



Spring Boot

Spring Framework, meestal afgekort tot Spring, is een open source framework gericht op ontwikkeling van software in de programmeertaal Java.

Het Framework maakt het mogelijk om op een simpele manier web applicaties te maken en te runnen. Het is ontstaan vanuit de doelstelling om de ontwikkeling van Java Enterprise applicaties (J2EE, JEE) te vereenvoudigen. Spring vergemakkelijkt de ontwikkeling van Java-toepassingen zowel in offline- als online-applicaties. De grootste voordelen van het framework zijn de slanke broncode en de eenvoud waarmee aanpassingen gedaan kunnen worden. Het Framework wordt al sinds 2002 ingezet en bestaat uit verschillende modules voor verschillende functionaliteiten.

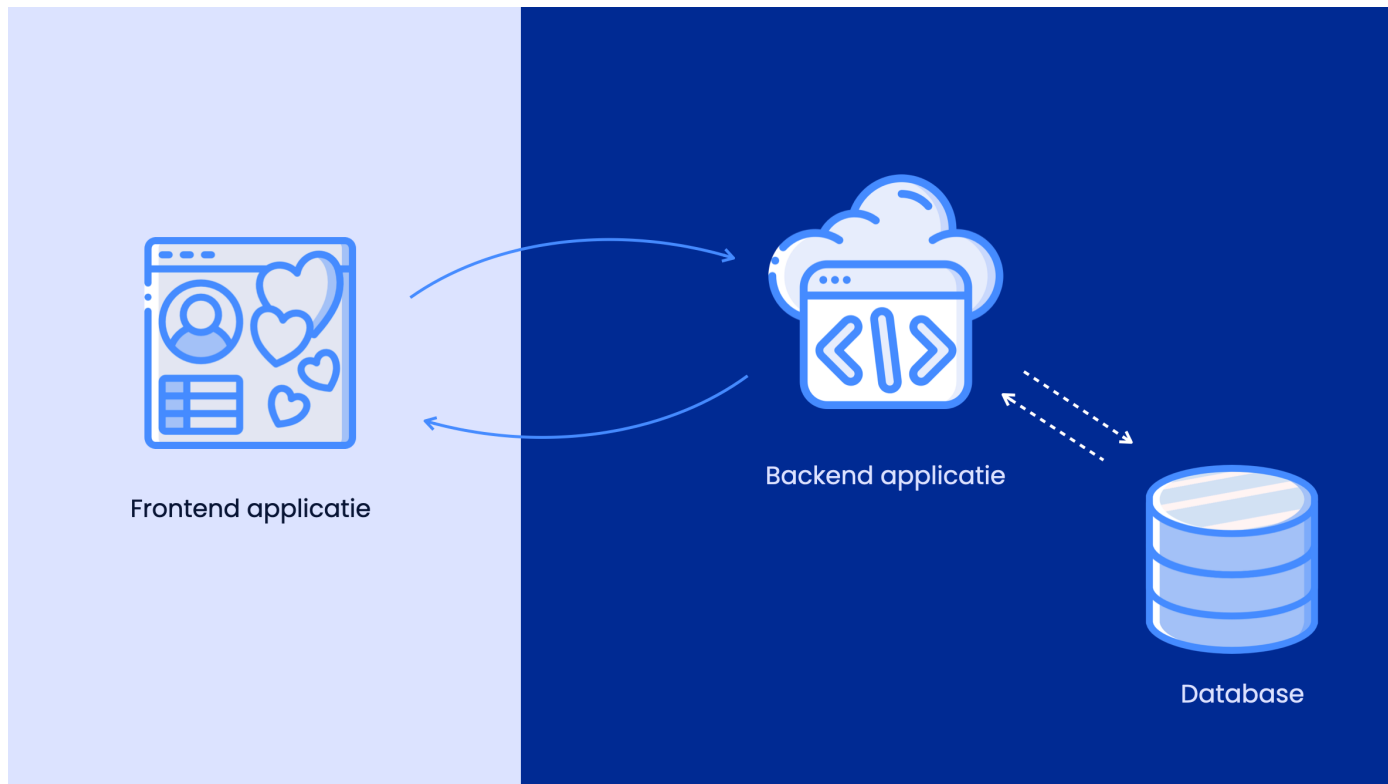
1. Inleiding

Spring is de perfecte keuze voor de meest uiteenlopende toepassingen in Java. Met Spring is het mogelijk om op een object georiënteerde manier applicaties te schrijven die flexibel, beter leesbaar en beter te testen zijn. In deze cursus gaan we leren hoe je een Java applicatie kunt ontwikkelen met het Spring Framework.

1.1 Inleiding

Software die je installeert op je computer is geschreven in één enkele programmeertaal. Denk bijvoorbeeld aan Adobe Photoshop: deze applicatie is zowel verantwoordelijk voor de functionaliteit, de grafische interface (het uiterlijk) en de verbinding tussen die twee (de logica). Want net als dat de applicatie een plekje heeft op het besturingssysteem, hebben jouw bewerkte foto-bestanden dat ook.

Wanneer we spreken over **webapplicaties** - applicaties die je kunt gebruiken via het internet - gebruik je de applicatie niet op je computer, maar in de browser. Een webbrowser, in tegenstelling tot jouw computer, is alleen in staat om webpagina's *weer te geven*. Wanneer de gebruiker data invoert of aanpast, is het niet mogelijk deze data ergens op te slaan! Wanneer de browser wordt afgesloten of de pagina wordt ververs, zullen alle gemaakte aanpassingen verdwijnen. Om deze reden bestaan webapplicaties uit twee losse delen: een **frontend** en een **backend**. De frontend is het gedeelte dat de gebruiker in de browser ziet. De backend is niet zichtbaar en verantwoordelijk voor de onderliggende logica van de applicatie en de dataopslag. De frontend en backend communiceren voortdurend met elkaar!



Wanneer een gebruiker diens profielpagina wil bekijken, heeft de webpagina natuurlijk informatie nodig over de gebruiker. De frontend doet direct een verzoek naar de backend: "Mag ik alle profiel-informatie van gebruiker #45567?". Het verzoek wordt via het internet verstuurd naar een specifiek webadres. Wanneer dit verzoek bij de backend aankomt, wordt er in de database gezocht naar de informatie van gebruiker #45567. Eenmaal gevonden, wordt deze informatie als platte tekst terug verstuurd naar de frontend. Dit zorgt ervoor dat de informatie op een fijne, overzichtelijke manier weergegeven wordt voor de gebruiker.

Vervolgens valt het de gebruiker op dat diens woonadres niet meer actueel is, dit moet aangepast worden! De informatie die in de invoervelden is ingevoerd, wordt door de frontend verzameld en meegestuurd met een nieuw verzoek aan de backend.

"Wil jij het bijgevoegde adres gebruiken om het adres van gebruiker #45567 te overschrijven?"

Omdat deze informatie naar een specifiek webadres wordt gestuurd, weet de backend precies hoe dit verzoek afgehandeld moet worden. De backend doorzoekt de database om de juiste gebruiker te vinden en zorgt er vervolgens voor dat de adres-data wordt overschreven. Wanneer dit gelukt is, stuurt de backend een succes-code terug naar de frontend. Zo kan de frontend aan de gebruiker bevestigen dat de adreswijziging gelukt is.

Wanneer wij backend applicaties willen ontwikkelen voor het web, zouden we dit theoretisch gezien kunnen bouwen in alleen Java. Dit kost echter heel veel tijd. We maken daarom gebruik van vele handige tools en frameworks die het Java-landschap heeft, om een volwaardige backend applicatie te kunnen bouwen.

1.2 Spring

Binnen het Java Ecosysteem stikt het van de tools die je kunt gebruiken om sneller (en vaak betere) software te ontwikkelen. Grofweg kunnen we deze tools verdelen in **frameworks** en **libraries**.

Libraries zijn stukken Java code die een andere developer heeft geschreven. Deze code kunnen we gebruiken om standaard problemen op te lossen, zodat we niet telkens het wiel opnieuw uit hoeven vinden. Neem bijvoorbeeld cryptografie: dat (groene) slotje in de adresbalk van je webbrowser.



Deze herken je vast. Dat slotje representeert een heel scala aan veiligheidsmaatregelen, waaronder encryptie (veilige versleuteling) van data. Het zal je niet verbazen dat zo iets als encryptie ontzettend moeilijk te programmeren is. Ten eerste heb je kennis nodig van zeer complexe wiskunde (cryptografie) en ten tweede mag jouw code geen enkele fout bevatten, omdat dit kan leiden tot slechte en onveilige encryptie. Fijn dat we daarvoor gewoon een encryptie library kunnen gebruiken!

Frameworks, daaropvolgend, gaan nog een stapje verder. Je zou kunnen stellen dat een framework ervoor zorgt dat je applicatie al voor een groot deel klaar is, vóór je eraan begint. Dat komt omdat applicaties in de basis allemaal dezelfde onderdelen nodig hebben: data opslag, dataverwerking, beveiliging, etc. Een Framework zet daarom een soort standaard raamwerk voor je op waarbinnen jij de vrijheid hebt om de code toe te voegen die uniek is voor jouw applicatie. Hoe dit precies werkt, behandelen we in deze cursus.

Door gebruik te maken van libraries en frameworks, kun je met Java veel verschillende soorten applicaties schrijven. Denk aan desktop applicaties, commandline tools en zelfs games! We zullen ons binnen deze leerlijn gaan focussen op het werken met het Spring framework. Spring is één van de veelzijdigste en krachtige frameworks in Java. Er bestaan ook nog andere soorten, zoals Quarkus en Micronaut, maar deze zullen we tijdens de leerlijn niet behandelen.

Door Spring te gebruiken voor onze webapplicaties besparen we onszelf een hoop tijd en moeite. Iedere webapplicatie kenmerkt zich door de volgende drie eigenschappen:

- De applicatie moet beschikbaar zijn via het internet;
- De applicatie moet iets doen met de data die het binnenkrijgt;
- De applicatie moet data op kunnen slaan in een database.

Laten we daar één voor één dieper op in gaan.

De applicatie moet beschikbaar zijn via het internet

Met die beschikbaarheid bedoelen we dat de applicatie zich moet gedragen als een **webserver**: wanneer een gebruiker een **url** invoert in de adresbalk van diens browser, moet de server de juiste webpagina serveren. Maar ook wanneer dit via een **url** gebeurt, moet de server de juiste **data** terugsturen. Een webserver, ook wel **HTTP-server** genoemd, kenmerkt zich door het feit dat deze in staat is om op verzoeken te reageren die binnenkomen via het internet.

Een url vraagt een website op via het internet en een url vraagt een stukje data op via het internet

Webservers zijn niets nieuws: dit is namelijk al jaren de basisfunctionaliteit van het internet. Iedere backend applicatie heeft een webserver nodig! Hoewel het leuk en leerzaam is om een eigen webserver te bouwen, is het voor productiesoftware erg fijn wanneer je deze "van de plank kunt pakken". Omdat Spring standaard een geconfigureerde HTTP-server bevat, kunnen we bij het bouwen van een nieuw project direct gebruik maken van deze functionaliteit nog voordat we een regel code hebben hoeven schrijven. Hoe fijn is dat?!

Aan alleen een webserver hebben we echter niet genoeg. We moeten er namelijk voor zorgen dat we een logische structuur hanteren in onze applicatie, waar continue data doorheen stroomt. Je kunt dit vergelijken met een website: daar staat niet alle informatie op de homepage pagina gepropt. Die informatie is verdeeld over specifieke pagina's die naar elkaar linken om jou op een logische manier door de website te sturen. Zo verwacht je op de homepage van www.novi.nl een overzicht te zien van onze hogeschool en opleidingen, als ik een gesprek wil met een studietoelichting moet ik op de contact-pagina zijn (www.novi.nl/contact) en als ik wil inloggen op EdHub moet ik naar mijn account navigeren (www.novi.nl/login). Daarnaast mag mijn account alleen toegankelijk zijn als ik ingelogd ben - en dan enkel en alleen voor mij. Op dezelfde manier is de data die onze webserver aanbiedt verspreid over verschillende url's.

De applicatie moet iets doen met de data die binnenkomt

Dit statement is iets lastiger, gezien we als developers natuurlijk verantwoordelijk zijn voor de functionaliteit van de applicatie. Iedere applicatie is anders, net als de data die we nodig hebben om de applicatie te laten werken. Een applicatie om online in te bankieren wordt heel anders opgezet dan een webshop. Laten we daarom voor nu even aannemen dat ze hier wel onze eigen code voor moeten schrijven.

De applicatie moet data op kunnen slaan in een database

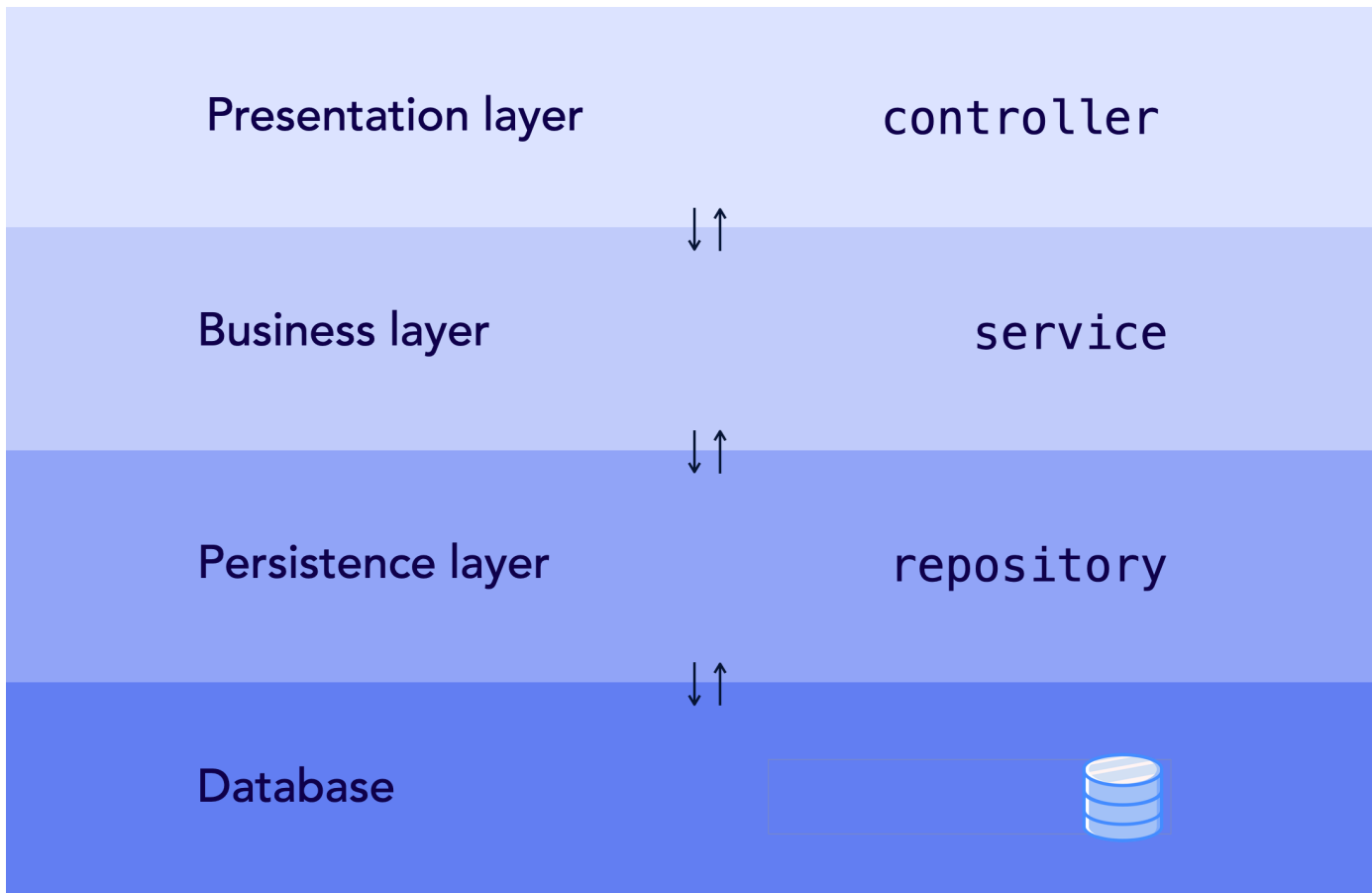
Data opslaan in een database is juist wel weer een proces dat gemakkelijk te standaardiseren is! Wij zijn namelijk niet de eerste die onze data ergens willen opslaan... Dat is het hele idee van een backend. Daarnaast spreken bijna alle databases dezelfde taal: **SQL** (Structured Query Language). Ook hiervoor geldt dat Spring automatisch met de meeste databases kan praten. Hoewel dit ietsje meer werk vergt dan de HTTP-server, krijg je deze functionaliteit van Spring cadeau.

