FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Labo frameworks voor serverapplicaties
Reeks 1: REST, Spring

28 september 2020

1 Inleiding

In Gebruikersinterfaces maakten jullie onder andere kennis met HTML5 en Javascript. De combinatie van beiden laat toe om complexe **client-side webapplicaties** te maken. In dit vak beginnen we met **server-side programmeren**. Er bestaan verschillende technologieën (binnen verschillende programmeertalen) om dit te doen maar in Java zijn **Servlets** nog altijd de **basiscomponenten** waarop verschillende frameworks gebouwd zijn.

Servlets laten toe om code aan de serverkant uit te voeren. Een servlet genereert dan uitvoer die een webbrowser rechtstreeks kan weergeven (HTML-uitvoer) of die eerst nog verwerkt moet worden (JSON, XML, ... uitvoer). Een servlet is niets meer dan een gewone Java-klasse (die overerft van een Servlet-klasse, bv. HttpServlet). In deze klasse wordt er dan een methode overschreven die een (HTTP-)request kan afhandelen. De servlets draaien binnen een webcontainer die verantwoordelijk is voor het aanmaken van de servlet-objecten en voor het toewijzen van verzoeken aan de servlet-objecten. In dit labo gebruiken we **Apache Tomcat** als webcontainer.

Servlets bieden dus een vrij low level manier aan om webapplicaties te bouwen. Manueel een HTML-pagina opbouwen binnen een servlet is een achterhaalde manier van werken. Tegenwoordig zijn er betere oplossingen zoals het **Spring-framework** in Java. Dit framework gebruikt wel nog steeds servlets achter de schermen, maar biedt een productievere omgeving aan de developer.

Gedurende de volgende drie labo's ontwikkelen we de **backend voor een blog** met een achterliggend Content Management System (CMS). De beheerder van de blog zal eenvoudig met een webformulier nieuwe artikels kunnen toevoegen aan de blog zonder dat er iets moet veranderen aan de broncode. Al deze functionaliteit wordt ter beschikking gesteld via een REST-API.

2 Spring-Framework

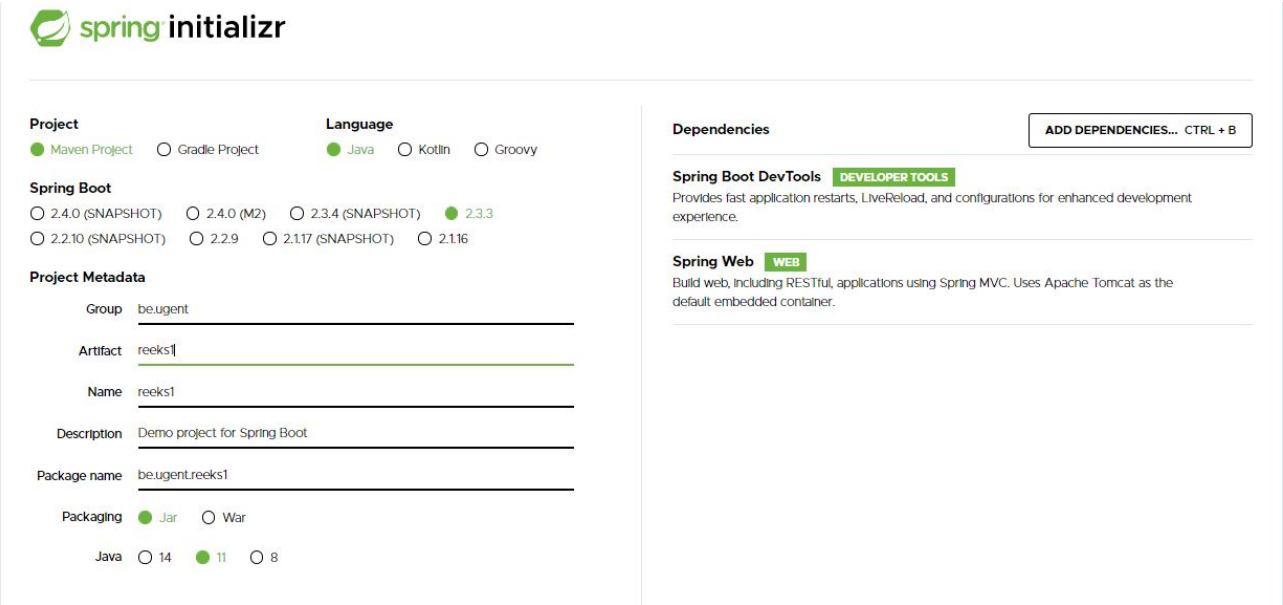
Spring is een open-source, lightweight container en framework voor het bouwen van **Java enterprise applicaties**. Dit framework zorgt ervoor dat je als developer kan focussen op het probleem dat je wil oplossen met een zo min mogelijk aan configuratie. Het Spring-ecosystem

