidige gebruiker hoort. Als de cookie de waarde `0AAB9C8DE666` heeft haalt de webserver de session van Jan op uit zijn RAM geheugen. De webserver biedt je code die session aan zodat je hem kan lezen of wijzigen. Als de cookie de waarde `DD12EE78A4B5` heeft haalt de webserver de session van Mieke op uit zijn RAM geheugen.
- Als tijdelijke cookies uitgeschakeld zijn in de browser, voegt de webserver aan elke URL die hij in de HTML van de response opneemt de session ID toe. De URL `/pizzas` wordt bij Jan `/pizzas;jsessionid=0AAB9C8DE666`. Zo'n URL kan een onderdeel zijn van een hyperlink tag of van een form tag. De webserver voegt de session ID toe als je de URL aanmaakt met de tag `<c:url ...>` of `<spring:url>`. Als de gebruiker een hyperlink aanklikt of een form submit, verwerkt de webserver de bijbehorende request. De webserver leest de session ID die onderdeel is van de request URL. Als de request URL bijvoorbeeld `/pizzas;jsessionid=0AAB9C8DE666` is, weet de webserver dat de session ID van de gebruiker `0AAB9C8DE666` is en haalt de session van Jan op uit zijn RAM geheugen. De webserver biedt je code die session aan zodat je hem kan lezen of wijzigen.

Direct na het aanmaken van een session weet de webserver niet of in de browser tijdelijke cookies ingeschakeld zijn. Hij gebruikt daarom in de eerste response beide strategieën: een tijdelijke cookie én URL rewriting. Als de volgende request een cookie JSESSIONID bevat, zijn cookies ingeschakeld, en past de webserver geen URL rewriting meer toe.



Belangrijk: de webserver stuurt enkel de session ID naar de browser, nooit de session data. Session data mag dus confidentiële informatie bevatten.

36.4 Webserver verwijdt een session

Als een session gedurende een aantal minuten niet aangesproken werd, verwijdt de webserver die session uit zijn geheugen. Men zegt dat de session 'vervalt'. Dit gebeurt bijvoorbeeld omdat de gebruiker de browser sluit of naar een andere website gaat.

Elk webserver merk heeft een eigen waarde voor het aantal minuten waarna een session verval. Bij Tomcat is dit 30 minuten.

36.5 Voorbeeld 1

De gebruiker tikt zijn email adres. Je onthoudt dit email adres in de session van de gebruiker. Zo lang de gebruiker op de website blijft, kan je zijn email adres terug lezen uit de session.

36.5.1 Session data als Spring bean

Je beschrijft de methods, die je wil uitvoeren op de data die je onthoudt in de session van de gebruiker, in een interface:

```
package be.vdab.pizzaluigi.web;
interface Identificatie {
    String getEmailAdres();
    void setEmailAdres(String adres);
}
```

❶

- (1) Je hebt de interface enkel in de huidige package `be.vdab.pizzaluigi.web` nodig, dus hoeft hij niet public te zijn.

Je implementeert deze interface in een class:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Component
@SessionScope
class DefaultIdentificatie implements Serializable, Identificatie {
    private static final long serialVersionUID = 1L;
    @Email
    private String emailAdres;
    @Override
    public String getEmailAdres() {
        return emailAdres;
    }
    @Override
    public void setEmailAdres(String adres) {
        this.emailAdres = adres;
    }
}
```

❶

❷

❸

- (2) De data die je wil onthouden in de session moet een Spring bean zijn.
- (3) Standaard maakt Spring één bean bij het starten van de applicatie. Alle gebruikers delen die ene bean. Dit is niet wat je van session data verwacht. Met `@SessionScope` maakt Spring een bean per gebruiker die de website bezoekt. Spring houdt die bean bij in de session van die gebruiker.
- (4) De data in de session moet serializable zijn.

De gebruiker moet zijn email adres kunnen intikken. Je maakt daartoe een controller:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Controller
@RequestMapping("identificatie")
class IdentificatieController {
    private static final String VIEW = "identificatie";
    private final Identificatie identificatie;
    IdentificatieController(Identificatie identificatie) {
        this.identificatie = identificatie;
    }
    @GetMapping
    ModelAndView identificatie() {
        return new ModelAndView(VIEW, "identificatie", identificatie);
    }
    private static final String REDIRECT_NA_SUBMIT="redirect:/";
    @PostMapping
    String identificatie(@Valid DefaultIdentificatie identificatie,
        BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
            return VIEW;
        }
        this.identificatie.setEmailAdres(identificatie.getEmailAdres());
        return REDIRECT_NA_SUBMIT;
    }
}
```

