FACULTEIT INGENIEURSWETENSCHAPPEN
EN ARCHITECTUUR

Labo frameworks voor serverapplicaties
Reeks : REST, Spring, monitoring, testen en JPA

2 en 9 oktober 2024

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

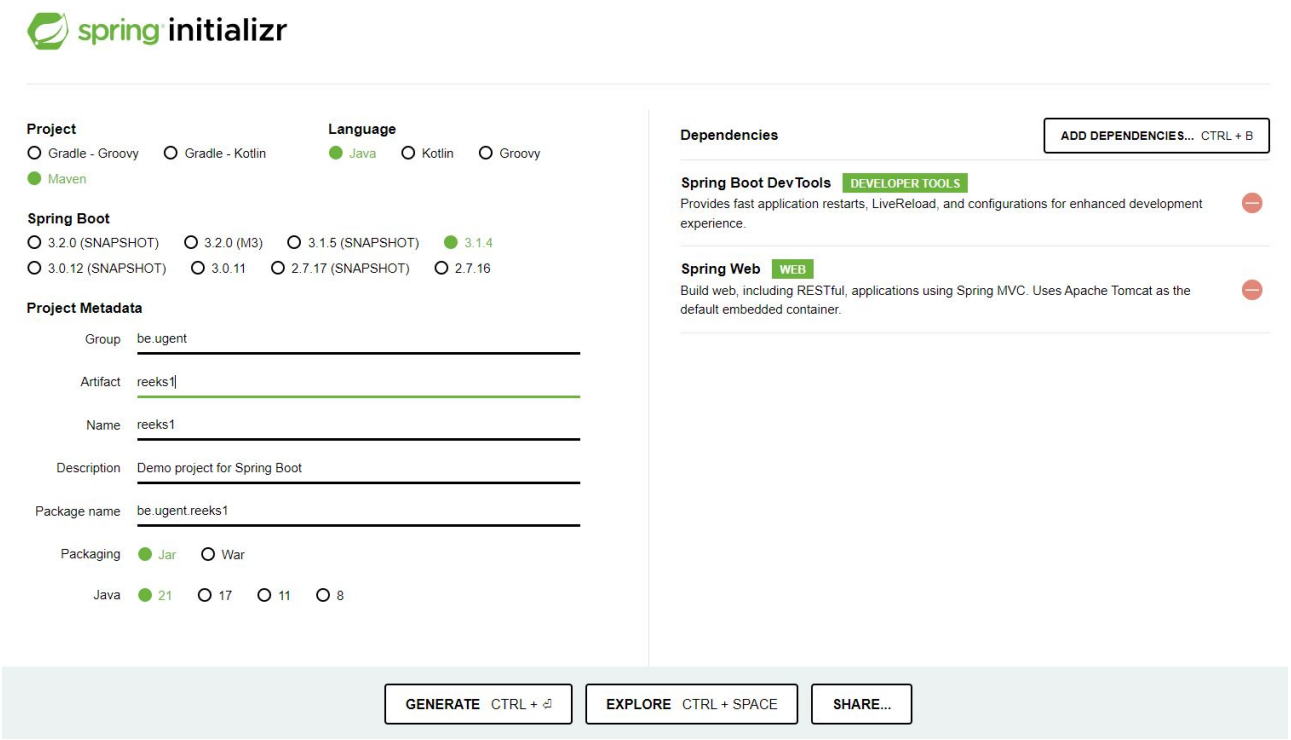
Spring is een open-source, lightweight container en framework voor het bouwen van **Java enterprise applicaties**. Dit framework zorgt ervoor dat je als developer kan focussen op het

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 en laat je startprojecten genereren voor allerlei types toepassingen. Via de website https://start.spring.io of rechtstreeks in IntelliJ kan je een nieuw (maven)project aanmaken. Maven is een **build automation tool** die 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
- Bezoek jouw webserver via de browser http://localhost:8080. Wat krijg je te zien en is dit te verwachten?