1.3 Spring Boot

Intmiddels kan je wel begrijpen waarom Spring zo'n krachtig framework is. Het faciliteert developers in het razendsnel bouwen van complexe applicaties, omdat we ons niet bezig hoeven te houden met terugkerende patronen zoals het managen van de database of het opzetten van webservers. We kunnen ons direct focussen op de **businesslogica** van de applicatie: de functionaliteiten die de applicatie uniek maken.

Tijdens de cursus werken we met **Spring Boot**, dit is gebaseerd op het Spring framework. Wat de exacte verschillen zijn tussen Spring Boot en het Spring Framework, leggen we later uit. Voor nu refereren we naar zowel het Spring Framework als Spring Boot wanneer we het over Spring hebben.

Spring baseert zich op "Convention over Configuration". Dit houdt in dat als je Spring gebruikt, "dingen" werken zoals de meeste developers het verwachten: volgens de gebruikelijke standaarden (conventies). Wil je die "dingen" toch anders laten werken? Dan kun je dit configureren (aanpassen) of zelf programmeren. Hierbij is het wel zo dat webapplicaties vaak volgens een standaard architectuur worden gebouwd. Deze software architectuur bestaat uit meerdere lagen die met elkaar samenwerken.



Om dit beter te begrijpen, kunnen we het beste de drie belangrijke eigenschappen van een webapplicatie er weer bijpakken:

De applicatie moet beschikbaar zijn via het internet

Om dit te bereiken gebruikt Spring **controllers**. Een controller is een Java class die, op een specifieke *url*, via het internet te bereiken is. Wanneer je in jouw webbrowser naar dat specifieke adres navigeert, zorgt het framework dat de class - de controller - die gekoppeld is aan dat adres de vraag ontvangt en vervolgens een antwoord terugstuurt. Zo kun je bijvoorbeeld een User-controller hebben voor alle verzoeken omtrent gebruikersinformatie.

De applicatie moet iets doen met de data die het binnenkrijgt

Om dit te bereiken gebruikt Spring **services**. Een service is een Java class die de business logica uitvoert. Denk bijvoorbeeld aan de logica die we nodig hebben om banktransacties correct te kunnen verwerken, maar ook functionaliteiten zoals:

- Als een gebruiker de waarde van zijn huis en inkomen instuurt, wordt de hypotheekrente berekend;
- Als de betaling van een order binnenkomt, moet de geconfigureerde auto die gekoppeld is aan deze order, besteld worden;
- Als de ingelogde gebruiker naar zijn bestellingsoverzicht navigeert, moet de gebruiker alle bestellingen tot 1 jaar terug kunnen inzien.

Om de software begrijpelijk te houden, delen we deze logica op in verschillende services met ieder hun eigen doel

De applicatie moet data op kunnen slaan in een database

Om dit te bereiken gebruikt Spring **repositories**. Een repository is een Java class die de koppeling naar de database verzorgt. We kunnen een repository gebruiken om data naar de database te sturen (op te slaan) of om specifieke data uit te lezen.

Wanneer een verzoek of data binnenkomt in onze backend, zullen deze altijd via de controller, service en repository moeten "reizen" om in de database terecht te komen. Een *short-cut* nemen is niet mogelijk. Laten we eens naar de situatie kijken waarin we een webshop hebben ontwikkeld. De frontend stuurt het volgende verzoek:

"Geef mij het bestellingsoverzicht van gebruiker #j4454"

Dit komt binnen bij de Order-controller op de url `www.onze-webshop-backend.nl/api/orders/j4454`. De Order-controller heeft geen interne kennis van de webshop en geeft het verzoek door aan de Order-service. De Order-service weet precies wat er moet gebeuren! Op basis van de datum van vandaag, kan het tijdsbestek worden vastgesteld waarbinnen de opgevraagde orders moeten vallen. De Order-service stuurt een verzoek naar de order-repository:

"Haal de bestellingen op die gekoppeld zijn aan gebruiker #j4454. Geef mij alleen de bestellingen die gedaan zijn vanaf 01-01-2021 en succesvol afgerond zijn".

De Order-repository vertaalt dit verzoek naar SQL en vraagt op die manier aan de database om de juiste informatie op te halen. Eenmaal opgehaald geeft de repository de data terug aan de Order-service. De Order-service zet de data om naar een format dat we kunnen versturen via het internet: **JSON**. De Order-controller stuurt deze JSON data weer terug naar de frontend. Zo zie jij als gebruiker jouw bestellingsoverzicht in de browser!

1.4 Een spring project maken

Het maken van een Spring project hoeven we gelukkig niet met de hand te doen. Hiervoor maken we gebruik van de **Spring Initializr**: een tool die direct een geconfigureerd project voor ons maakt dat wij kunnen openen in een IDE. Deze tool vindt je op <http://start.spring.io>. Zorg dat je de volgende waardes invult en aanvinkt:

- **Project:** Maven
- **Language:** Java
- **Spring Boot:** 3.0.x (de subversies kunnen verschillen, zolang de main-versie hoger dan 3 is)
- **Group:** Deze mag je uiteraard zelf bedenken! Wat dacht je van `com.novi.org.nar1eike` of `com.bananas`?
- **Artifact & Name:** Hier vul je de naam van jouw project in
- **Packaging:** JAR
- **Java:** versie 17

Tenslotte mag je aan de rechterkant een dependency toevoegen. Gezien we een webserver willen maken, hebben we de dependency **Spring Web** nodig.



Project

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ Java ☐ Kotlin ☐ Groovy

☒ Maven

Spring Boot

☐ 3.0.2 (SNAPSHOT) ☒ 3.0.1 ☐ 2.7.8 (SNAPSHOT) ☐ 2.7.7

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 19 ☒ 17 ☐ 11 ☐ 8

Dependencies

[ADD DEPENDENCIES...](#)

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Wanneer je tevreden bent, mag je op de 'Generate'-knop drukken (of CMD + Enter voor Mac / CTRL + Enter voor Windows). Jouw boilerplate project wordt dan automatisch als ZIP gedownload naar jouw Downloads-map. **Pak deze eerst even uit** en open daarna het project door te dubbelklikken op de pom.xml. Deze zal er ongeveer zo uit zien:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.1</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>com.example</groupId>
  <artifactId>testproject</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>testproject</name>
  <description>testproject</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

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

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

Het meeste zul je wel herkennen, maar er zijn twee zaken die opvallen: de <parent>-tag en de <plugin>-tag. Hoewel je de <parent>-tag waarschijnlijk nog niet hebt gebruikt, hebben we het hier wel kort over gehad in het hoofdstuk over Build Tooling. Met deze tag zorgen we er namelijk voor dat we standaard-configuraties meenemen uit een parent-POM, zodat je deze niet telkens opnieuw hoeft te schrijven. In ons geval krijgen we een hoop instellingen mee van de spring-boot-starter-parent:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.1</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Dit zorgt ervoor dat we de benodigde dependencies van Spring kunnen gebruiken en met behulp van dependency management wordt automatisch de versienummer geregeld. Daarom hebben de dependencies spring-boot-starter-web en spring-boot-starter-test geen versienummer: deze staan gespecificeerd in de parent-POM. Op die manier zorgt Spring dat alle Spring dependencies goed met elkaar samenwerken. Weer een zorg minder! In de <plugin>-tag zie je een build-plugin staan, de spring-boot-maven-plugin. Via de parent-POM wordt de juiste versie van deze plugin geconfigureerd. Het fijne is dat deze plugin een aantal 'doelen' toevoegt aan het project, waardoor je het project gemakkelijker kunt aansturen. Zo kun je een Spring applicatie starten via de commandline, door het volgende in de terminal in te voeren:

```
mvn spring-boot:start
```

Wanneer we een Spring project starten, "draait" het project op een webadres dat alleen bereikbaar is op jouw eigen computer. Bij conventie is dit altijd `http://localhost:8080`. Om een Spring project op onze computer te draaien, hebben we geen internetverbinding nodig (vandaar de naam *local*). Het gebruik van een lokale hostnaam maakt het gemakkelijk om onze applicaties snel op te starten en geïsoleerd te kunnen gebruiken wanneer de applicatie in ontwikkeling is.

Wanneer de applicatie af is, zal de hostnaam moeten verwijzen naar de locatie van de server die het project host. Dit noem je een domeinnaam: een uniek webadres dat verwijst naar een webpagina of webserver op het internet. Net als de fietsenmaker een domeinnaam heeft gekocht voor zijn website (www.fietsenmaker-het-hoelje.nl) zul jij ook een domeinnaam moeten kopen voor jouw backend-applicatie wanneer je deze toegankelijk wil maken via het internet (www.bibliotheek-backend-server.nl).

Om de Spring applicatie vervolgens (lokaal!) weer te stoppen, gebruik je het volgende commando:

```
mvn spring-boot:stop
```

Ook kan de Maven-plugin het artifact "repackagen" met het commando:

```
mvn spring-boot:repackage
```

Ben je benieuwd wat deze plugin nog meer voor je kan doen? Lees dan de [uitgebreide documentatie](#) eens door.

1.5 PostgreSQL

Een webservice maakt gebruik van een database om gegevens op te slaan. Spring Boot kan gebruik maken van veel verschillende **Database Management Systemen**: Oracle, MS Server, MySQL, DB2, PostgreSQL en vele anderen.

In deze cursus wordt gebruik gemaakt PostgreSQL. [Download](#) en installeer PostgreSQL op jouw systeem. In de cursus wordt gebruik gemaakt van een database genaamd springboot met als eigenaar een gebruiker *springboot* met wachtwoord *springboot*.

Database: springboot

Gebruiker: springboot

Wachtwoord: springboot

1.6 Postman

Ten slotte gaan we gebruik maken van de applicatie [Postman](#) voor het versturen van HTTP-requests naar de webservice en het weergeven van de ontvangen response. Postman biedt de mogelijkheid verschillende soorten requests te versturen, data mee te geven en de gewenste authenticatiegegevens in te voeren.

Body

Status: 200 OK Time: 929ms Size: 566 B

```
{
  "data": {
    "shop": {
      "id": "gid://shopify/Shop/17681717",
      "name": "johns-apparel",
      "email": "johns@johns-apparel.com"
    },
    "customers": {
      "edges": [
        {
          "node": {
            "id": "gid://shopify/Customer/6581271756",
            "displayName": "Beatrice Alighieri",
            "phone": "+14155550271",
          }
        }
      ]
    }
  }
}
```

Response Size	
Body	231 B
Headers	335 B
Request Size	
Body	220 B
Headers	362 B
All size calculations are approximate	

2. RESTful webservice

In dit hoofdstuk worden webservices behandeld. En dan met name RESTful webservices. Hierin lees je voornamelijk achtergrondinformatie over de concepten die bij een RESTful webservice van belang zijn. Bijvoorbeeld hoe HTTP hier in een essentiële rol in speelt.

2.1 Wat is REST?

Wat is REST?

Zoals je inmiddels hebt geleerd, ga je tijdens deze cursus aan de slag met het ontwikkelen van een webservice. Omdat jouw backend door iedere willekeurige frontend aangesproken moet kunnen worden, is het handig als de communicatie volgens een vaste set regels verloopt. Er zijn verschillende protocollen die hiervoor kunnen worden gevolgd: één daarvan is **REST**. REST is een afkorting voor **RE**presentational **S**tate **T**ransfer. Het principe van REST is in 2000 bedacht door [Dr. Roy Thomas Fielding](#).

REST is dus geen Framework of Library, want het bestaat niet uit een heleboel voorgedefinieerde code die je kunt aanspreken. Het is een set van principes en beperkingen die definiëren hoe webstandaarden (zoals HTTP en URLs) moeten worden gebruikt om schaalbare en onderhoudbare webdiensten te creëren. Wanneer iedereen (frontenders, backenders en data engineers) zich aan deze principes en beperkingen houdt, kunnen we al onze applicaties (frontend, backend en database) vrijwel naadloos op elkaar aansluiten. We zijn hier echter bezig met een backend framework, dus wij focussen ons enkel op hoe

we de regels van REST in de backend kunnen implementeren om uiteindelijk een gedistribueerde webapplicatie te maken. Of web-API, mocht je cool willen doen bij je vrienden.

Wat is een gedistribueerde webapplicatie?

In de context van REST, verwijst "gedistribueerd" naar een systeem waarin verschillende delen van de applicatie of service zich op verschillende machines of locaties bevinden en met elkaar communiceren via het internet. In een gedistribueerd systeem draaien bijvoorbeeld de database, de backend en de frontend allemaal op een ander systeem of een andere Cloud-omgeving. REST is erop gebouwd dat deze applicaties, hoe verschillend ze ook zijn, toch in staat zijn om met elkaar te communiceren en samen te werken.