- (1) Je injecteert de Spring bean die toegang geeft tot de session data met het email adres van de huidige gebruiker.
- (2) Je geeft het object met het email adres door aan de JSP. Je gebruikt het daar als form object. Er zijn twee mogelijkheden. Eerste mogelijkheid: de gebruiker surft voor de eerste keer naar de pagina. De webserver heeft dan voor die gebruiker een nieuwe session gemaakt. Deze session bevat een nieuw aangemaakt DefaultIdentificatie object. Tweede mogelijkheid: de gebruiker surft voor een volgende keer naar de pagina. De webserver heeft dan de session van de huidige gebruiker opgehaald uit zijn RAM geheugen. Deze session bevat het DefaultIdentificatie object van de vorige requests van die gebruiker. Je kan dus uit dit object het email adres lezen dat de gebruiker bij vorige requests intikte.
- (3) De method bij (4) zal het submitten van de form verwerken. Hierbij *wijzigt* data in de session van de gebruiker. Je submit daarom de form met method="post" in plaats van met method="get".
- (4) Deze method verwerkt de request nadat de gebruiker de form submit. Spring heeft een nieuw DefaultIdentificatie object aangemaakt en opgevuld met het adres dat de gebruiker intikte. Je valideert dit object.
- (5) Je brengt het ingetikte adres over naar het DefaultIdentificatie object in de session van de gebruiker. Zo is het ter beschikking bij volgende requests van de gebruiker.

Je maakt identificatie.jsp:

```
<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
<%@taglib prefix='form' uri='http://www.springframework.org/tags/form'%>
<!doctype html>
<html lang='nl'>
<head>
    <c:import url='/WEB-INF/JSP/head.jsp'>
        <c:param name='title' value='Identificatie'/>
    </c:import>
</head>
```

```

<body>
  <c:import url='/WEB-INF/JSP/menu.jsp'/>
  <h1>Identificatie</h1>
  <c:url value='/identificatie' var='url'/>
  <form:form action='${url}' modelAttribute='identificatie' method='post'
    id='identificatieForm'>
    <form:label path='emailAdres'>Email adres:
      <form:errors path='emailAdres' /></form:label>
    <form:input path='emailAdres' autofocus='autofocus' required='required'
      type='email' />
    <input type='submit' value='OK' id='okKnop' />
  </form:form>
  <script>
    document.getElementById('identificatieForm').onsubmit = function() {
      document.getElementById('okKnop').disabled = true;
    };
  </script>
</body>
</html>

```

Je kan de website uitproberen. Nadat je je geïdentificeerd hebt kan je andere pagina's van de website bezoeken. Als je terugkeert naar de identificatiepagina zie je dat je email adres onthouden is (in de session van de gebruiker).

Je kan meerdere gebruikers van de website simuleren door de website te openen in meerdere browsers van verschillende merken (bijvoorbeeld eenmaal in Firefox en eenmaal in Chrome). Je kan je op de ene browser identificeren met een email adres en op de andere browser met een ander email adres.

Je toont ook op de welkompagina de indentificatie van de gebruiker.