bestaat uit verschillende projecten (Zie http://www.spring.io/projects). In deze reeks starten we met een **basis Spring Boot applicatie** die dan stap voor stap uitgebreid wordt.

3 Nieuw project

SpringBoot helpt je om Spring-toepassingen te maken met minimale configuratie. Het genereert startprojecten voor allerlei types Spring-toepassingen. Op de site https://start.spring.io kan je een nieuw (maven)project aanmaken. Maven is een **build automation tool** dat onder andere alle dependencies beheert en het project uitvoert.

- Maak een nieuw project aan met Spring Initializr en open het in je IDE (je kan best IntelliJ gebruiken).
  - Voeg bij dependency "Spring Web" en "Spring DevTools" toe. Andere dependencies voegen we later manueel toe.
  - Optioneel: vul project metadata in.



- Bestudeer de gegenereerde files, waar kan je de toegevoegde dependencies terug vinden?
- Start de server en bekijk de logs. Als alles goed ging, heeft onze applicatie een Tomcat Server op poort 8080 gestart. http://localhost:8080

4 REST-API - Deel 1

Via een REST-API kunnen we eenvoudige CRUD-operaties (Create, Read, Update en Delete) uit voeren op resources, in ons geval blogposts. Spring laat ons toe om een REST-API aan te maken zonder dat we zelf moeten instaan voor de conversie tussen Java-objecten en JSON/XML/..

- Maak een nieuwe Java-klasse `BlogPost` aan die een **blogpost** voorstelt. Een blogpost heeft minimaal een titel en content. Zorg ervoor dat deze klasse een default constructor heeft en getters en setters voor alle attributen.

- Maak een **blogpost DAO**-klasse (Data Access Object) `BlogPostDao` die de blogberichten bijhoudt in een collectie. Deze klasse simuleert een verbinding met een databank, later gebruiken we een echte database. Voeg een "Hello World" blogpost toe aan de collectie en voorzie een methode om alle posts op te halen. Door de klasse te annoteren met **@Service**, maakt Spring een bean aan die dan gebruikt kan worden voor dependency injection.
- Maak een **Controller**-klasse die de HTTP-requests zal afhandelen (extra info https://spring.io/guides/gs/rest-service/).
  - Voeg de annotatie **@RestController** toe aan deze klasse.
  - Injecteer de BlogpostDao met dependency injection. Hiervoor maak je een constructor aan die een parameter van het type "BlogPostDao" verwacht of gebruik "@Autowired" bij het attribuut. Spring injecteert tijdens constructie dan een instantie van de DAO.
  - Annoteer een methode met **"@GetMapping"** om een endpoint te implementeren dat alle blogposts in onze DAO teruggeeft.
- Herstart de server en bekijk de blogposts in de browser.
- Gebruik vervolgens **Postman** (download van https://www.postman.com/product/api-client/) om de "Hello World"-blogpost op te halen in zowel JSON- als XML-formaat, gebruik hiervoor accept-headers van het HTTP-bericht. Gebruik je liever een commandline tool, dan is **curl** een alternatief. Open een console-venster en gebruik curl zoals beschreven in How to send a header using a HTTP request through a curl call?. Wat merk je als je XML probeert op te halen?
  Tip oplossing:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