2.2 API's

Een API is een **Application Programming Interface**, maar waarschijnlijk zegt dat je nog steeds vrij weinig. Laten we het vergelijken met een situatie die we kennen. Als we een API in de praktijk willen uitbeelden, zouden we naar een restaurant gaan. Stel je voor dat je uit eten bent: je zit aan tafel en bent klaar om te bestellen. Je hebt een manier nodig om jouw bestelling door te geven aan de keuken, maar ook om jouw gerecht bij jou aan tafel terug te ontvangen. Dit kan de kok niet zelf doen, want die is aan het koken. Evengoed kun je dit niet zelf doen, want jij bent de klant (en anders was je wel thuis gaan eten).

Je hebt iets nodig om de klant die het eten bestelt (de frontend die de data nodig heeft) en de kok die het bereidt (de backend en de database waar de gegevens staan), met elkaar te verbinden. Hier komt de ober - of in ons geval de API - in beeld! De ober neemt je bestelling op, brengt die naar de keuken, en vertelt de keuken welk gerecht ze mogen bereiden. Als het klaar is brengt hij de response, in dit geval het eten, terug naar jou. Een API is dus eigenlijk simpelweg de tussenpersoon.

Door middel van een vastgestelde set regels (het REST protocol) begrijpen de applicatie en de database wat ze van elkaar willen. Dit doet een ober natuurlijk ook. Je kunt niet met je bestek op tafel slaan en verwachten dat de ober begrijpt dat je graag een bord pasta wilt bestellen. We beginnen onze bestelling altijd met "Ik zou graag wat eten willen bestellen" en vullen dit aan met onze bestelling: "Graag de pasta bolognese, alsjeblieft." In een API werkt dit net zo. We specificeren wat voor verzoek we doen (data opvragen, verwijderen of juist versturen) en waar dit verzoek naartoe moet, door middel van een specifiek webadres: het **endpoint**. Dat verzoek komt daarom bij onze database controller uit, omdat alleen die controller naar dat endpoint luistert.

Web-API's worden gebruikt in een breed scala aan toepassingen, zoals social media platforms, financiële diensten, e-commerce websites, enz. Het werken met een API is namelijk heel flexibel. Ontwikkelaars gebruiken web-API's om nieuwe applicaties te bouwen, bestaande applicaties uit te breiden en applicaties te integreren met andere systemen. Wil je een voorbeeld zien van een web-API? Kijk dan eens naar de [Poké API](#) die te gebruiken is om informatie over Pokemons op te vragen.

Side note: In de tekst hierboven spreken we over web-API, dan zul je je vast afvragen "Is er dan ook een niet-web API?" Die is er zeker! Je JDK zit er vol mee zelfs. Wanneer je bijvoorbeeld een ArrayList maakt in Java, dan maakt je gebruik van de Collections API die je kunt vinden in de package Java.Util.Collections.)

2.3 De regels van REST

Je mag jezelf pas een RESTful web-API noemen wanneer je je houdt aan de volgende regels:

- Client en Server staat los van elkaar. Dit betekent ook dat de code aan de client-kant (de frontend) zomaar omgeruild kan worden met een compleet andere applicatie, zonder dat de server (de backend) daar iets van hoeft te weten.
- Een REST applicatie is **stateless**. Dit houdt in dat de server niks hoeft te weten over de staat van de client. In REST wordt dit geïmplementeerd door gebruik te maken van *resources* in plaats van *commands*. Resources zijn de zelfstandige naamwoorden van het web. In de zin: "Geef mij een banaan", zou banaan een resource zijn. Een resource kan elk object, document of "ding" beschrijven dat je wilt opslaan of wilt verzenden over het web. Door deze beperking levert REST een betere prestatie en is het betrouwbaarder en schaalbaarder.
- De data is **cacheable**. Cacheable betekend dat de data op verschillende plekken opgeslagen kan worden om later *hergebruikt* te worden.
- Het is een gelaagd systeem. Deze hiërarchie is natuurlijk terug te vinden in de verdeling van frontend/backend/database, die we eerder al hebben genoemd, maar binnen de backend zien we ook een hiërarchische verdeling in de presentation layer (controller), business layer (service) en de persistence layer (repository).
- Een REST applicatie heeft een uniforme interface (API). Dit kun je bereiken door rekening te houden met de volgende 5 principes:
 - Gebruik HTTP-methoden zoals GET, POST, PUT, DELETE en PATCH. Deze methoden hebben elk hun specifieke functie en geven een stukje context over het verzoek dat wordt gemaakt. Door deze methoden correct te gebruiken, kunnen API-gebruikers gemakkelijker voorspellen hoe ze met de API moeten interacteren en kunnen API-ontwikkelaars gemakkelijker de API implementeren en onderhouden.
 - Gebruik een uniforme URI-structuur: Zorg ervoor dat de URI's van je API eenvoudig, consistent en gemakkelijk te begrijpen zijn voor API-gebruikers. Gebruik bijvoorbeeld duidelijke namen voor resources en gebruik placeholders voor parameters.
 - Gebruik een uniforme datastructuur: Het is belangrijk om een uniforme datastructuur te gebruiken om ervoor te zorgen dat de API-gebruikers gemakkelijk de data kunnen begrijpen en verwerken. Je kunt hiervoor XML of JSON gebruiken, maar moderne applicaties gebruiken hoofdzakelijk JSON.
 - Hanteer consistentie in foutafhandeling: Zorg ervoor dat de API-foutcodes en foutmeldingen uniform zijn en goed gedocumenteerd zijn, zodat API-gebruikers gemakkelijk kunnen begrijpen wat er mis is gegaan en hoe ze het probleem kunnen oplossen. Zorg dat er betekenisvolle error-messages worden verstuurd.
 - Documenteer de API duidelijk: Het is belangrijk om de API goed te documenteren, zodat API-gebruikers een duidelijk beeld hebben van de functionaliteit van de API, de parameters en de mogelijke antwoorden.
- (Optioneel) Een REST applicatie geeft ook de mogelijkheid om code op te sturen die uitgevoerd kan worden of gedownload kan worden. Deze functionaliteit is niet verplicht voor REST en wordt verder ook niet behandeld in deze cursus.

Wanneer we aan al deze voorwaarden voldoen, kunnen we zeggen dat onze applicatie **"RESTful"** is.

2.4 Communicatie in REST

Communicatie tussen de frontend en de backend van een REST applicatie verloopt via HTTP requests. HTTP (**H**ypertext **T**ransfer **P**rotocol) is een algemeen protocol voor communicatie op internet en is dus niet specifiek voor REST bedoeld. Het heeft een aantal handige tools waar we in REST dankbaar gebruik van kunnen maken, namelijk:

- **HTTP-methodes:** Eerder al hebben we resources genoemd als de zelfstandige naamwoorden van een REST applicatie. HTTP-methodes kun je zien als de werkwoorden van een REST applicatie. REST maakt gebruik van de HTTP-methoden om aan te geven welke bewerking op een resource moet worden uitgevoerd. Er zijn verschillende methoden beschikbaar in HTTP, zoals GET, POST, PUT, DELETE en PATCH. Deze methodes staan ook wel bekend als CRUD (Copy, Read, Update, Delete), hoewel dat eigenlijk slaat op de bewerkingen die je kunt doen op een database. Uiteindelijk is dat ook wel ons doel met een HTTP-request, om iets te doen in de database, maar het is wel belangrijk om te realiseren dat een HTTP-methode geen database-methode is.
- **URI's (Uniform Resource Identifiers):** URI's zijn de grammatica van een REST applicatie. URI's worden opgebouwd uit de verschillende elementen, waaronder een HTTP-methode en de naam van een resource, om de juiste resource te kunnen bereiken. URI's worden gebruikt om de resources te identificeren. Elke resource in de REST-API heeft een unieke URI die deze identificeert. De URI kan worden gebruikt om de resource op te halen, te updaten of te verwijderen.
- **HTTP-Statuscodes:** HTTP-statuscodes worden gebruikt om de status van de operatie aan te geven. Er zijn verschillende statuscodes beschikbaar in HTTP, zoals 200 OK, 201 Created, 404 Not Found, enzovoort. Deze statuscodes worden gebruikt om de API-gebruiker te informeren over het resultaat van de operatie en worden meegegeven als return waarde aan een request. Wanneer de statuscode niet expliciet gezet wordt, vult Spring het in als 200 OK.
- **Headers:** HTTP-headers worden gebruikt om extra informatie te verstrekken over de request of response. Ze kunnen worden gebruikt om verschillende dingen te specificeren, zoals de geaccepteerde content-types, de gebruikte encoding, de lengte van de response enzovoort.

2.5 Een voorbeeld

Een RESTful webservice bestaat dus uit een combinatie van HTTP methodes, de HTTP request URL en de HTTP status code.

Stel dat een webservice is geïmplementeerd op een server met het adres <http://mijnwebservice.nl>. Via deze webservice is de collectie books beschikbaar. Door de volgende endpoints aan het adres toe te voegen en de juiste HTTP methode te gebruiken kunnen de volgende acties worden uitgevoerd:

- **GET /books** om alle book items in de collectie op te vragen
- **GET /books/2323** om één specifiek boek (met id 2323) op te vragen.
- **POST /books** om een boek toe te voegen aan de collectie. De gegevens worden dan meegegeven in de *body* van het request.
- **DELETE /books/2323** om een specifiek boek uit de collectie te verwijderen
- **PUT /books/2323** om de gegevens van een specifiek boek aan te passen. De gegevens worden dan meegegeven in de *body* van het request.

De PATCH methode wordt ook wel gebruikt in een RESTful webservice om aan te geven dat een item moet worden bijgewerkt. Dit is vergelijkbaar met de PUT methode. Het verschil is dat een PUT altijd het gehele item update en een PATCH slechts een gedeeltelijke update uitvoert.

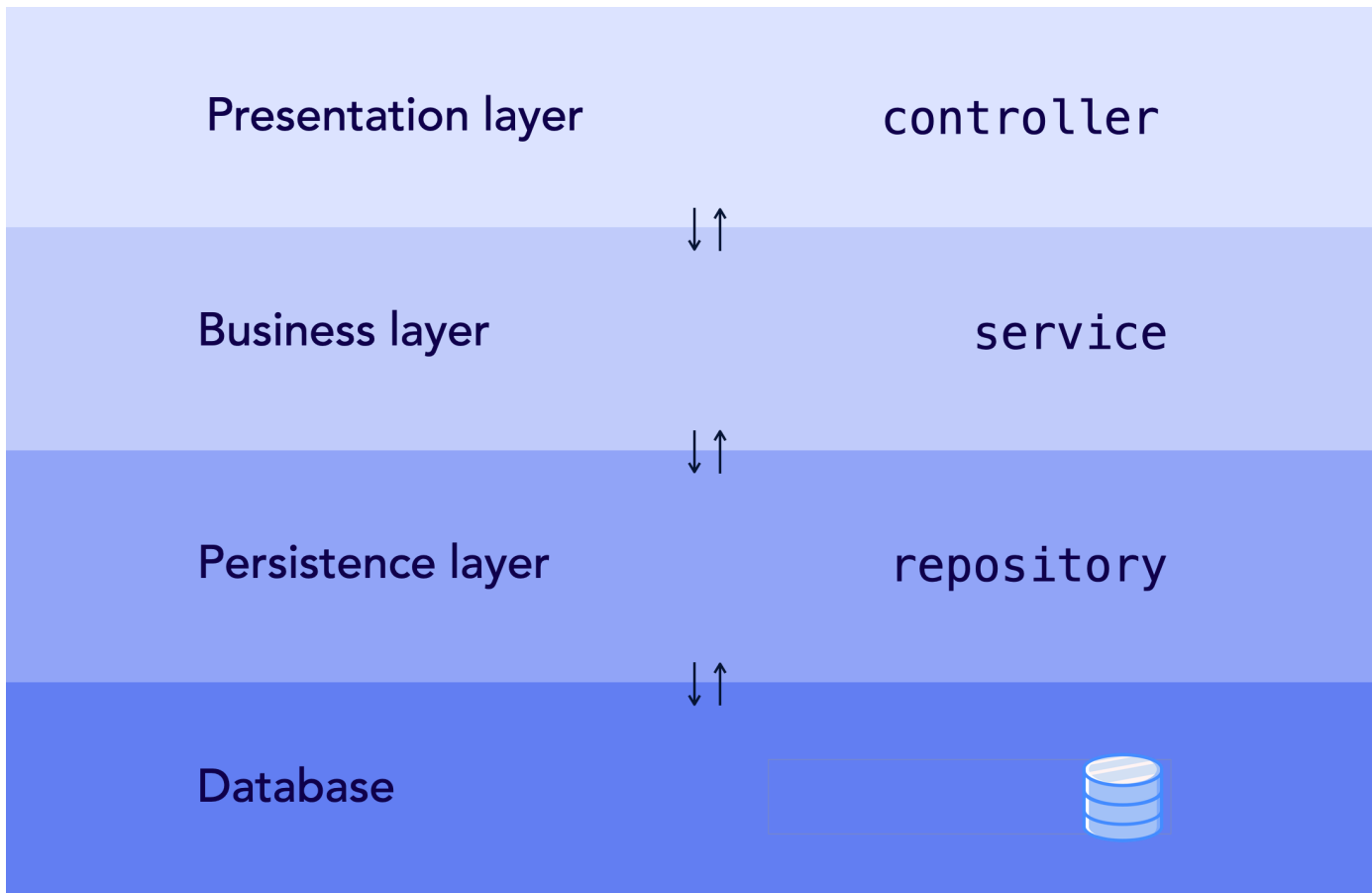
2.6 Zelftest

3. Controller

In dit hoofdstuk beginnen we met het echte bouwen van een webservice. We beginnen met de controllers. Hierin worden de verzoeken die vanuit de client binnen komen, ontvangen en gedelegeerd naar andere stukken van het programma om verwerkt te worden. Ook het antwoord dat terug wordt gestuurd, wordt vanuit de controller verzonden.

3.1 Inleiding

In dit hoofdstuk beginnen we met het maken van een echte webservice. Je herinnert je misschien dit plaatje nog wel, over de hiërarchische structuur van Spring:



We beginnen met de bovenste laag van de hiërarchie, de Controller. De Controller kun je zien als de "voordeur" van onze resource. Omdat een applicatie altijd meerdere soorten resources bevat, hebben we hier ook meerdere, aparte Controllers voor.

De Controller is de laag in de applicatie waar we de HTTP-requests ontvangen. Het is ook de laag vanuit waar we de response uiteindelijk terugsturen. Voor de logica die nodig is om de resource op te vragen, of de database te bewerken, zullen de details van het request worden doorgestuurd naar de service-laag (en vervolgens de repository- en database-laag). Voor nu bespreken we eerst de Controller in isolatie.