Je maakt een private variabele die verwijst naar de data in de session van de gebruiker:

```
private final Identificatie identificatie;
```

Je maakt een constructor met dependency injection:

```
IndexController(Identificatie identificatie) {
  this.identificatie = identificatie;
}
```

Je voegt in de method index volgende opdracht toe voor het return statement:

```
modelAndView.addObject("identificatie", identificatie);
```

Je voegt in index.jsp volgende regel toe voor </body>:

```
<p>${identificatie.emailAdres}</p>
```

Je kan de website opnieuw uitproberen.



Je commit de sources . Je publiceert op GitHub.

36.6 Voorbeeld 2

Je maakt een pagina met een mandje waar de gebruiker pizza's kan aan toevoegen.

Je beschrijft de methods, die je wil uitvoeren op de data die je onthoudt in de session van de gebruiker, in een interface:

```

package be.vdab.pizzaluigi.web;
import java.util.List;
interface Mandje {
  void addPizzaId(long pizzaId);
  public List<Long> getPizzaIds();
}

```

Je implementeert deze interface in een class:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Component
@SessionScope
class DefaultMandje implements Serializable, Mandje {
    private static final long serialVersionUID = 1L;
    private final List<Long> pizzaIds = new ArrayList<>();
    @Override
    public void addPizzaId(long pizzaId) {
        pizzaIds.add(pizzaId);
    }
    @Override
    public List<Long> getPizzaIds() {
        return pizzaIds;
    }
}
```

❶

- (1) Je onthoudt zo weinig mogelijk data in de session van de gebruiker.
Je onthoudt daarom geen pizzas's, enkel pizza id's.

Je maakt een class die zal dienen als basis van het form object:

```
package be.vdab.pizzaluigi.web;
class MandjeForm {
    private long pizzaId;
    // je maakt een getter en een setter voor pizzaId
}
```

Je maakt een controller:

```
package be.vdab.pizzaluigi.web;
// enkele imports
@Controller
@RequestMapping("mandje")
class MandjeController {
    private final Mandje mandje;
    private final PizzaService pizzaService;
    MandjeController(Mandje mandje, PizzaService pizzaService) {
        this.mandje = mandje;
        this.pizzaService = pizzaService;
    }
    private List<Pizza> maakPizzasVanPizzaIds(List<Long> pizzaIds) {
        List<Pizza> pizzas = new ArrayList<>(pizzaIds.size());
        for (long id : pizzaIds) {
            pizzaService.read(id).ifPresent(pizza -> pizzas.add(pizza));
        }
        return pizzas;
    }
    private static final String VIEW = "mandje";
    @GetMapping
    ModelAndView toonMandje() {
        return new ModelAndView(VIEW)
            .addObject(new MandjeForm())
            .addObject("allePizzas", pizzaService.findAll())
            .addObject("pizzasInMandje",
                maakPizzasVanPizzaIds(mandje.getPizzaIds()));
    }
}
```

❶

❷

❸

❹

```

private static final String REDIRECT_NA_TOEVOEGEN = "redirect:/mandje";
@PostMapping
String voegPizzaToeAanMandje(MandjeForm form) {
    mandje.addPizzaId(form.getPizzaId());
    return REDIRECT_NA_TOEVOEGEN;
}
}

```

- (1) Deze method krijgt een verzameling pizza id's als parameter binnen. Dit zullen de pizza id's zijn die je onthoudt in het mandje in de session van de gebruiker. De method geeft de verzameling Pizza objecten terug die bij het mandje horen. Je zal deze verzameling doorgeven aan de JSP, die hiermee de pizza's in het mandje zal afbeelden.
- (2) Je maakt een form object. Je geeft het door aan de JSP.
- (3) Om een pizza toe te voegen zal de gebruiker een pizza kiezen uit de lijst met alle pizza's. Je geeft deze lijst door aan de JSP.
- (4) De JSP toont ook de pizza's die momenteel in het mandje liggen. Je geeft deze pizza's door aan de JSP.

Je maakt mandje.jsp:

```

<%@page contentType='text/html' pageEncoding='UTF-8' session='false'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<%@taglib prefix='form' uri='http://www.springframework.org/tags/form'%>
<!doctype html>
<html lang='nl'>
<head>
    <c:import url='/WEB-INF/JSP/head.jsp'>
        <c:param name='title' value='Mandje'/>
    </c:import>
</head>
<body>
<c:import url='/WEB-INF/JSP/menu.jsp' />
    <h1>Mandje</h1>
    <c:url value='/mandje' var='url' />
    <form:form action='${url}' modelAttribute='mandjeForm' method='post'
        id='mandjeform'>
        <form:label path='pizzaId'>Pizza: <form:errors path='pizzaId' /></form:label>
        <form:select path='pizzaId' items='${allePizzas}' itemLabel='naam'
            itemValue='id' />
        <input type='submit' value='Toevoegen' id='toevoegknop'>
    </form:form>
    <c:if test='${not empty pizzasInMandje}'>
        <h2>Pizza's in mandje</h2>
        <ul>
            <c:forEach items='${pizzasInMandje}' var='pizza'>
                <li><c:out value='${pizza.naam}' /></li>
            </c:forEach>
        </ul>
    </c:if>
    <script>
        document.getElementById('mandjeform').onsubmit = function() {
            document.getElementById('toevoegknop').disabled = true;
        };
    </script>
</body>
</html>

```

- (1) Je maakt een keuzelijst met alle pizza's waaruit de gebruiker er één kan selecteren. Je vermeldt bij `items` de verzameling waarmee Spring de keuzelijst vult. Je vermeldt bij `itemLabel` de eigenschap die Spring van ieder item in de keuzelijst toont. Je vermeldt bij `itemValue` de eigenschap van het gekozen item die de browser doorstuurt als de gebruiker de form submit. Als de gebruiker bijvoorbeeld pizza 7 kiest en de form submit, zal de browser in de request het getal 7 naar de website sturen.

De HTML die deze tag genereert ziet er als volgt uit:

```
<select id="pizzaId" name="pizzaId">
  <option value="1">Prosciutto</option>
  <option value="2">Margheritta</option>
  <option value="3">Calzone</option>
  <option value="4">Fungi & Olive</option>
</select>
```

Je kan de website opnieuw uitproberen. Je kan pizza's in het mandje leggen. Je kan daarna andere pagina's van de website bezoeken. Als je terugkeert naar het mandje, zie je nog altijd de pizza's die je er in had gelegd. Dit komt omdat je de pizza id's onthoudt in de session van de gebruiker.



Je commit de sources. Je publiceert op GitHub.

36.7 Session fixation



Session fixation is een hacker techniek. De hacker kan die toepassen als de session id is opgenomen in de URL van de requests. De hacker stuurt een request naar de website. Hij ziet in de response een URL met zijn session id. Hij stuurt daarna een mail naar een gebruiker. Deze mail bevat een hyperlink naar de website, met de session id van de hacker in de URL. Als de gebruiker deze hyperlink volgt, voert hij een handeling uit op de website alsof hij de hacker is. Deze handeling kan bijvoorbeeld het overschrijven van geld zijn op de rekening van de hacker.

Je voegt volgende regel toe aan `application.properties`, om session fixation te voorkomen:

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



- (1) Je definieert dat de Spring de session id enkel naar de browser stuurt als een tijdelijke cookie, nooit in de URL.



Zoek de friet: zie takenbundel



Sauzen raden: zie takenbundel

37 GETALOPMAAK, DATUMOPMAAK, TIJDOPMAAK


37.1 Getalopmaak

Je doet de opmaak van getallen gebaseerd op het land van de gebruiker. Voorbeelden:

- Je toont getallen aan een Belgische gebruiker met een komma tussen eenheden en decimalen, en een punt tussen duizendtallen (1.000,23).
- Je toont getallen aan een gebruiker uit de USA met een punt tussen eenheden en decimalen, en een komma tussen duizendtallen (1,000.23).

Spring helpt je bij deze opmaak. Spring baseert het land van de gebruiker op de request header `Accept-Language`. De browser stuurt die met iedere request mee. Deze header bevat de taal en het land van de gebruiker. Als deze header bijvoorbeeld `n1-be` bevat, betekent dit dat de gebruiker Nederlands spreekt en in België woont. Deze header kan ook meerdere taal-land combinaties bevatten. De meest geprefereerde combinatie is als eerste vermeld. Voorbeeld: `n1-be, en-us`.

De gebruiker bepaalt in de browser instellingen de inhoud van de header. Bij 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 kan taal-land combinaties toevoegen met Select a language to add en Add.
5. Je kan de volgorde instellen met Move Up en Move Down.

Je definieert de opmaak van een numerieke variabele (`int`, `double`, `BigDecimal`, ...) met `@NumberFormat` voor die variabele. Je geeft een parameter `style` of een parameter `pattern` mee waarmee je de getalopmaak definieert.

`style` heeft als type de enum `Style`, met volgende mogelijkheden:

style	Betekenis	Getal	Opgemaakt (Europees formaat)
NUMBER	puntjes tussen 1000 tallen	2100,12	2.100,12
CURRENCY	zelfde als NUMBER, mét muntcode	2100,12	2.100,12 €
PERCENT	In percentagevorm	0.37	37%

Voorbeeld: `@NumberFormat(style = Style.NUMBER)`

`style` is soms beperkt. Je kan bvb. het aantal cijfers na de komma niet definiëren.

Je kan bijvoorbeeld `50` niet tonen als `50.00`, `style` toont `50` altijd als `50`.

Als je dit wel wil, vervang je `style` door `pattern`. Je geeft dan een opmaakpatroon mee.

Je kan in dit patroon volgende tekens gebruiken:

- 0 cijfer altijd tonen
- # cijfer enkel tonen als dat nodig is
- , scheidingsteken tussen 1000 tallen
- . scheidingsteken tussen eenheden en decimalen

Voorbeelden:

pattern	Getal	Opgemaakt (Europees formaat)
<code>#,##0.00</code>	1000	1.000,00
<code>#,##0.00</code>	1000000	1.000.000,00
<code>#,##0.00</code>	50	50,00

Je tikt in de class `Pizza` vóór `private BigDecimal prijs`
`@NumberFormat(pattern = "0.00")`

37.2 spring:eval

Spring bevat een JSP tag eval.

Je verwijst in het attribuut expression naar een attribuut voorzien van `@NumberFormat`.

De tag toont dit attribuut, mét de opmaak in de bijbehorende `@NumberFormat`.

Je gebruikt dit als voorbeeld in `pizza.jsp`:

1. Je importeert de tag library die de eval tag bevat.

```
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
```
2. Je vervangt `${pizza.prijs}` door

```
<spring:eval expression='pizza.prijs'/>
```

 Bemerk dat je de expressie (`pizza.prijs`) niet omringt met `${` en `}`.

Je kan de website uitproberen. Je ziet de prijs met een komma tussen eenheden en decimalen als je als land België gekozen hebt in Firefox.

Je ziet de prijs in dollar verkeerd: met een punt tussen eenheden en decimalen.

Je zal nu ook deze prijs correct tonen. Je hebt hierbij wel een uitdaging: je stuurt deze prijs rechtstreeks als `BigDecimal` van de `PizzaController` naar de JSP:

```
modelAndView.addObject("inDollar", euroService.naarDollar(pizza.getPrijs()));
```

Deze prijs is geen private variabele (binnen een ander object) waarvoor je `@NumberFormat` kan tikken. Je maakt een class om dit op te lossen:

```
package be.vdab.pizzaluigi.valueobjects;
// enkele imports
public class Dollar {
    @NumberFormat(pattern = "0.00")
    private final BigDecimal waarde;
    public Dollar(BigDecimal waarde) {
        this.waarde = waarde;
    }
    public BigDecimal getWaarde() {
        return waarde;
    }
}
```

Je wijzigt in `PizzaController` de regel