5 REST-API - Deel 2

In deel 1 hebben we een basis REST-API gemaakt met één endpoint om alle blogpost op onze server terug te sturen in het gewenste formaat van de client. In deel 2 focussen we op alle mogelijke (CRUD) operaties die we op de posts kunnen uitvoeren.

- Voordat we endpoints kunnen toevoegen aan de controller moeten we de **DAO** uitbreiden met volgende methodes:
  - Post toevoegen
  - Post verwijderen
  - Specifieke post opvragen op id
  - Post updaten
- Implementeer volgende REST-functionaliteit in de **Controller**, voor elk endpoint van je API maak je een functie met de juiste annotatie (**"@GetMapping", "@DeleteMapping", "@PutMapping", "@PostMapping"**).
  - Eén specifieke blogpost ophalen a.d.h.v een id. Indien een onbestaande blogpost wordt opgehaald moet er een `BlogPostNotFoundException` gegooid worden. Deze exceptie

wordt dan opgevangen door een handler. Zorg ervoor dat de HTTP-status "404 not found" is. Maak gebruik van **"@PathVariable"** om een deel van de url op te vragen.

- Een bericht toevoegen. Maak gebruik van **"@RequestBody"** om de body van de HTTP-request op te vragen. De HTTP-response bevat de locatie header en HTTP-status 201: created (zie What is the preferred way to specify an HTTP "Location" Response Header in Spring MVC 3?)

```
// uriBuiler is van het type UriComponentsBuilder
// en wordt meegegeven als parameter
URI location = uriBuilder
        .path("/blogposts/{id}")
        .buildAndExpand(postAdded.getId())
        .toUri();
return ResponseEntity.created(location).build();
```

- Een bericht verwijderen, resulteert in een 204 HTTP-status.
- Een bericht aanpassen, resulteert in een 204 HTTP-status.
- Test elke actie uit in Postman of curl.

6 Testen van de REST-API

Tot nu toe hebben we de REST-API telkens manueel getest met Postman. Eens onze applicatie groeit, willen we niet bij elke aanpassing alle endpoints opnieuw manueel testen, daarom maken we enkele testen in Java die dit automatisch voor ons controleren. Vanuit Java-code kan je gebruikmaken van de klasse **"Web(Test)Cient"**. Deze laat ons toe om de REST-API aan te spreken.

- Voeg testen toe aan het project.
  - In de folder **src/test/java** vind je een klasse geannoteerd met "@SpringBootTest".
  - **"Web(Test)Cient"** maakt deel uit van een apart test-framework. Je moet dus de nodige afhankelijkheden toevoegen, door de scope op test te plaatsen zal deze dependency enkel beschikbaar zijn tijdens testing.

    ```
    <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
            <scope>test</scope>
    </dependency>
    ```

  - Om dit test-framework te gebruiken, voeg je de annotatie **"@AutoConfigureWebTestClient"** toe. De eerste annotaties geeft aan dat de webapplicatie gestart moet worden op een andere poort zodat er geen conflicten zijn. De laatste annotatie zorgt dat de `WebTestClient` geïnjecteerd kan worden.

    ```
    @SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
    @AutoConfigureWebTestClient
    ```

  - Elke methode in deze klasse met de annotatie **"@Test"** gebruik je om een aspect van de applicatie te testen. Binnen deze methodes gebruik je de klasse `WebTestClient` om te communiceren met de REST-API. Maak gebruik van assertions om de resultaten te valideren. Zie ook Spring doc testing, benefit of assertThat over assert en webclient vs resttemplate.
- Implementeer een test voor elke CRUD operatie op een blogpost. Vergeet de BlogPostNotFoundException ook niet te testen.

