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

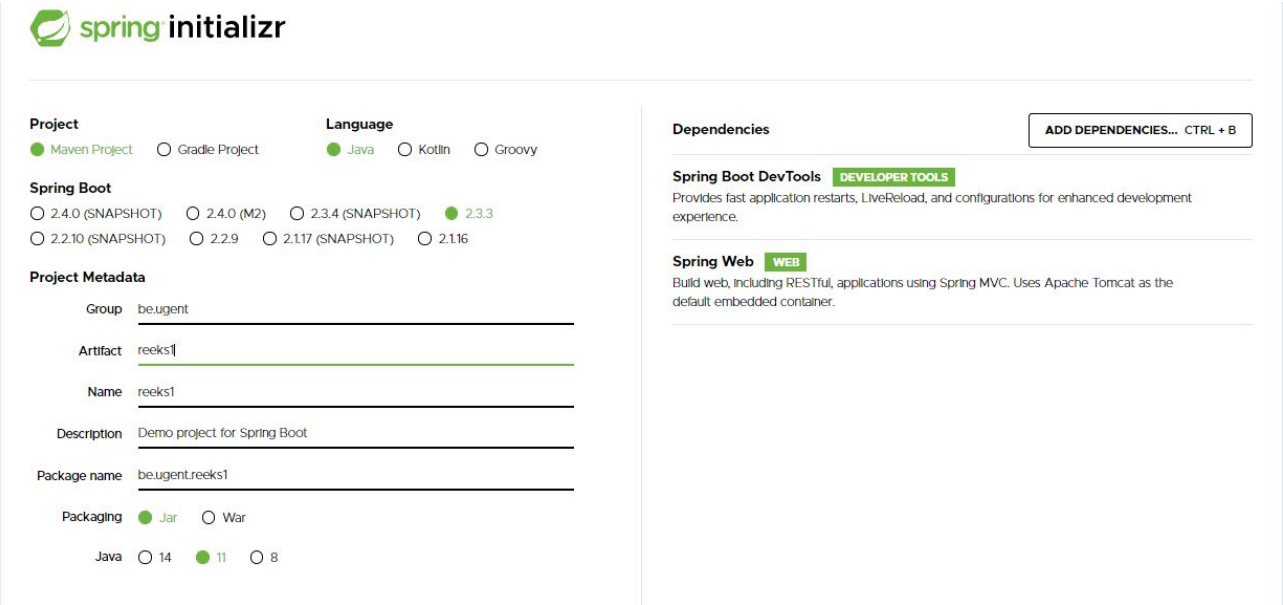
Spring is een open-source, lightweight container en framework voor het bouwen van **Java enterprise applicaties**. Dit framework zorgt ervoor dat je als developer kan focussen op het probleem dat je wil oplossen met een zo min mogelijk aan configuratie. Het Spring-ecosystem

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.