```
modelAndView.addObject("inDollar", euroService.naarDollar(pizza.getPrijs()));
```

naar

```
modelAndView.addObject("inDollar",
    new Dollar(euroService.naarDollar(pizza.getPrijs())));
```

Je vervangt in `pizza.jsp` `${inDollar}` door

```
<spring:eval expression='inDollar.waarde'/>
```

Je kan de website terug uitproberen.

37.3 Datumopmaak, tijdopmaak

Je doet de opmaak van datums gebaseerd op het land van de gebruiker. Voorbeelden:

- Je toont datums aan Belgische gebruikers als dag/maand/jaar (31/1/2017).
- Je toont datums aan Amerikaanse gebruikers als maand/dag/jaar (1/31/2017)

Je definieert de opmaak van een `Date`, `LocalDate`, `LocalTime` of `LocalDateTime` variabele met `@DateTimeFormat` voor die variabele. `@DateTimeFormat` heeft een parameter `style`.

Deze definieert de opmaak van het datumdeel en/of het tijddeel.

Het eerste teken in `style` definieert de opmaak van het datumdeel:

Teken	Betekenis	Toegepast op 1/4/2010 (Europees formaat)
S	Short style	1/04/10
M	Medium style	1-apr-2010
L	Long style	1 april 2010
F	Full style	donderdag 1 april 2010
-	Datumdeel niet tonen	

Het tweede teken in `style` definieert de opmaak van het tijdsdeel:

Teken	Betekenis	Toegepast op 14:28:56 (Europees formaat)
S	Short style	14:28
M	Medium style	14:28:56
L	Long style	14:28:56 CET (CET betekent Central European Time)
F	Full style	14:28 u. CET
-	Tijdsdeel niet tonen	

Voorbeeld: je toont het datumdeel in short style en je toont het tijdsdeel niet.

```
@DateTimeFormat(style = "S-")
```

Je zal opmaak doen op de datum-tijd laatst bezocht op de welkompagina. Ook hier heb je de uitdaging dat je deze waarde in `IndexController` rechtstreeks naar de JSP stuurt:

```
modelAndView.addObject("laatstBezocht", laatstBezocht);
```

Deze datumtijd is geen private variabele (binnen een ander object) waarvoor je `@DateTimeFormat` kan tikken. Je maakt een class om dit op te lossen:

```
package be.vdab.pizzaluigi.valueobjects;
// enkele imports
public class DatumTijd {
    @DateTimeFormat(style = "SS")
    private final LocalDateTime waarde;
    public DatumTijd(LocalDateTime waarde) {
        this.waarde = waarde;
    }
    public LocalDateTime getWaarde() {
        return waarde;
    }
}
```

Je wijzigt in `IndexController` de regel

```
modelAndView.addObject("laatstBezocht", laatstBezocht);
```

naar

```
modelAndView.addObject("laatstBezocht",
    new DatumTijd(LocalDateTime.parse(laatstBezocht)));
```

❶

(1) De method `parse` verwacht een `String` met een datum-tijd als parameter.

De method geeft een `LocalDateTime` terug met dezelfde datum-tijd waarde.

Je importeert in `index.jsp` de tag library die de `eval` tag bevat:

```
<%@taglib prefix='spring' uri='http://www.springframework.org/tags'%>
```

Je vervangt in `${laatstBezocht}` door