4 REST API - Deel 1

Via een REST API kunnen we eenvoudige CRUD-operaties (Create, Read, Update en Delete) uit voeren op resources, in dit geval blogposts. Spring laat ons toe om een REST API

aan te maken zonder dat we zelf moeten instaan voor de serialisatie van Java-objecten naar 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) `BlogPostDaoMemory` die de blogberichten bijhoudt in een collectie. Deze klasse simuleert een verbinding met een databank, later zullen we een echte database gebruiken. Voeg ook 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/).
  - Injecteer de BlogPostDaoMemory met dependency injection.
  - Implementeer een endpoint 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/downloads/) 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

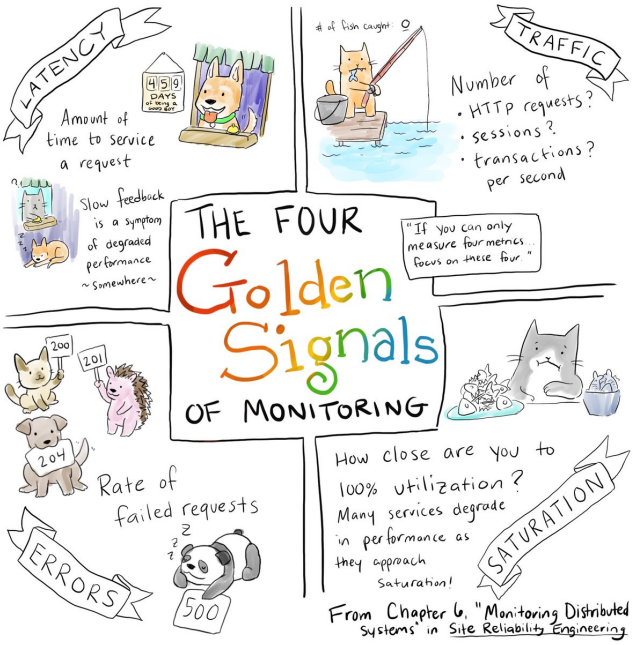
In deel 1 hebben we een basis REST-API gemaakt met één endpoint om alle blogpost op onze server terug te sturen in het gewenste formaat van de client. In deel 2 focussen we op alle mogelijke (CRUD) operaties die we op de posts kunnen uitvoeren.

- 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**.

- Eén specifieke blogpost ophalen a.d.h.v een id. Indien een onbestaande blogpost wordt opgehaald moet er een `BlogPostNotFoundException` gegooid worden. Test de verschillende manieren uit om deze exceptie op te vangen: annoteren foutklasse, handlermethode in de controller, opgooien van een `ResponseStatusException`, ... Meer info op https://www.baeldung.com/exception-handling-for-rest-with-springZorg ervoor dat de HTTP-status "404 not found" is.
- Een bericht toevoegen. De HTTP-response bevat de locatie header en HTTP-status 201: created (zie Add location header to Spring MVC's POST response?)
- Een bericht verwijderen, resulteert in een 204 HTTP-status, als het bericht bestaat. Anders is de HTTP-status 404.
- Een bericht aanpassen, resulteert in een 204 HTTP-status. Indien het id in het path niet overeenkomt met het id in de body wordt een HTTP-status 409 - conflict terug gestuurd. Bestaat de blogpost niet, dan is HTTP-status 404 - not found. Maak een oplossing die gebruik maakt van `ResponseEntity` en één die fouten opgooit.
- Test elke actie uit in Postman of curl.

6 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 endpoints 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 blogpost. Spring Actuator bevat de Micrometer bibliotheek. Deze laat ons toe om metrics aan te maken onafhankelijk van het later gebruikte monitoring systeem en kent verschillende types: counters, gauges, timers, ... 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}

7 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 vanuit de code.

- Voeg testen toe aan het project.
  - **"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>
    ```

  - Meer info op Spring doc testing en benefit of assertThat over assert.
- Implementeer een test voor elke CRUD-operatie op een blogpost. Test ook het ophalen van een blogpost in XML- en JSON-formaat alsook de BlogPostNotFoundException.

8 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/reference/jpa/query-methods.html
- 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. Zorg ervoor dat beide klassen dezelfde interface implementeren. Tip: maak gebruik van "refactoring" om een interface te extraheren uit de originele DAO.
- Doordat we hier kozen voor een in-memory database starten we telkens van een lege database. Zolang de applicatie runt is het mogelijk om de inhoud van deze database te bekijken. Voeg volgende configuratie "spring.h2.console.enabled=true" toe aan `application.properties`. In de logs kan je nu vinden hoe je de database kan bekijken.
- 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.

- Als we de testen uit vorige deel opnieuw uitvoeren, falen er een paar testen doordat de database de “hello world”-blogpost niet bevat. Zorg ervoor dat de testen opnieuw slagen door het “test” profile te gebruiken.