Dependency

Om een eigen Spring Boot project op te zetten, hebben we in het vorige hoofdstuk al gezien hoe we daar de Spring Initializr voor kunnen gebruiken. Daar hebben we onder het kopje "dependencies" de "Spring Web" dependency geselecteerd. In ons `pom.xml` bestand zie je die ook terug:

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

Deze dependency hebben we nodig om een Controller kunnen maken. Dit geeft ons namelijk toegang tot verschillende annotaties en classes die we nodig hebben.

3.2 Annotaties

In Java gebruiken we **annotaties** om metadata toe te voegen aan onze code. Deze extra informatie is geen onderdeel van het programmeer zelf, want een annotatie doet in feite niets. Het voegt informatie toe aan een class of methode die door een ander proces gebruikt zou kunnen worden (bijvoorbeeld tijdens het compileren). Zo kun je bijvoorbeeld aangeven dat een specifieke methode behandeld moet worden als een test-methode door de testing-library, of dat een class gezien moet worden als een Controller-class door Spring.

Annotaties zijn een **language feature** van Java: functionaliteiten van de taal die je er gratis bij krijgt. Waarschijnlijk vraag je je af waarom we annotaties niet direct in de cursus Java behandeld hebben. De reden daarvoor is dat annotaties in *plain* Java weinig gebruikt worden. Pas wanneer je tools en libraries gaat toevoegen - zoals Spring! - worden ze interessant. Spring maakt namelijk ontzettend veel gebruik van annotaties.

Je herkent een annotatie aan het `@`-symbool, gevolgd door de naam van de annotatie:

```
@Test
public static void specialMethod() {
}
```

Deze annotatie herken je vast! De `@Test`-annotatie geeft aan dat de methode die daarna volgt gebruikt wordt om een stukje van onze code te testen. Zo heb je er echter nog veel meer. Een simpele notatie heeft alleen een naam, maar je kunt annotaties ook **parameters** meegeven die in meer detail beschrijven wat deze class of methode moet voorstellen.

Wanneer we een Spring project hebben gemaakt via Spring Initializr, dan kunnen we daar in de `src-map` een nieuwe map maken die we `controller` noemen. Daarin stoppen we een nieuwe class, zoals bijvoorbeeld `SchoolNameController`. Onderstaande code staat in deze `SchoolNameController`-class. Deze regels code werken in een Spring Boot applicatie als een volledig werkende (Rest)Controller. Dit betekent dat als we deze class toevoegen aan een Spring Boot project, we de URL van het project in een webbrowser kunnen typen en deze controller kunnen aanspreken via de URL: `http://localhost:8080/name`.

```
// SchoolNameController.java
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SchoolNameController {

    @GetMapping(value = "/name")
    public String getName() {
        return "NOVI";
    }
}
```

De reden dat deze Java-class de string "NOVI" teruggeeft wanneer je `http://localhost:8080/name` bezoekt, is volledig te danken aan de annotations die hier gebruikt zijn. Je hebt inmiddels geleerd dat een Controller in Spring een Java-class bereikbaar maakt over het internet (HTTP). Je mag er dus vanuit gaan dat Spring als framework in staat is om HTTP-requests af te handelen. Wanneer Spring ziet dat een class geannoteerd is met `@RestController`, neemt het daardoor aan dat er methoden in deze class zitten die *gemapped* moeten worden (beschikbaar moeten worden gemaakt) op HTTP-uri's. Spring ziet vervolgens dat de methode in deze class is geannoteerd met `@GetMapping(value = "/name")`. Nu heeft Spring alle informatie om zichzelf te configureren. Als er een request binnenkomt op `http://localhost:8080/name` roept Spring de `SchoolNameController` aan die op diens beurt de `getName()`-methode aanroept. Wij hebben geprogrammeerd dat deze methode een String teruggeeft, maar om deze te kunnen versturen over het internet zet Spring dit voor ons om naar platte tekst: JSON (of XML), want HTTP kent immers geen String, dat is iets van Java.

Op deze manier kunnen wij onze Java-classes schrijven zoals we gewend zijn, maar regelt Spring het ontvangen en afhandelen van dataverzoeken voor ons. De annotations die we daarbij gebruiken, helpen Spring dus bij het toevoegen van functionaliteit op de juiste plek. Verget je de annotatie te gebruiken? Dan werkt de Controller niet meer.

Side note: Er zijn nog meer mogelijkheden om de endpoints te configureren binnen Spring waar we later op ingaan. Het is bijvoorbeeld mogelijk om onder `@RestController` een tweede Annotatie te zetten, namelijk `@RequestMapping("test")`. Dit zou Spring vertellen dat de base url voor deze class `http://localhost:8080/test` is. Dit zorgt ervoor dat onze url dus `http://localhost:8080/test/some` zou worden, erg handig als alle endpoints die je binnen de controller definieert onder `/test` vallen. Door *convention over configuration* hoeven we deze annotatie niet op te geven als we geen aangepaste base url willen.

RestController

We hebben inmiddels geleerd dat we boven een Controller-class altijd de annotatie `@RestController` moeten gebruiken. Deze annotatie zorgt er voor dat Spring bij het opstarten deze class inleest en in de Spring-context (dit behandelen we later) zet. Wanneer vervolgens een HTTP-Request ingezonden wordt vanuit het web, weet Spring dat er één of meer REST-controllers in de context staan die gebruikt kunnen worden om dat HTTP-request op te vangen. Dit regelt Spring allemaal zelf, het enige wat je dus hoeft te doen is je Controller klasse te beginnen met `@RestController`.

Endpoint

In het voorbeeld zagen we dat er een value-attribuut werd meegegeven aan de `@GetMapping` annotatie:

```
@GetMapping(value = "/name")
public String getName() {
    return "NOVI";
}
```

Wanneer we een URL bezoeken, krijgen we een website. Wanneer we een URI aanspreken krijgen we een resource.

Deze value specificeert naar welke specifieke URI we ons HTTP-request moeten versturen om deze specifieke Controller-methode aan te kunnen spreken. In ons geval was dat `/name`, dus wordt de volledige URI `localhost:8080/name`. Het host-nummer heeft Spring voor ons automatisch ingesteld. Localhost staat namelijk voor het IP-adres `127.0.0.1`, wat leidt naar je eigen computer. Het hostnummer, `8080`, is de specifieke poort op dat IP-adres. Spring kiest automatisch port `8080`, maar je zou ook een andere port kunnen kiezen!

Met behulp van het value-attribuut kun je dus meerdere methodes in een Controller zetten die allemaal beschikbaar zijn op een ander endpoint.

* In de map `main/resources` het bestand `application.properties` vinden. Hierin kun je het portnummer veranderen in bijvoorbeeld `8081` door `server.port=8081` aan te passen.

3.3 Mapping

Met een RESTful webservice willen we eigenlijk alle CRUD-operaties op de resource kunnen uitvoeren. Dat kan door gebruik te maken met de verschillende HTTP- methodes in combinatie met verschillende endpoints. Voor een bibliotheek applicatie zou dat er als volgt uit kunnen zien:

Http Method	Endpoint	Action	Annotation
GET	/books	Read many	@GetMapping("/books")
POST	/books	Create	@PostMapping("/books")
GET	/books/{id}	Read one	@GetMapping("/books/{id}")
PUT	/books/{id}	Update	@PutMapping("/books/{id}")
PATCH	/books/{id}	Partial update	@PatchMapping("/books/{id}")
DELETE	/books/{id}	Delete	@DeleteMapping("/books/{id}")

Je ziet hier dat je voor de meest voorkomende HTTP-methodes speciale annotations hebt.

GET-request

Een GET-request mag nooit een verandering veroorzaken in de resource en wordt daarom ook wel veilig genoemd. Bij het doen van een GET-request wordt er dus data opgevraagd bij de server. De server gaat in de database opzoek naar de data en stuurt dit terug in een HTTP-response.

Het return-type van de methode die alle boeken opvraagt, verschilt natuurlijk per implementatie. In ons geval zouden we een String, een `List<Book>` of een `ResponseEntity<Object>` kunnen teruggeven. We gebruiken de laatste optie, omdat daar de meeste mogelijkheden mee zijn. Spring Boot vertaalt de return-value van de methode naar JSON en plaatst dit in de body van de response. We kunnen immers alleen JSON of XML over het internet verzenden. Dit zou er zo uitzien:

```
@RestController
public class BooksController {
    @GetMapping("/book")
    public ResponseEntity<String> getBook() {
        return ResponseEntity.ok("Novi");
    }
}
```

Met behulp van de library Jackson wordt de return-value omgezet naar JSON. Naast de data die erin staat, wordt iedere response voorzien van een status code die laat zien of het request succesvol is afgehandeld. Bij een GET-request wordt er een 200 OK status code teruggegeven als de operatie is gelukt.

Afhankelijk van de reden waarom een request niet is geslaagd, kan een andere status code worden teruggegeven. Hierover later meer.

POST-request

Het POST-request wordt gebruikt om een item toe te voegen aan de collectie. Denk bijvoorbeeld aan het toevoegen van een boek aan de bibliotheek of het registreren van een nieuw gebruikersaccount bij een social media platform. Hiervoor wordt er dus data *verstuurd* naar de server. De server voegt het item toe, maar stuurt geen data terug, zoals bij een GET-request. Wel wordt er in de response-header een endpoint meegegeven waarmee het nieuwe item weer kan worden opgehaald. Deze kun je vinden onder de veldnaam "location". De status code die wordt teruggegeven als het toevoegen van het item is gelukt, is een 201 Created code.

```
@PostMapping("/book")
public ResponseEntity<Object> addBook(@RequestBody String title) {
    // (hier wordt iets opgeslagen in de database)...
    String name = "NOVI";
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{name}")
        .buildAndExpand(name).toUri();
    return ResponseEntity.created(location).build();
}
```



```
}
```

DELETE-request

Een DELETE-request geeft aan dat het item uit de collectie moet worden verwijderd. Bij het doen van een DELETE-request wordt er geen data meegestuurd naar de server. Via het endpoint, waarin bijvoorbeeld de ID van het item wordt gespecificeerd, wordt al duidelijk welke item er precies verwijderd moet worden.

```
@DeleteMapping("/books/{id}")
public ResponseEntity<Object> deleteBook(@PathVariable int id) {
    ..(hier wordt een naam verwijderd uit de database)...
    return ResponseEntity.noContent();
}
```

PUT-request

Een PUT-request geeft aan dat het item uit de collectie moet worden aangepast of vervangen. De gegevens van het nieuwe item worden, net als bij een POST-request, verstuurd naar de server. De server wijzigt daarmee het item en stuurt een status-code terug. Bij success is dat standaard 204 No Content.

```
@PutMapping("books/{id}")
public ResponseEntity<Object> updateBook(@PathVariable int id, @RequestBody String bookTitle) {
    ..(hier wordt een resource aangepast)
    return ResponseEntity.noContent();
}
```

3.4 Request parameters and responses

We hebben drie verschillende methoden om input mee te geven aan een request:

- Met behulp van een **PathVariable**. Het komt vaak genoeg voor dat we slechts één boek of één blogpost willen ophalen in plaats van alle items uit de collectie. Zo'n item heeft altijd een *id*: een sleutelveld die uniek is voor het item binnen de collectie van items in de resource. Vaak is dit een nummer, maar het kan ook een String zijn. In de request-mapping wordt zo'n id aangegeven als {id} om aan te geven dat deze vervangen kan worden door de letterlijke waarde van de ID. Een GET-request dat gebruik maakt van een pathVariable zal er dan zo uit zien:

```
@GetMapping("/books/{id}")
public ResponseEntity<Object> getBook(@PathVariable long id) {
    return ResponseEntity.ok(...);
}
```

- RequestParam**: wanneer we niet precies weten naar welke ID's we op zoek zijn, kan het zijn dat we een zoekcriterium mee willen geven. We willen niet alle items uit de collectie, maar alleen die items die voldoen aan een opgegeven criterium. Hiervoor gebruik je een **query string**: een string die je aan het eind van het endpoint mag toevoegen door er een ? aan toe te voegen. Denk aan /books?title=harry of /books?writer=rowling. Je mag ze zelfs aan elkaar plakken door het & teken te gebruiken: /books?title=harry&writer=rowling. In de Controller-methode kunnen deze extra velden worden afgevangen met de @RequestParam annotate. De Controller-methode krijgt argumenten mee die qua naamgeving overeen komen met de velden in de querystring:

```
@GetMapping("/books")
public ResponseEntity<Object> getAllBooks(@RequestParam String title) {
    return ...;
}
```

- RequestBody**: dit wordt gebruikt voor POST-requests en PUT-requests om daadwerkelijk data mee te sturen. Deze annotate stelt je dus in staat om complete objecten mee te sturen in een HTTP request. Deze meegestuurde objecten kun je vervolgens gebruiken om op te slaan, of om bestaande data aan te passen.

Responses voor bad requests

Tot nu toe zijn we ervan uit gegaan dat de RESTful request is geslaagd en dat er een 2xx status code wordt teruggegeven. Echter, het kan ook gebeuren dat een request niet slaagt. Er is bijvoorbeeld een niet bestaande ID meegegeven, of je bent als gebruiker niet geautoriseerd om de operatie uit te voeren. In die gevallen wordt een 4xx status code teruggegeven.

Dit verzorgen we in de backend doormiddel van **exceptions**. Het is dan wenselijk eigen exceptions klassen te definiëren. Als zich een onwenselijke situatie voordoet, kan een exception worden gegenereerd. Spring Boot heeft een annotatie @ControllerAdvice waarmee een speciale Exception-Controller kan worden aangegeven. Een exception eindigt dan in een controller-methode die als response de juiste status code teruggeeft.

```
public class RecordNotFoundException extends RuntimeException {
    private static final long serialVersionUID = 1L;
    public RecordNotFoundException() {
        super();
    }
    public RecordNotFoundException(String message) {
        super(message);
    }
}
```

```
@ControllerAdvice
public class ExceptionController {
    @ExceptionHandler(value = RecordNotFoundException.class)
    public ResponseEntity<Object> exception(RecordNotFoundException exception) {
        return new ResponseEntity<>(exception.getMessage(), HttpStatus.NOT_FOUND);
    }
}
```

```
if (...) { // determine if id exists
    throw new RecordNotFoundException('ID cannot be found');
}
```

4. Domain Models

Dit hoofdstuk gaat over het modelleren van het domein. Daarmee bedoelen we de entiteiten die van belang zijn voor de applicatie die u wilt ontwikkelen. In het voorbeeld webservice, de bibliotheek, gaat het domein bijvoorbeeld over boeken, schrijvers, exemplaren van boeken en leden. In Java worden deze domeinentiteiten vertegenwoordigd door klassen. We nemen een uitstapje naar object georiënteerd programmeren en het opstellen van een UML klassendiagram, waar de domeinmodellen vertegenwoordigd worden door een domein model of domein klasse. Tenslotte zul je leren hoe je de domeinmodellen klaar kunt maken voor het werken met een database, waar de domeinmodellen door tabellen en records (rijen) worden vertegenwoordigd. De verschillende termen voor domeinentiteiten worden vaak door elkaar gebruikt.