```
<spring:eval expression='laatstBezocht.waarde' />
```

Je kan de website opnieuw uitproberen.



Je commit de sources. Je publiceert op GitHub.

38 CUSTOM TAGS

Je gebruikt in je JSP's tags zoals `<c:forEach>`. Je maakt in dit hoofdstuk eigen (custom) tags. Een custom tag kan hetzelfde als de JSTL tag `import`, maar vereist minder tikwerk bij het oproepen en heeft meer mogelijkheden naar parameters toe.

38.1 TLD bestand

Je verzamelt custom tags, die een samenhangend geheel vormen, in een tag library.

Je maakt per tag library één TLD (Tag Library Descriptor) bestand.

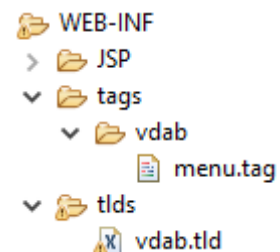
Dit XML bestand bevat volgende informatie:

- De naam en de locatie van elke tag in de tag library.
- De URI waarmee je de tag library associeert
De conventie is dat de URI van je firma een onderdeel is van de URI.
Op die manier heeft elke tag library op de wereld een unieke URI.
- De voorkeur prefix (bvb `vdab`) die je gebruikt als je in een JSP naar de tag library verwijst.

Je plaatst een TLD in de folder `WEB-INF`, of in een subfolder van `WEB-INF`.

Je maakt een custom tag:

1. Je maakt in `WEB-INF` een folder `tags`.
2. Je maakt in `tags` een folder `vdab`.
3. Je klikt met de rechtermuisknop op `vdab`.
4. Je kiest `New, Other, Web, JSP Tag` en `Next`.
5. Je tikt menu bij `File Name` en je kiest `Finish`.



Je maakt een TLD:

1. Je maakt in `WEB-INF` een folder `tlds`.
2. Je klikt met de rechtermuisknop op `tlds`.
3. Je kiest `New, Other, XML, XML File` en `Next`.
4. Je tikt `vdab.tld` bij `File Name` en je kiest `Finish`.

38.2 vdab.tld

```
<?xml version='1.0' encoding='UTF-8'?>
<taglib version='2.1' xmlns='http://java.sun.com/xml/ns/javaee'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation='http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd'>
  <tlib-version>1.0</tlib-version>
  <short-name>vdab</short-name>
  <uri>http://vdab.be/tags</uri>
  <tag-file>
    <name>menu</name>
    <path>/WEB-INF/tags/vdab/menu.tag</path>
  </tag-file>
</taglib>
```

①
②
③
④
⑤
⑥

- (1) Je definieert met `tlib-version` een versienummer van de tag library.
- (2) Je definieert met `short-name` de voorkeur prefix voor de tag library.
- (3) Je definieert met `uri` de URI van de tag library.
- (4) Je definieert per tag die behoort tot de tag library een element `tag-file`.
- (5) Je definieert met `name` de tag naam.
- (6) Je definieert met `path` de plaats en de naam van het tag bestand.

38.3 menu.tag

Je wijzigt `menu.tag`:

```
<%@tag description='menu' pageEncoding='UTF-8'%>
```

①

- (1) Een custom tag bevat een page directive tag. `description` bevat een optionele omschrijving van wat de tag doet. De IDE toont die omschrijving als je de tag gebruikt in een JSP.

Je kopieert onder deze regel alle regels uit `menu.jsp`, behalve de eerste regel.

Je verwijdert `menu.jsp`.

38.4 Custom tag gebruiken in JSP

Je tikt in elke JSP, behalve `head.jsp`, onder `<%@page ... %>`

```
<%@taglib uri='http://vdab.be/tags' prefix='vdab'%>
```

Je vervangt in elke JSP, behalve `head.jsp`, `<c:import url='/WEB-INF/JSP/menu.jsp'/>` door `<vdab:menu/>`

Je kan de website uitproberen.

38.5 Custom tag attributen

Je kan een custom tag voorzien van één of meerdere attributen (parameters).

Als je de custom tag oproept in een JSP, geef je waarden mee voor de attributen.

Je kan per attribuut volgende eigenschappen definiëren

- `name`. Deze eigenschap is verplicht.
- `description`. Deze eigenschap is optioneel.
- `required`. Is het verplicht dit attribuut mee te geven als je de tag oproept in een JSP? Default is een attribuut optioneel.
- `type`. Het datatype van het attribuut. Deze eigenschap is optioneel. Als het attribuut een waarde van een verkeerd type binnenkrijgt, werpt Java een Exception bij het uitvoeren van de website.

Je maakt een custom tag `head` ter vervanging van `head.jsp`.

Je maakt `head.tag`:

```
<%@tag description='head onderdeel van pagina' pageEncoding='UTF-8'%>
<%@attribute name='title' required='true' type='java.lang.String'%>
<%@taglib prefix='c' uri='http://java.sun.com/jsp/jstl/core'%>
<title>${title}</title>
<link rel='icon' href='<c:url value="/images/pizza.ico"/>' type='image/x-icon'>
<meta name='viewport' content='width=device-width,initial-scale=1'>
<link rel='stylesheet' href='<c:url value="/css/pizzaluigi.css"/>'>
```

①

(1) Je definieert één attribuut met een page directive attribute.

Je tikt bij `name` de attribuut naam.

Je geeft bij `required` aan dat dit attribuut verplicht in te vullen is.

Je tikt bij `type` het attribuut type.

Je registreert de tag in `vdab.tld`. Je tikt onder `</tag-file>`:

```
<tag-file>
  <name>head</name>
  <path>
    /WEB-INF/tags/vdab/head.tag
  </path>
</tag-file>
```

Je verwijdert `head.jsp`.

Je vervangt in elke JSP `<c:import url='/WEB-INF/JSP/head.jsp'>...</c:import>` door `<vdab:head title='Tik hier wat bij value van c:param stond'/>`

Je kan de website uitproberen.



Je commit de sources. Je publiceert op GitHub.



Custom tags: zie takenbundel

39 STAPPENPLAN

Je kan dit stappenplan volgen als je een web applicatie maakt met Spring:

1. Je voegt regels toe aan `application.properties`:

```
spring.mvc.view.prefix:/WEB-INF/JSP/
spring.mvc.view.suffix:.jsp
spring.datasource.url=xxx
spring.datasource.username=yyy
spring.datasource.password=zzz
```

Je schrijft een `DataSourceTest`.

Je maakt alle classes die personen en/of dingen uit de werkelijkheid voorstellen.

Je plaatst een berekening die met zo'n class te maken heeft heeft in de class zelf, niet in JSP's, 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 getPrijsInclusief() {
        return prijsExclusief.multiply(BigDecimal.ONE
            .add(btwPercentage.divide(BigDecimal.valueOf(100), 2,
                RoundingMode.HALF_UP)));
    }
}
```

Eerste voordeel: je kan de berekening 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 zou zijn opgenomen).

Je test de correcte werking van zo'n method met een unit test.

Je definieert de opmaak van `BigDecimal` en `Date` private variabelen met `@NumberFormat` en `@DateTimeFormat`.

Je voegt bean validation annotations toe als de gebruiker objecten van de class toevoegt of wijzigt.

2. Je doet volgende stappen per nieuwe pagina van de web applicatie:
 - a. Je voegt de nodige repository interfaces, implementatie classes en methods toe.
Je injecteert de `JdbcTemplate` in de constructor.
Je schrijft integration tests voor de repositories.
 - b. Je voegt de nodige service interfaces, implementatie classes en methods toe.
Je tikt `@Service` en `@Transactional` bij deze classes.
Je injecteert de nodige repositories in de constructor.
 - c. Je voegt de nodige controller class en/of methods toe. Je tikt `@Controller` en `@RequestMapping` bij deze class. Je injecteert de nodige services in de constructor.
Als je session data nodig hebt, onthoud je daarin geen entities, maar identifiers van die entities.
 - d. Je voegt een JSP toe.
 - e. Je test de pagina in de browser.

40 HERHALINGSOEFENINGEN



Gastenboek: zie takenbundel



Gastenboekbeheer: zie takenbundel

41 COLOFON

Domeinexpertisemanager:	Jean Smits
Moduleverantwoordelijke:	Hans Desmet
Medewerkers:	Hans Desmet
Versie:	1/10/2018
Nummer dotatielijst:	