7 Datalaag

Nu hebben we een functionele backend voor een blog geïmplementeerd. De REST-API houdt nu alle blogposts bij in het geheugen. Meer realistisch is het opslaan van de data in een database. Voor deze opdracht maken we gebruik van H2 embedded database maar deze zou eenvoudig kunnen vervangen worden door een ander type database.

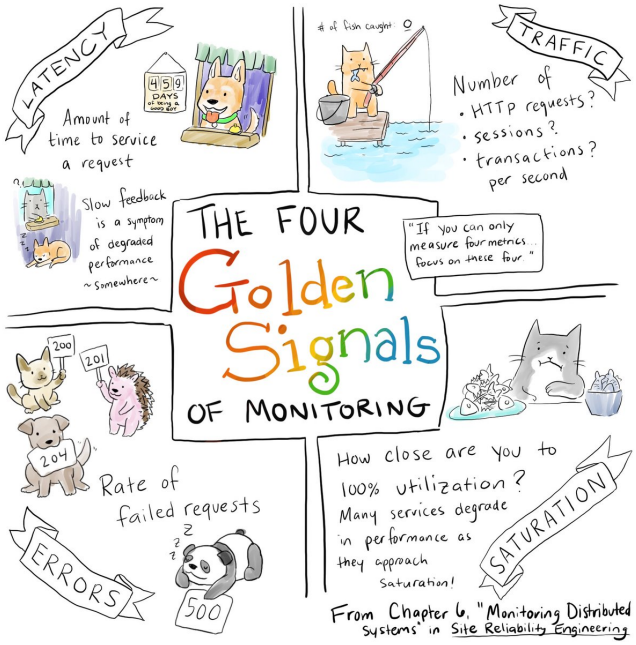
- Voeg de dependency voor de H2 database en JPA toe aan de dependencies voor dit project. https://www.baeldung.com/spring-boot-h2-database
- Bij de interactie met de database maakt het Spring-framework ons het leven veel makkelijker. We moeten enkel de data klasse correct annoteren en een interface aanmaken die zelf de interface "JpaRepository" uitbreidt. Spring zal dan een bean aanmaken met een connectie naar onze H2 database. Terwijl je zelf queries kan schrijven met sql biedt

jpa je de mogelijkheid aan om de queries te maken aan de hand van de methode naam. https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

- Maak een nieuwe klasse aan die exact dezelfde functionaliteit aanbiedt als onze blogpost DAO, maar nu gebruik maakt van de net aangemaakte repository om de data op te slaan in de databank. Vergeet deze klasse niet te annoteren met **"@Service"**. Zorg ervoor dat beide klassen dezelfde interface implementeren. Tip: maak gebruik van "refactoring" om een interface te extraheren uit de orignele DAO.
- De applicatie bevat nu 2 manieren om blogposts op te slaan (memory & DB). Door gebruik te maken van de OOP principes en **profiles** in spring kan je eenvoudig wisselen tussen beide opties. Maak een profile "test" aan die gebruik maakt van de DAO in het geheugen. Als dit profile niet actief is, maken we gebruik van de database. https://www.baeldung.com/spring-profiles
- Als we de testen uit vorige deel opnieuw uitvoeren, falen er een paar testen doordat de database het hello world blogpost niet bevat. Zorg ervoor dat de testen uitvoeren met het "test" profile.

8 Monitoring

Voor bedrijven is het monitoren van applicaties heel belangrijk. Bijvoorbeeld als een bankapplicatie opeens geen betalingen meer uitvoert, wil je dit als ontwikkelaar niet te weten komen via een boze email van de klant maar via een automatische melding. Hiervoor moet onze applicatie bepaalde *metrics* ter beschikking stellen. Volgens Google bestaan er 4 golden signals die je moet monitoren.