4.1 Inleiding

Het **domein** is de verzameling van alle modellen die van belang zijn voor het ontwikkelen van je applicatie. Deze modellen kunnen met verschillende namen benoemd worden, maar het komt allemaal op hetzelfde neer. In de voorbeeld webservice, de bibliotheek, bestaat het domein bijvoorbeeld uit boeken, schrijvers, exemplaren van boeken en leden. In Java worden deze domeinmodellen vertegenwoordigd door klassen. We nemen een uitstapje naar object georiënteerd programmeren en het opstellen van een UML-klassendiagram, waar de domeinmodellen vertegenwoordigd worden door een domein model of domein klasse. Tenslotte zul je leren hoe je de domeinmodellen klaar kunt maken voor het werken met een database, waar de domeinmodellen door tabellen en records (rijen) worden vertegenwoordigd. De verschillende termen voor domeinentiteiten worden vaak door elkaar gebruikt.

4.2 Domeinmodel

Een applicatie heeft altijd betrekking op een bepaald domein. De voorbeeldapplicatie heeft als domein een bibliotheek, maar een domein zou ook een bank of garagebedrijf kunnen zijn. Het domein beschrijft dus eigenlijk waar de applicatie over gaat.

De kern van een applicatie zijn dan de klassen die te maken hebben met het domein van de applicatie. Bij een bibliotheek zijn dat bijvoorbeeld boeken, leden en geleende boeken. Het vinden van de benodigde domeinklassen voor een applicatie is essentieel voor een goede opbouw van de applicatie. Om de juiste klassen te vinden voor een domein, kunnen naar de zelfstandige naamwoorden kijken. Een klant weet wat hij wil en schrijft zijn wensen op. Voor onze bibliotheek applicatie zouden we onderstaande tekst kunnen hebben gekregen:

—
Onze bibliotheek is al 100 jaar oud en we doen al 100 jaar onze administratie op dezelfde manier. Het nieuwe management wil echter met de tijd mee gaan en daarom willen we een digitale administratie.
We willen bij kunnen houden welke boeken we in onze voorraad hebben en welke boeken zijn uitgeleend. We willen ook weten hoeveel exemplaren we van elk boek op voorraad hebben.
Van een boek willen we in ieder geval de titel, auteur en schrijver weten.

We willen bij houden van welke auteurs we boeken op voorraad hebben. We willen van die auteurs ook een korte biografie kunnen geven.
Ook willen we onze leden administratie in diezelfde applicatie kunnen regelen. We willen van leden kunnen bij houden welke boeken ze hebben, hun NAW-gegevens en of ze nog openstaande boetes hebben.

Probeer hier zelf maar eens de domein modellen uit deze tekst te halen aan de hand van de zelfstandige naamwoorden. Vind je dezelfde modellen als in de inleiding staan, of misschien nog meer?

Note: Let erop dat je uiteindelijk altijd de enkelvoudsvorm van een zelfstandig naamwoord gebruikt voor je domein model, dus niet "bananen", maar "banaan". Ook gebruiken we volgens conventie Engelstalige klasse namen in onze code, dus "banaan" zal in je code "banaan" heten. De software development branche is internationaal van aard, dus grote kans dat een niet-nederlands-spreekende developer uiteindelijk aan jou code zal komen te werken en die moet ook kunnen begrijpen wat daar staat. In communicatie met een klant die uitsluitend Nederlands spreekt, is het wel goed om de communicatie, zoals een bespreking over het domein, in het Nederlands te doen. De code blijft wel, volgens conventie, in het Engels.

4.3 Klassendiagram

Domein modellen geven we vaak weer in een **klassendiagram**. Klassen diagrammen zijn een superhandig communicatie-hulpmiddel om enerzijds met collega-developers te discussiëren over de code en het design daarvan en anderzijds om aan de klant te laten zien hoe wij (de developers) diens idee hebben vormgegeven/gaan vormgeven. Uiteraard gaan we hier veel verder op in in de cursus Software Design en Documentatie.

4.4 Modelklasse

Wanneer we de domein-klassen duidelijk hebben, kunnen we dit vertalen naar model-klassen. Een model klasse, ook wel **entiteit** genoemd, is de data-klasse waarin we de data van ons domein model kwijt kunnen. Dit klinkt allemaal wat abstract, laten we daarom maar beginnen door te kijken hoe dit er in Java uit ziet. In ons project kunnen we onder `src/main/java/nl\novi\bibliotheek` een nieuwe package aanmaken genaamd "models". In deze models-package kunnen we al onze modellen zetten, maar voor nu maken we een nieuwe Class genaamd Book.

Note: Kijk voor de juiste mappen-structuur ook nog eens naar het hoofdstuk over Build Tooling in de cursus Programmeren met Java.

```
// src/main/java/nl/novi/bibliotheekApplicatie/models/Book.java
@Entity
@Table(name = "books")
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String isbn;
    @Column(name = "title")
    private String mainTitle;
    private String genre;
    public Long getId(){
        return id;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public String getGenre() {
        return genre;
    }
    public void setId(Long id){
        this.id = id;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    }

    public void setSubtitle(String subtitle) {
        this.subtitle = subtitle;
    }
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }
    }
}
```

Het eerste wat opvalt is dat er weer een nieuwe annotatie boven de klasse staat, namelijk `@Entity`. Deze annotatie is echter iets anders dan de `@RestController` en vergelijkbare annotaties (zogenaamde "stereotype annotaties", maar dat wordt later uitgelegd). De `@Entity` annotatie is namelijk niet van Spring, maar van Hibernate. **Hibernate** is een stukje software dat in Spring geïntegreerd zit, om Java en Spring code te vertalen naar de database taal SQL.

Wat de `@Entity` annotatie doet is dat het zorgt dat het een **persistance object** wordt, een object dat *ge-persist* kan worden in de database. **Persisten** betekend ookwel het opslaan en behouden van data. Hiermee willen we dus eigenlijk zeggen deze klasse vertaald kan worden naar een tabel in de database. De velden in deze klassen worden de kolommen van de tabel. Hoewel deze Book-klasse dus heel simpel is (een POJO*), is het wel heel belangrijk en kan het met annotaties zelfs erg gecompliceerd worden.

Naast de `@Entity` annotatie, staat er ook een `@Table` annotatie boven de klasse. Deze annotatie doet op zich niks, immers zegt de `@Entity` annotatie al de we van deze klasse een tabel willen maken, maar deze annotatie accepteert wel een aantal parameters waarmee we de specificaties van de tabel kunnen aanpassen. Eén van die parameters is `name`. Hiermee kunnen we de naam van de tabel aanpassen. Wanneer we hiervoor niks invullen, wordt de naam de klasse gebruikt. Het is echter conventie om tabellen een meervoudsnaam te geven en dat doen we bij klassen juist niet.

Een van die annotaties is de `@Id` annotatie. Dit is de belangrijkste annotatie en ook de enige die verplicht is. Wanneer je een klasse annoteert met `@Entity` en je definieert er geen `@Id` bij, dan zal IntelliJ daar een foutmelding over geven. De `@Id` annotatie maakt dat dit veld de *primary key* van de tabel wordt. Wanneer we het straks over relaties gaan hebben, zal dit dus ook de *foreign key* van die relatie worden. De `@Id` annotatie heeft, net als de *primary key* van een tabel, twee belangrijke regels: het moet uniek zijn en het mag niet `null` zijn (je moet het invullen).

De `@GeneratedValue` annotatie zorgt er voor dat Hibernate de waarde van ons id zelf bepaald én verhoogd. Zoals je net al gelezen hebt, moet het id uniek zijn en mag het niet `null` zijn. De `@GeneratedValue` annotatie helpt je daarbij. Wanneer je als id een incrementele waarde gebruikt, zoals een Integer, een Long of een UUID, kan hibernate ervoor zorgen dat bij elke nieuwe entiteit die opgeslagen wordt in de database, er automatische een unieke id wordt ingevuld. De eerste entiteit zou bijvoorbeeld id=1 krijgen, de tweede entiteit id=2, dan id=3, id=4, id=5, etc. De `@GeneratedValue` annotatie accepteert een aantal parameters, zoals een "strategy" parameter waarin je de GenerationType kunt zetten:

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

`GenerationType.IDENTITY` zorgt er voor dat hibernate simpelweg het id-veld vult met een steeds oplopende waarde. Dit kan gebruikt worden voor alle numerieke waardes en UUID's.

`GenerationType.SEQUENCE` is een paar milliseconden trager dan `IDENTITY`, maar kan wel gebruikt worden voor databases die `IDENTITY` niet ondersteunen, zoals Oracle 11g. Met `GenerationType.SEQUENCE` kun je ook een eigen sequence generator definiëren, maar dat valt buiten de scope van deze cursus.

Wanneer we niks invullen, default het naar `GenerationType.AUTO` en wordt de meest toepasselijke variant gekozen.

Waar we de `@Table` annotatie konden gebruiken om de specificaties van de tabel aan te passen, kunnen we de `@Column` annotatie gebruiken om de specificaties van de kolom aan te passen. Deze annotatie doet op zich niets, maar door de parameters in te vullen kunnen we de kolom aanpassen. Zo kunnen we de `name`-parameter gebruiken om de naam van de kolom aan te passen. Het veld "mainTitle" zou in de tabel "main_title" heten, maar omdat we in de `@Column` annotatie de naam hebben aangepast, zal deze kolom nu "title" heten. De `@Column` annotatie heeft nog een aantal andere nuttige attributen, zoals "nullable" en "unique" (deze twee vormen samen de `@Id` annotatie). Je kunt zelf kijken welke attributen er nog meer in deze annotatie zitten door er met met de CTRL-toets ingedrukt op te klikken.

Note: Hibernate vertaalt de klassen en velden voor ons naar tabellen en kolommen, maar dat doet Hibernate met SQL. Een belangrijke beperking van SQL is dat het geen hoofdletters kent. Wanneer we in java dus (volgens conventie) namen in camelCase schrijven, kan dit niet vertaald worden naar SQL. Hibernate heeft hiervoor als oplossing om het dan maar naar snake_case te vertalen. Waar je bij camelCase woorden aan elkaar kunt plakken door elk nieuw woord met een hoofdletter te beginnen, kun je met snake_case woorden aan elkaar plakken door elk nieuw woord met een laag streepje (_) te beginnen.

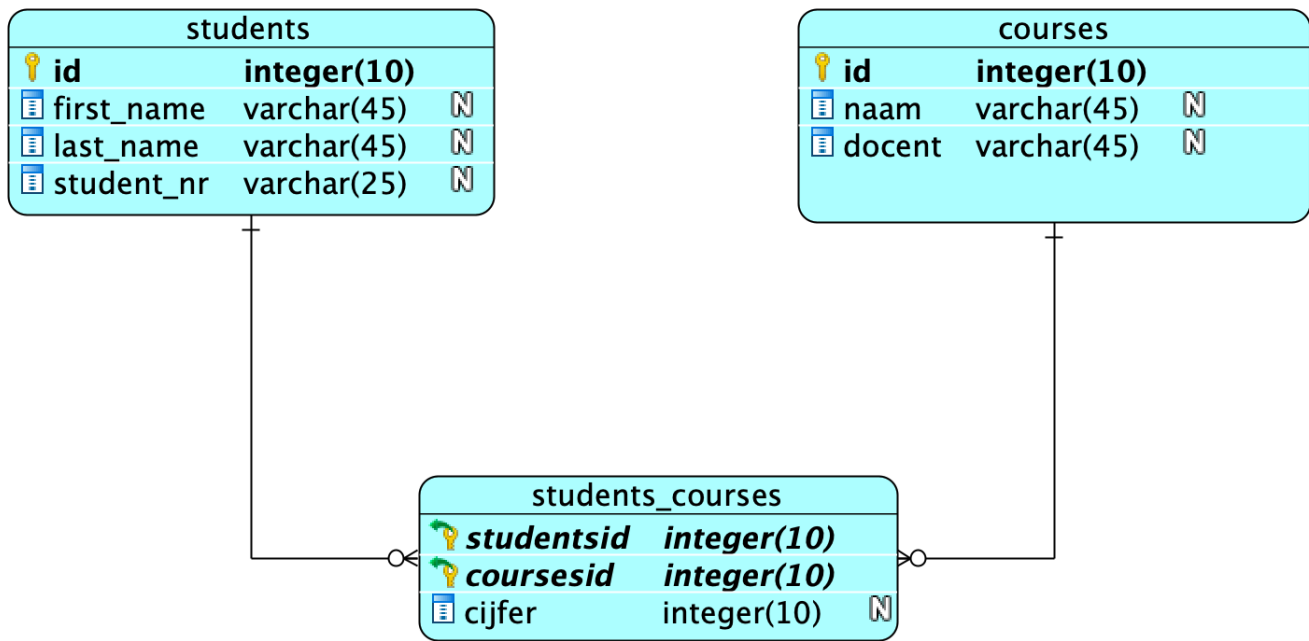
*Een POJO is een **Plain Old Java Object**. Dit zijn klassen, zoals entity-klassen, die enkel een paar velden hebben met getters, setters, constructors en eventueel ook een equals en hashCode methode om het af te maken. Deze objecten bevatten geen logica en worden voornamelijk als "data container" gebruikt.

4.5 Database

Een backend applicatie bestaat altijd uit een applicatie (de Java of Spring Boot code) en een database. Hierin zijn records beschikbaar die we via de webservice beschikbaar willen stellen aan de gebruiker. Er is natuurlijk een relatie tussen de objecten in de applicatie en de records in de database. De objecten worden opgehaald uit de database en nieuwe objecten worden aan de database toegevoegd.

En toch is er ook een verschil. In de database spreken we van entiteiten. Entiteiten zijn de tabellen die in een database zijn gecreëerd. Vanuit een database oogpunt zijn er regels waaraan de tabellen moeten voldoen. De tabellen bevatten velden met een primitieve type, zoals een getal, tekst of een datum. Ieder veld heeft een primair sleutelveld waarvan de waarde voor iedere record uniek is en die gebruikt wordt om één record specifiek aan te geven. Relaties tussen tabellen worden gerealiseerd door een vreemde sleutel die verwijst naar een primaire sleutel in een ander tabel.

Het klassendiagram die we eerder hebben besproken wordt dan vertaald in een entiteiten diagram (ER-diagram).



Hierin zijn de verschillende tabellen herkenbaar die min of meer vergelijkbaar zijn met de klassen in het klassendiagram. In een ER-diagram worden ook alle primaire sleutels en vreemde sleutels weergegeven.

Een belangrijk verschil tussen klassen en entiteiten is dat een attribuut in een klasse als een collectie (een List of Set) kan zijn gedefinieerd, wat veel voorkomt. In een entiteit kan dat niet en zal dit worden gesplitst in twee tabellen.

4.6 Hibernate

Om het omzetten van objecten naar entiteiten en van entiteiten weer terug naar objecten te realiseren maakt Spring Boot gebruik van een ORM: een Object Relational Mapping. De ORM die door Spring Boot wordt gebruikt is Hibernate.

Student Class in Object Model

```

@Entity
@Table(name = "students")
public class Student {

    @Id
    @GeneratedValue
    Long id;

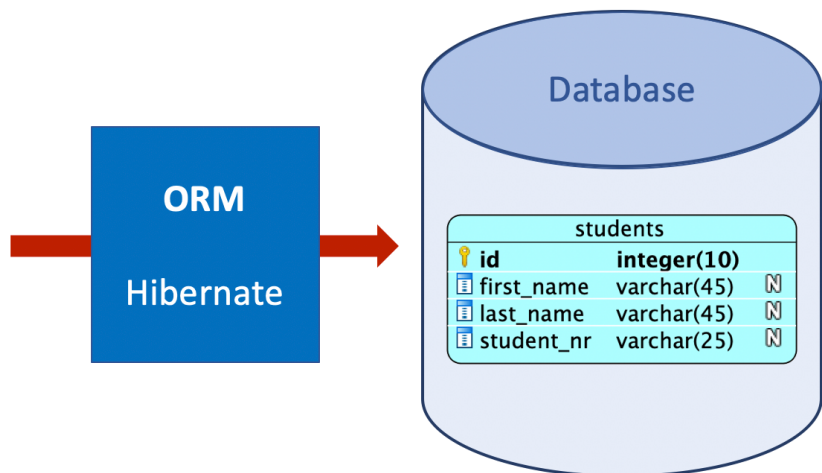
    @Column(name = "first_name")
    String firstName;

    @Column(name = "last_name")
    String lastName;

    @Column(name = "student_nr")
    String studentNr;

}
  
```

Student Table in Relational Model



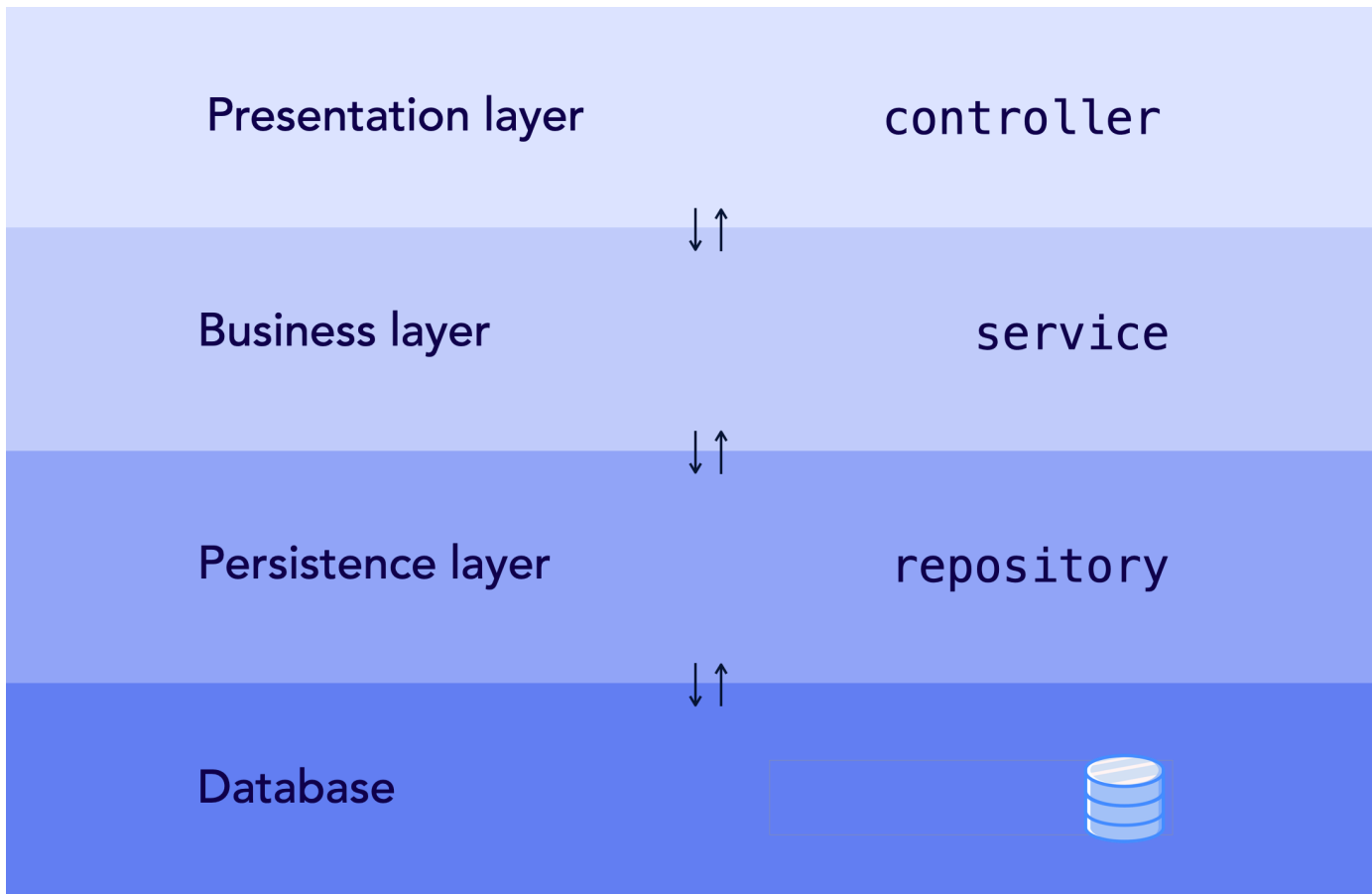
Hibernate kent aan de ene kant dus de Java klassen en objecten en aan de andere kant de tabellen van de database. Hibernate verzorgt de vertaalslag.

5. Repository

Dit hoofdstuk gaat over het werken met databases. Binnen Spring Boot worden databases benaderd via repositories waardoor directe communicatie met de database met behulp van SQL queries niet nodig is.

5.1 Inleiding

Inmiddels heb je geleerd hoe je Controllers kunt opzetten in Spring. In dit hoofdstuk ga je leren hoe je **repositories** kunt gebruiken: de repository is de schakel tussen de backend-applicatie en de daadwerkelijke database. Hierbij slaan we de Service nog even over, dit wordt later behandeld.



Om met de repository te kunnen werken, moeten we weer een aantal dependencies in de pom.xml toevoegen:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <scope>runtime</scope>
</dependency>
```

5.2 Repository

De Repository is niet erg ingewikkeld om te maken. Laten we als voorbeeld een simpele Repository maken voor ons Book-model uit het vorige hoofdstuk.

```
// nl/novi/BibliotheekApplicatie/repositories/BookRepository.java
public interface BookRepository extends JpaRepository<Book, Long> {
    Optional<Book> findByTitle(String Title);
}
```

Het eerste wat opvalt is misschien wel dat we hier een Interface maken in plaats van een class. Dit is inderdaad erg bijzonder, want we hoeven van deze interface nooit een implementatie te schrijven. Spring doet dat achter de schermen voor ons.

Het volgende wat we zien is dat we een andere interface *extend*en, in dit geval de JpaRepository. Dit is een interface met een aantal voor gedefinieerde functies, zoals `save(Entity entity)` om een entity in de database op te slaan en `findById(Long id)` om een entity te vinden met een specifiek id. Behalve de JpaRepository, zijn er nog andere opties om te *extend*en. De JpaRepository is de meest uitgebreide repository, maar PagingAndSortingRepository is de meest eenvoudige. Daartussen zit ook nog de CrudRepository die eigenlijk al voldoende functionaliteit heeft voor wat je in de eindopdracht wil kunnen.

Welke basis-repository we ook kiezen om te *extend*en, we moeten er altijd twee Generics aan meegeven. De eerste is het type van de entiteit. In ons geval is dat dus Book. De tweede Generic is het type *van het @Id veld* in die entiteit. Bij Book is het type van het @Id veld Long, dus vullen we hier Long in. Wanneer we naast de functies van de super interface nog andere functionaliteiten willen gebruiken, kunnen we die zelf definiëren. In het voorbeeld zien we al dat we een functie hebben gedefinieerd:

```
Optional<Book> findByTitle(String Title);
```

Deze functie haalt een boek op aan de hand van de titel van dat boek. Dit soort methodes kun je maken aan de hand van bepaalde keywords, zoals `findBy`, `existsBy`, `countBy`. Het tweede gedeelte van deze methode, `Title`, is de naam van een veld in de entiteit. Daarnaast kunnen we nog specificaties aan de zoekopdracht toevoegen, zoals `ignoreCase` om hoofdlettergevoeligheid uit te zetten of `containing` om te zoeken naar een boek die de String gedeeltelijk in de titel bevat. Wanneer we dan bijvoorbeeld naar "steen" zoeken, zou het boek "de steen der wijzen" als resultaat gegeven kunnen worden. Wil je weten welke keywords je nog meer allemaal kan gebruiken? Kijk dan eens in Appendix C van de [data-jpa documentatie](#).

Naast dat we met keywords methodes kunnen maken die dankzij Spring-magie werken, kunnen we ook methodes maken met de `@Query` annotatie. Aan deze annotatie kun je een SQL-commando meegeven die uitgevoerd wordt wanneer de methode aangeroepen wordt.

```
@Query("Select b from books b where title=:title and genre=:genre")
List<Book> BooksWithTitleAndGenre(@Param("title") String title, @Param("genre") String genre)
```

Deze methode is dus niet opgebouwd uit keywords, maar voert de SQL-query uit die erboven staat. Je ziet dat we met de `@Param` annotaties variabelen kunnen meegeven aan de SQL-query. Dit is dus een zeer krachtige tool om snel de juiste resultaten uit je database te kunnen halen.

Note: Wanneer we een query doen op niet-unieke zoek criteria, dan kunnen we meerdere resultaten en is het dus verstandig om een collectie te retourneren. Een veld in een entiteit is pas uniek wanneer je dat met de `@Column` annotatie aan geeft (behalve het @Id veld).

Het zal je vast al opgevallen zijn dat we vaak geen Book retourneren uit een repository methode, maar een `Optional<Book>`. Een `Optional` is een datatype van Java en is eigenlijk een container dat iets of niets kan bevatten, net zoals een `ArrayList` een container is dat heel veel dingen kan bevatten. Een `Optional` heeft, net als een `ArrayList`, een `Generic` nodig, ofwel, de diamantjes `<>`, waarin je aangeeft wat het type van de inhoud van de `Optional` is.

Een `Optional` heeft twee methodes waarmee we kunnen checken of er iets in zit of niet, namelijk `isPresent()` en `isEmpty()`. Wanneer we zeker hebben gesteld dat er werkelijk iets in de `Optional` zitten, kunnen we het er uit halen met de `get()` methode.

Je kunt een Optional ook als type van een parameter in een controller methode gebruiken, om die parameter optioneel te maken.

5.3 Application.properties

application.properties is een bestand dat je kunt vinden in de src/resources map. In dit bestand kun je regels toevoegen (properties) die de "convention of configuration"-instellingen van Spring overschrijven. Je hebt dit bestand misschien al eerder gebruikt om de regel server.port=8081 in te vullen, omdat jouw applicatie om een of andere reden aangaf dat poort 8080 bezet was.

Nu gaan we daar nog wat regels aan toevoegen. Als eerste de volgende twee regels om aan te geven welke database we gebruiken en wat het adres is waarop we die database kunnen bereiken. In ons geval heet de database "springboot". Zorg data deze naam ook overeenkomt met jou eigen database (Je hebt hier al een database voor gemaakt zoals beschreven in de cursus Databases)

```
spring.sql.init.platform=postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/springboot
```

Dan de volgende twee regels, om aan te geven wat de gebruikersnaam en password zijn om in je database in te loggen. Zorg dat dit overeenkomt met jou eigen database en zorg dat je password uniek is (en dus niet ook voor je email, facebook en bankgegevens wordt gebruikt). Het wachtwoord hoeft niet gecompliceerd te zijn voor deze cursus, zoiets als "geheim" is zelfs al voldoende.

```
spring.datasource.username=postgres
spring.datasource.password=geheim
```

Dan de volgende vier regels om aan te geven welke database we gebruiken. We gebruiken voor deze cursus standaard PostgreSQL, maar wanneer je H2 of MySQL gebruikt is dit dus anders.

```
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database=postgresql spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
```

Als laatste voegen we nog de volgende regels toe om aan te geven dat we willen dat Hibernate voor ons tabellen aanmaakt, dat die tabellen elke keer opnieuw gemaakt worden als we onze applicatie opstarten, dat we in de console te zien krijgen welke database calls worden uitgevoerd en dat we willen dat de data.sql ingeladen wordt.