- Binnen spring bestaat er een project actuator die voor ons endpoints configureert die deze info ter beschikking stelt. Voeg de correcte dependency toe aan je project.
- Herstart je server en browse naar http://localhost:8080/actuator. Dit toont je een overzicht van alle beschikbare endpoints.
- Default zijn alle enpoints met gevoelige informatie over de applicatie uitgeschakkeld. Activeer

de *metrics* endpoint. Op welke url vind je nu de *up time* van de server en het aantal *HTTP requests*?

- De bestaande metrics zijn niet altijd voldoende, vaak wil je ook applicatie specifieke informatie monitoren. In ons geval zijn dit de operaties op een blog post. Spring Actuator bevat de Micrometer bibliotheek. Deze laat ons toe om *metrics* aan te maken onafhankelijk van het later gebruikte monitoring systeem en kent 4 types: counters, gauges, timers en histogrammen. Voorzie een metric van het juiste type dat voor elk uitgevoerde CRUD operaties het aantal bijhoudt.

  Tip: Maak gebruik van tags en een goede naming convention. http://localhost:8080/actuator/metrics/{metric_name}?tag={tag_key}:{tag_value}

9 Beveiliging

Uiteraard kan beveiling niet ontbreken in onze blog. Enkel de admin van de blog mag een post plaatsen die voor iedereen zichtbaar is.

- Voor de security maken we gebruik van "Spring Security". Voeg de dependency manueel toe aan "pom.xml" (maven). Door deze dependency toe te voegen worden alle HTTP-aanvragen default doorgestuurd naar een login-pagina. In de logs vind je een random password voor de default user "user". Hiermee zou je moeten kunnen inloggen. Zie ook https://docs.spring.io/spring-security/site/docs/current/reference/html5/#getting-maven-boot.

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

- Voeg admin account toe, met de rol ADMIN. Maak een klasse aan die overerft van "WebSecurityConfigurerAdapter" en annoteer die met "@Configuration". Vergeet de paswoorden niet the encrypteren. https://www.baeldung.com/java-config-spring-security
- In dezelfde klasse kunnen we ook configureren welke pagina's beveiligd moeten worden. Zorg ervoor dat enkel de admin posts kan toevoegen, aanpassen of verwijderen en iedereen posts kan opvragen. Maak gebruik van "GlobalMethodSecurity". https://www.baeldung.com/spring-security-method-security
- Doordat we de default instellingen nu overschrijven, moeten we ook explicit het type van authentication instellen. Voor deze opdracht mag je gebruik maken van basic authentication.
- Test of de beveiling correct werkt zowel in de browser als met Postman. Met Postman zal je merken dat je een 403 - Forbidden ontvangt als je een nieuwe post aanmaakt, dit is doordat CSRF dit voorkomt. Disable CSRF om met Postman de beveiliging uit te testen. https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html
- Update de testen door de basic authentication toe te voegen aan de testresttemplate/web-testclient.

10 Afwerking

In dit laatste deel link je je backend aan de gegeven front-end. Indien je voorgaande stappen nauwkeurig hebt gevolgd zou, moet je geen (grote) aanpassingen meer maken.

- Plaats de startcode (JS & HTML) onder **/src/main/resources/static**. Alles in deze map is statisch beschikbaar vanuit de browser.
- Maak indien nodig aanpassingen in **admin.js** (url, attributen).
- Beveilig de **admin.html** pagina zodat enkel ingelogde admins deze kunnen zien.
- Op de index pagina is er code voorzien om te zoeken naar blogposts. Met de opgedane kennis uit vorige delen zou je deze functionaliteit moeten kunnen implementeren op de server. Kies zelf op welke manier je wil zoeken (bv. in de titel, content of beide).

11 Uitbreidingen (optioneel)

Nu hebben we een werkend content management system voor een eenvoudige blog maar er zijn nog veel uitbreidingen mogelijk.

- Zorg voor paginering.
- Voeg tags toe aan blogposts en laat toe om hierop te filteren.