```
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
```

Data.sql

Het is tijdens development handig om je database te vullen met test-data. Wanneer je bijvoorbeeld een GET-request wilt testen in postman, wil je niet eerst een POST-request hoeven doen. Je wilt meteen een GET-request kunnen doen die meteen data moet terug kunnen geven.

Gelukkig heeft Spring daar een heel makkelijk systeem voor, de data.sql. Dit bestand kun je aanmaken in de src/resources map. Je kunt daar simpelweg een nieuw file aanmaken en deze "data.sql" noemen.

Let op: wanneer we het bestand "data.sql" hebben aangemaakt en in de application.properties hebben aangegeven dat we dit willen gebruiken, dan mag het bestand niet leeg zijn. We moeten er dan SQL in zetten, want anders start de applicatie niet meer op!

Om gebruik te maken van data.sql moeten we wel zorgen dat we dit in de application.properties correct hebben ingesteld. Vervolgens kunnen we in de data.sql INSERT-statements (en andere SQL) zetten om onze database te vullen. Bijvoorbeeld:

```
INSERT INTO books (id, isbn, title, genre) VALUES (1, '1232143143', 'Novi', 'educatief'),(2,'563455423412','Koken met bananen','kookboeken');
```

5.4 Repository hierarchy

Zoals eerder genoemd hebben zijn er drie repositories die we kunnen extenden. De PagingAndSortingRepository heeft enkel de volgende twee methodes:

PagingAndSortingRepository extends CrudRepository

Iterable findAll(Sort sort)

Page findAll(Pageable pageable)

De CrudRepository extends de PagingAndSortingRepository en heeft dus naast het bovenstaande ook deze methodes:

CrudRepository

<S extends T> **save**(S entity)

<S extends T> Iterable<S> **saveAll**(Iterable<S> entities)

Optional<T> **findById**(ID id)

Iterable<T> **findAll**()

Iterable<T> **findAllById**(Iterable<ID> ids)

long **count**()

void **deleteById**(ID id)

void **delete**(T entity)

void **deleteAll**(Iterable<? extends T> entities)

void **deleteAll**()

De JpaRepository heeft al het bovenstaande en nog deze:

JpaRepository extends PagingAndSortingRepository

<S extends T> List<S> **saveAll**(Iterable<S> entities)

void **flush**()

<S extends T> S **saveAndFlush**(S entity)

void **deleteInBatch**(Iterable<T> entities)

void **deleteAllInBatch**()

T **getOne**(ID id)

<S extends T> List<S> **findAll**(Example<S> example)

<S extends T> List<S> **findAll**(Example<S> example, Sort sort)

5.5 Spring context en dependency injection

Om te zorgen dat spring überhaupt opstart, gebruiken we een annotatie. Deze heb je al gekregen toen je het project aanmaakte met Spring Initializr. Het gaat om de annotatie `@SpringBootApplication`. Deze staat boven jouw Main-class, de klasse die dezelfde naam heeft als je project. Deze annotatie zorgt voor drie dingen:

- De SpringBoot autoconfiguratie wordt ingesteld.
- Spring gaat een "ComponentScan" uitvoeren, waardoor geannoteerde klasse als onze `@RestController`-klasse worden gevonden in ingelezen.
- Spring gaat een configuratie scan uitvoeren waarin alle classes worden ingelezen (en uitgevoerd) die de `@Configuratie` annotatie hebben. Dit is dus handig als je eigen configuratie wilt inzetten, wat we later ook zullen doen voor de security

De component scan is waarom we in onze Controller gewoon een Repository kunnen zetten, zonder dat we het `new` keyword hoeven gebruiken. De `RestController` en de `@Repository` annotatie van je repository super-klasse maken van die klasse een **Bean**. Beans zijn instanties van classes die door ons dependency injection framework worden gemaakt en gemanaged. In ons geval spreken we van **Spring Managed Beans**. Om een **Bean** te maken, gebruiken we in de basis de `@Component` annotatie, maar we hebben al gezien dat er verschillende annotaties zijn met een ingebakken `@Component` annotatie. Wanneer je bijvoorbeeld dieper in de `@RestController` annotatie kijkt met CTRL+klik, zul je ook zien dat daar uiteindelijk een `@Component` annotatie in zit.

Om onze repository in onze controller te kunnen gebruiken, gebruiken we geen "new" keyword, maar laten we Spring de juiste Bean vinden door middel van dependency injection. Hoe werkt dependency injection?

Dependency injection is heel simpel. Wanneer we in onze controller gebruik willen maken van onze repository, dan hoeven dat alleen maar aan te geven. We declareren een globale variabele en vullen die in, in de constructor.

```
public class BookController {
    private final BookRepository bookRepository;

    public BookController(BookRepository bookRepository){
        this.bookRepository = bookRepository;
    }
}
```

We hadden al een Bean van onze `BookRepository` in de Spring Applicationcontext gezet, door deze te extenden met `JpaRepository` (in de superklasse staat de `@Component` annotatie ergens. Je moet misschien even wat lagen van overerving doorzoeken). Dankzij het Dependency Injection framework kunnen we die Bean hier weer uit de Applicationcontext halen en gebruiken.

Note: We kunnen dependencies ook injecteren zonder de constructor te gebruiken. Dit kun je doen door de `@Autowired` annotatie te gebruiken boven de klasse variabele of boven de getter van die klassevariabele. Constructor injection heeft echter de voorkeur, omdat je de klasse variabele dan "final" kunt maken. Zo weet je zeker dat de repository (in dit geval) niet gaandeweg gewijzigd kan worden.

6. Service

In dit hoofdstuk wordt de zogenaamde servicelaag opgezet. Hierin kunnen de specifieke regels worden opgenomen die binnen het domein van toepassing zijn.

6.1 Business Logica

De servicelaag van een webservice is de plek waar code wordt geplaatst die te maken heeft met de specifieke toepassing. Dit wordt ook wel de business logica genoemd. Code waarin aannames en voorwaarden worden geïmplementeerd die te maken hebben met het domein waarover de webservice gaat.

De servicelaag is de laag die zich bevindt tussen de controllers aan de voorkant en het opslaan van gegevens aan de achterkant. Je zou ook kunnen zeggen dat de service laag de intelligentie van de applicatie vormt.

De controllers die we in het vorige hoofdstuk hebben besproken, hebben als taak de communicatie met de client af te handelen met behulp van HTTP. De controllers zijn in Spring Boot afhankelijk van de service laag voor het verwerken van de verzoeken die binnen komen. De service klassen valideren en interpreteren de binnenkomende data, zetten data manipulatie opdrachten uit richting de database en combineren gegevens voor een zinnvolle response. De data voor response wordt teruggegeven aan de controller laag die op zijn beurt de data in een HTTP response inpakt en doorstuurt naar de client.

6.2 Service klasse

Binnen Spring Boot wordt de service laag aangegeven met de `@Service` annotatie.

In de bijbehorende implementatie van de interface wordt de daadwerkelijke code geschreven. Alle in de interface gedefinieerde methoden moeten in de implementatie worden gerealiseerd.

```
@Service
public class BookService {

    public List<Book> getBooks() {
```

```

        return listOfBooks.getBooks();
    }

    public Book getBook(int id) {
        Book bookFound = listOfBooks.getBook(id);
        if (bookFound == null)
            throw new RecordNotFoundException("Cannot find book")
        return bookFound;
    }

    public int addBook(Book book) {
        return listOfBooks.addBook(book);
    }

    public void removeBook(int id) {
        Book bookFound = listOfBooks.getBook(id);
        if (bookFound == null)
            throw new RecordNotFoundException("Cannot find book")
        listOfBooks.removeBook(id);
    }

    public void updateBook(int id, Book book) {
        Book bookFound = listOfBooks.getBook(id);
        if (bookFound == null)
            throw new RecordNotFoundException("Cannot find book")
        listOfBooks.updateBook(id, book);
    }
}

```

Merk op dat in de service ook wordt gecheckt dat de gevraagde record ook echt bestaat. Zo niet dan wordt er een `RecordNotFoundException` gegenereerd. Die wordt zoals eerder besproken afgevangen en resulteert in een 404 Not Found response.

6.3 AutoWired

Tijdens het starten van de Spring Boot applicatie kan Spring Boot service detecteren en alvast instantiëren. Om de service elders in de applicatie te gebruiken kan de `@Autowired` annotatie worden gebruikt. Normaliter zal dit nodig zijn in de controller klasse.

```

private BookService bookService;

@Autowired
public class BookController(BookService bookService) {
    this.bookService = bookService;

    ...
}

```

De methoden in de service kunnen in de controller methodes worden als volgt worden gebruikt `bookService.getBooks()`.

De complete controller methode wordt dan:

```

@GetMapping(value =("/{id}")
public ResponseEntity<Object> getBook(@PathVariable("id") long id) {
    return ResponseEntity.ok().body(bookService.getBookById(id));
}

```

Zo hebben we een goede scheiding aangebracht tussen de controller en de servicelaag.

7. Data Transfer Objects

Dit hoofdstuk gaat over Data Transfer Objects (DTO). DTO's worden gebruikt om de data te structureren die bij de HTTP endpoints worden gebruikt te scheiden van de domeinmodellen die binnen de applicatie worden gebruikt.

7.1 Waarom DTO's?

Tot nu toe hebben we in de controllers de Request body en de Response body gebaseerd op de domeinclassen die we gemaakt hebben. Dit is niet altijd even wenselijk. Een deel van de interne structuur van de webservice ligt nu op straat. Vaak is het beter de data die de gebruiker aanbiedt in een POST of een PUT request maar aan banden te leggen. Sowieso willen we data van de gebruiker verifiëren voordat we de gegevens in de database opslaan.

Om dit voor elkaar te krijgen maken we gebruik van Data Transfer Objecten (DTO). Een Data Transfer Object is een soort tussenklasse. Bij een Request moet de gebruiker de gegevens aanbieden volgens de structuur van de DTO. De DTO wordt aan de betreffende service doorgegeven. De service klasse zorgt voor de vertaling naar een domeinklasse die via de repository in de database wordt gezet.

Zo zou voor een User klasse als deze:

```

class User {
    long id;
    String userName;
    String firstName;
    String lastName;
    String email;
    String password;
    String socialSecurityNumber;
    List<Authority> authorities;
}

```

... een DTO kunnen worden gedefinieerd zoals hieronder:

```

class UserDto {
    public String username;
    public String password;
    public String email;
    public List<String> authorities;
}

```

Alleen de relevante velden zijn nu opgenomen in de DTO en de authorities is een eenvoudige lijst van Strings in plaats van een lijst van Authority objecten. Aan de DTO klasse kan ook door Java validatie-annotaties worden toegevoegd om restricties op te leggen aan de waarden van attributen. Hiervoor is wel de hibernate validator nodig, die kan worden toegevoegd met behulp van Maven.


```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
  <version>2.7.5</version>
</dependency>
```

Annotation	Check
@Digits(integer=, fraction=)	Checks whether the annotated value is a number having up to integer digits and fraction fractional digits.
@Email	Checks whether the specified character sequence is a valid email address.
@Max(value=)	Checks whether the annotated value is less than or equal to the specified maximum.
@Min(value=)	Checks whether the annotated value is higher than or equal to the specified minimum
@NotBlank	Checks that the annotated character sequence is not null and the trimmed length is greater than 0.
@NotEmpty	Checks whether the annotated element is not null nor empty.
@Null	Checks that the annotated value is null
@NotNull	Checks that the annotated value is not null
@Pattern(regex=, flags=)	Checks if the annotated string matches the regular expression regex considering the given flag match
@Size(min=, max=)	Checks if the annotated element's size is between min and max (inclusive)
@Negative	Checks if the element is strictly negative. Zero values are considered invalid.
@NegativeOrZero	Checks if the element is negative or zero.
@Future	Checks whether the annotated date is in the future.
@FutureOrPresent	Checks whether the annotated date is in the present or in the future.

Annotation	Check
@PastOrPresent	Checks whether the annotated date is in the past or in the present.

Bovenstaand UserDto kan worden uitgebreid met een aantal validatie annotaties als volgt:

```
class UserDto {
    @NotBlank
    String userName;

    @NotBlank
    @Size(min=6, max=30)
    public String password;

    @NotBlank
    @Email
    String email;

    public List<String> authorities;
}
```

7.2 Mapping tussen DTO en model

Een Data Transfer Object kan dus gebruikt worden om de binnenkomende datastructuur onafhankelijk te maken van de domeinmodellen. De binnenkomende data moet echter wel worden omgezet van domeinobjecten die vervolgens via de repository worden opgeslagen in de database. De gegevens in een DTO moeten dus worden gemapt naar één of meerdere domeinmodellen. Dit gebeurt in de service laag. Een serviceklasse kan dus een DTO als argument hebben en eventueel ook een DTO teruggeven.

Het mappen van een DTO naar domeinmodellen is vaak een flinke klus. Ieder veld moet met een getter uit de DTO worden gehaald en met een setter in een nieuwe instantie van een domein model worden geplaatst. Met bovenstaand 'User' en 'UserDto' klassen wordt de mapping als volgt.

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PasswordEncoder passwordEncoder;

    public String createUser(UserRequestDto userRequestDto) {
        String encryptedPassword = passwordEncoder.encode(userRequestDto.getPassword());

        User user = new User();

        user.setUsername(userRequestDto.getUsername());
        user.setPassword(encryptedPassword);
        user.setEmail(userRequestDto.getEmail());
        user.setEnabled(true);
        user.addAuthority("ROLE_USER");

        User savedUser = userRepository.save(user);

        return savedUser.getUsername();
    }
    ...
}
```

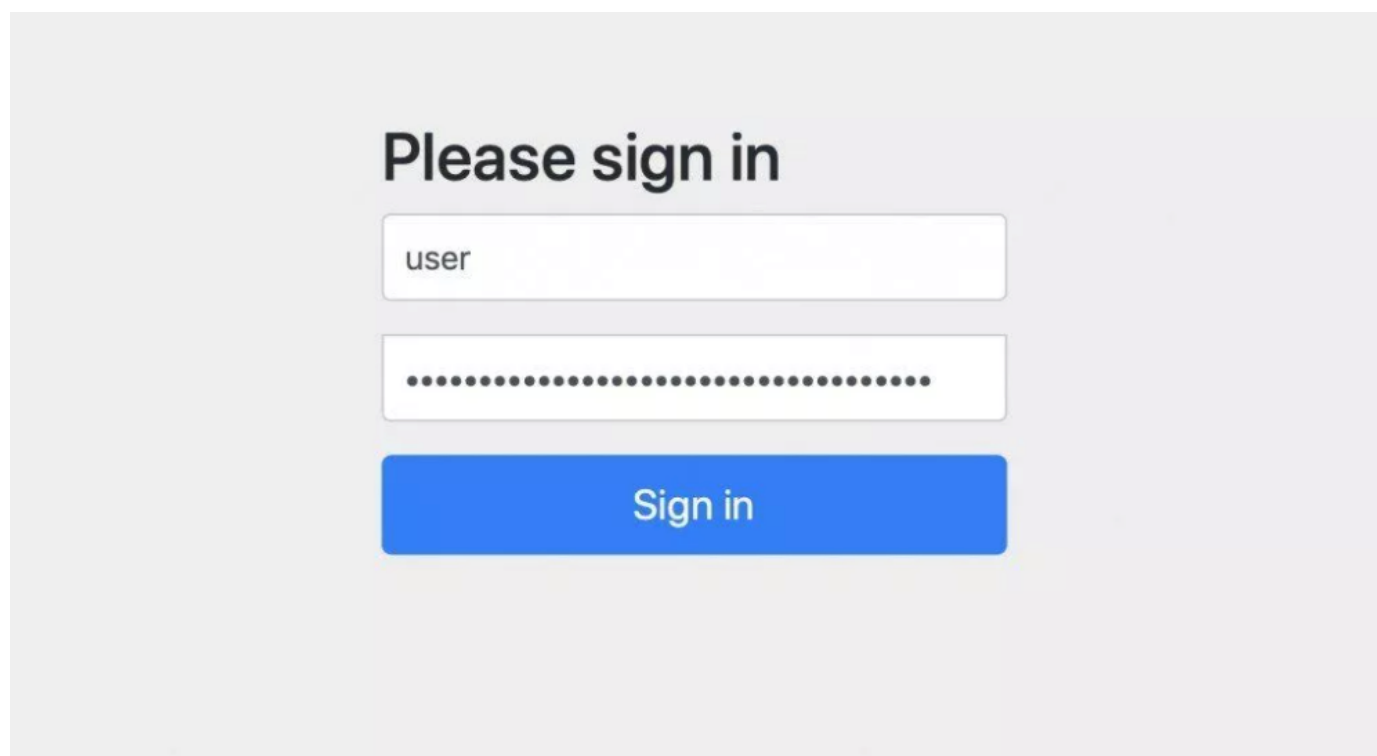
Bij deze mapping wordt ook het wachtwoord dat is opgegeven door de passwordEncoder versleuteld. De versleutelde versie wordt dan opgeslagen in de database.

8. Security

Security is een aspect dat natuurlijk heel belangrijk is bij webservices. Dit wordt in dit hoofdstuk besproken. Het draait hierbij grotendeels om gebruikersauthenticatie en autorisatie.

8.1 Authentication and authorization

Het beveiligen van toegang tot een applicatie bestaat uit twee delen. In eerste instantie wil je zeker weten wie de gebruiker is die de applicatie gebruikt. Dat wordt authenticatie genoemd. Hiervoor wordt bijvoorbeeld een gebruikersnaam en een wachtwoord gevraagd. Als de juiste combinatie van gebruikersnaam en wachtwoord wordt ingevoerd dan kun je ervan uit gaan dat dit inderdaad de gebruiker is die hij/zij zegt te zijn.



Er zijn vele manieren om de identiteit van de gebruiker te bevestigen. Zo kan naast het vragen om een gebruikersnaam en wachtwoord ook een derde item worden gevraagd. Bijvoorbeeld een code op een token-device. Dat wordt dan three-way-authentication genoemd.

Nadat de identiteit van de gebruiker is vastgesteld, wordt door de applicatie bepaald waar die gebruiker zoal toegang tot heeft. Wat mag de gebruiker zien? En wat mag de gebruiker doen? Dit zijn de rechten die een gebruiker heeft. Dit wordt autorisatie genoemd. Bij autorisatie wordt dus bepaald welke van de CRUD operaties de gebruiker mag uitvoeren op welke resource.

8.2 Security en Webservices

Bij een webservice zijn er een aantal aspecten die anders zijn dan bij applicaties of websites.

Ten eerste maakt een webservice geen gebruik van schermen. Dus ook niet van een loginscherm waarop de gebruiker een gebruikersnaam en wachtwoord kan invoeren. Een webservice maakt alleen gebruik van een HTTP-request en HTTP-response. Dit betekent dat de authenticatie gegevens in de request moeten worden meegestuurd.

Ten tweede is een Restful Webservice stateless. De webservice heeft geen geheugen. Iedere request en response staat op zichzelf en na het versturen van de response is de server alle interactie met de gebruiker vergeten. De gebruiker kan dus niet inloggen waarbij de identiteit van de gebruiker bij een volgende request bekend is. Om deze reden moet bij ieder request opnieuw de authenticatie gegevens worden meegestuurd.

8.3 HTTP

Zoals we gezien hebben heeft een HTTP-request een header en een body. In de header kunnen allerlei aanvullende gegevens worden meegestuurd. Dit is ook de plek waar de authenticatie gegevens van de gebruiker die de request doet naar de server worden gestuurd. HTTP kent hiervoor de standaard header: "Authorization".

Er zijn verschillende protocollen beschikbaar om authenticatie gegevens in een HTTP-request naar de server te versturen. Sommige daarvan zijn relatief eenvoudig en andere zijn wat complexer en zijn vaak gebaseerd op een reeks van berichten tussen client en server of maken zelfs gebruik van derde partijen. Hieronder is een lijst van verschillende protocollen voor authenticatie.

protocol

Basic authentication

Digest authentication

Bearer

OAuth

Basic Authentication

Basic Authentication is opgenomen in de definitie van HTTP. Bij Basic Authentication worden de gebruikersnaam en wachtwoord bij ieder HTTP request versleuteld meegestuurd in de header van de request. De versleuteling is eenvoudige Base-64 encoding. Base-64 encoding is een twoway encoding. Het versleutelde bericht kan ook weer worden gedecodeerd. Je kunt dit oefenen op de website <https://www.base64decode.org>.

username	password	username:password Base-64 encoded
springboot	p4ssw0rd	c3ByaW5nYm9vdDpwNHNzdzByZA==

Zoals hieronder weergegeven wordt deze encoded tekst bij Basic Authorization opgenomen in de header van de HTTP-request vooraf gegaan door de tekst "Basic" (en een spatie) in een header veld genaamd "Authorization".

Key	Value
Authorization	Basic c3ByaW5nYm9vdDpwNHNzdzByZA==

Omdat bij Basic Authentication zowel de gebruikersnaam als het wachtwoord in elke request wordt meegestuurd en deze beide eenvoudig te achterhalen zijn door een Base-64 decodering uit te voeren, biedt Basic Authentication weinig bescherming tegen kwaadwilligen die jouw wachtwoord willen achterhalen. Basic authentication wordt daarom altijd uitgevoerd in combinatie met HTTPS. HTTPS is een beveiligde variant van HTTP waarbij al het verkeer is versleuteld op basis van een server certificaat. HTTPS is daarom wel veilig waardoor Basic Authentication dus toch kan worden gebruikt.

Bearer Authentication

Om te voorkomen dat het wachtwoord bij ieder HTTP-request wordt meegestuurd kan Bearer Authentication worden gebruikt. Hierbij wordt voor een gebruiker een unieke token aangemaakt die kan worden gebruikt in plaats van het versturen van gebruikersnaam en wachtwoord.

8.4 Users en rollen

De meeste applicaties werken met gebruikers en rollen. Als een gebruiker geauthentiseerd is, kan aan de hand van de rollen die aan die gebruiker zijn toegekend worden bepaald welke functionaliteiten beschikbaar zijn. Een gebruiker kan meerdere rollen hebben. Een User en een Rol zijn entiteiten die we zelf kunnen maken. Een simpele variant daarvan kan als volgt zijn.

```
@Entity
public class User {
    @Id
    String username;
    String password;
}
```

Deze User zou dan ook nog een relatie met een Rol-klasse kunnen hebben. Zoals hierboven al genoemd werd, kan een User meerdere rollen hebben, maar het hoeft niet. Een Rol-klasse zou er als volgt uit kunnen zien:

```
@Entity
public class Rol{
    @Id
    Long id;
    String roleName;
}
```

Spring heeft zelf ook een ingebouwde User klasse. Op een paar *booleans* na, verschilt dat niet veel van de basis. Kijk maar eens <https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/core/userdetails/User.html>.

We zien in de Spring implementatie van User dat daar gesproken wordt over Authorities in plaats van rollen. Tussen die twee zit een subtiel verschil waar we in deze cursus niet verder op in gaan, het enige wat je hoeft te weten is dat wanneer je ervoor kiest om je User rollen te geven, dat je de rol-naam met een prefix "ROLE_" in je database moet opslaan. Als je een rol aan maakt in je code, dan schrijf je dat dus als volgt:

```
Role role = new Role();
role.setRoleName("ROLE_ADMIN");
En bij een Authority
Authority auth = new Authority();
Auth.setAuthName("ADMIN");
```

8.5 Spring Boot Security dependency

Om in een Spring Boot webservice met authentication en authorization te werken is het nodig om eerst een dependency aan het Maven pom.xml bestand toe te voegen.

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

Als je - nadat Maven de dependency heeft bijgewerkt - de applicatie probeert te gebruiken, zul je zien dat het niet meer werkt. Althans op een request dat eerder wel een reponse gaf, krijg je nu een "401 Unauthorized response". Spring Boot Security is geactiveerd en weigert terecht onze ongeautoriseerde request. We moeten het request nu voorzien van een juiste gebruikersnaam en wachtwoord. Omdat we nog niets hebben ingesteld voorziet Spring Boot ons van een gebruiker "user" met een wachtwoord dat is weergegeven in de opstart log van de applicatie. Kijk maar. Daar kom je de onderstaande regel tegen.

```
Using generated security password: ad0d042-04a5-42ad-ae74-659a36988994
```

Met behulp van Postman kunnen we een request sturen die een Authorization header, zoals in het vorige hoofdstuk is beschreven, heeft met daarin de Base-64 geencodeerde gebruikersnaam en wachtwoord. Navigeer naar het tabblad Authorization en selecteer Basic Auth (basic authentication). Vul vervolgens de gebruikersnaam en het wachtwoord in. Kijk ook even bij het tabblad Headers naar het veld Authorization. Je zult zien dat Postman de Authorization header heeft gemaakt inclusief de Base-64 encoding.

GET

http://localhost:8080/

Send

Params

Authorization ●

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Type

Basic Auth

The authorization header will be automatically generated when you send the request. [Learn more about authorization](#)

Username

user

Password

ad00d042-04a5-42ad-ae74-659a36988994

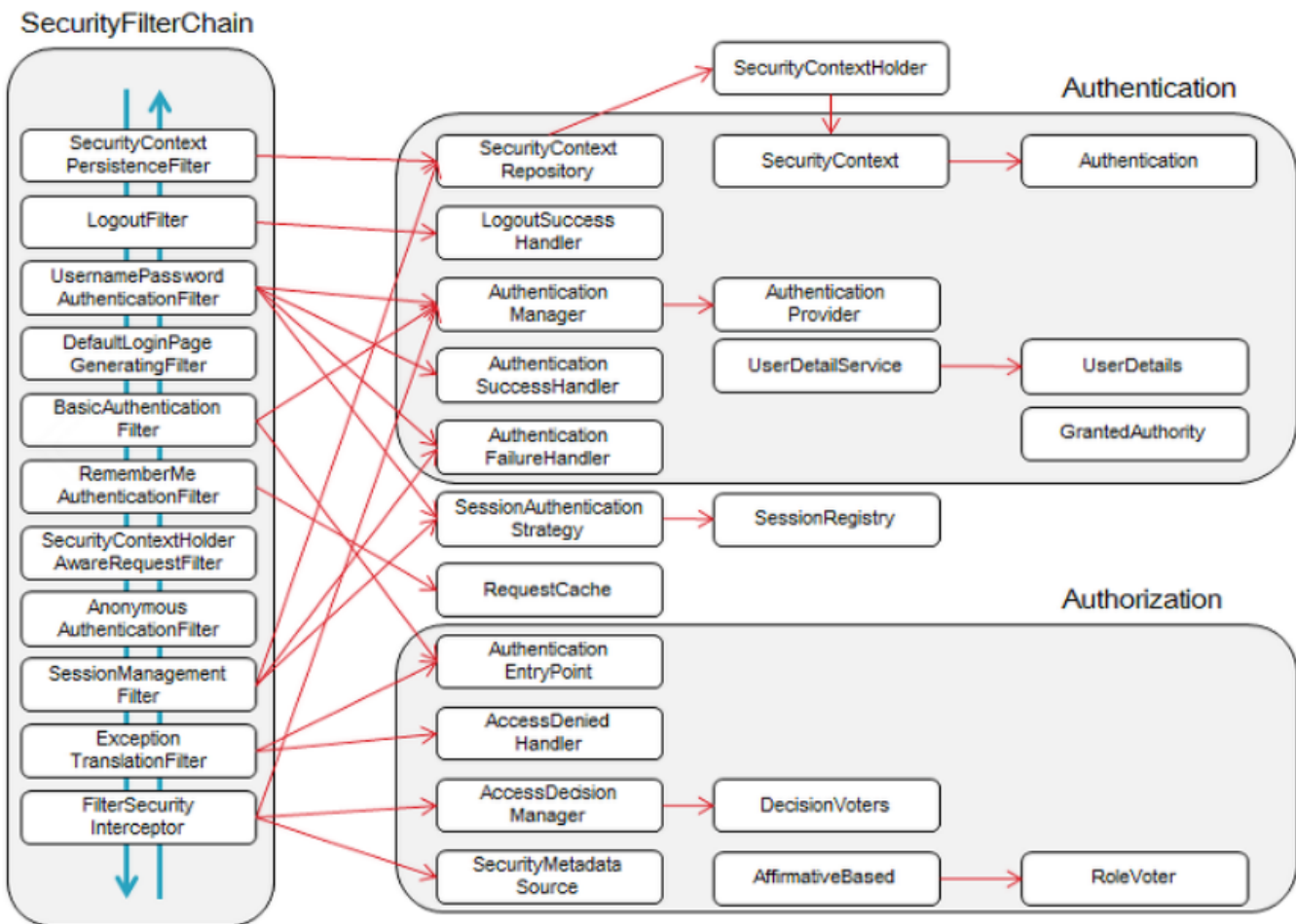
☒ Show Password

Als het goed is krijg je nu weer een normale response terug.

8.6 Spring Boot WebConfiguration

Zoals je al gemerkt hebt regelt Spring Boot een flink aantal zaken onder de motorkap tijdens het opstarten van de applicatie. Zo ook een aantal security issues. De standaard configuratie-instellingen die door Spring Boot zijn gekozen, kunnen we echter zelf weer aanpassen. Om zelf gebruikers voor onze applicatie te kunnen definiëren, moeten we aanpassingen maken in de configuratie van Spring Boot. Door de configuratie aan te passen kunnen we aangeven welke gebruikers gebruik mogen maken van de webservice en welke autorisaties die gebruikers hebben.

Spring's interne werking is gebaseerd op filters. Dat is een ingewikkeld web aan afhankelijkheden waar we in deze cursus niet verder op in zullen gaan. Het volgende plaatje is een illustratie van de filterchain:



Om onze eigen security hiertussen te plaatsen, gaan we een configuratie class maken:

```
// src.main.java.nl.novi.naam-van-jouw-project.config.springsecurityconfig.java

@Configuration
public class SpringSecurityConfig {
    @Bean
    public AuthenticationManager authenticationManager(HttpSecurity http) throws Exception {

    }

    @Bean
    protected SecurityFilterChain filter (HttpSecurity http) throws Exception {

    }
}
```

In deze configuratie klasse maken we minstens twee methodes aan. Beide methodes bevatten een `HttpSecurity`-object als parameter. We hoeven dat natuurlijk niet zelf in te vullen, dat doet Spring voor ons. Met de `HttpSecurity` die we van Spring krijgen, kunnen we dus alle gewenste security-instellingen

aanpassen. De eerste methode returned een `AuthenticationManager`: in deze methode kunnen we de authenticatie regelen. Ofwel: "dat je bent wie je zegt dat je bent". De tweede methode returned een `SecurityFilterChain`: in deze methode regelen we de autorisatie. Ofwel: dat je de juiste Rol of Authority hebt om toegang te krijgen tot een endpoint.

8.7 Gebruikers

Spring Boot security werkt met gebruikers die rollen toegewezen krijgen. Gebruikers hebben een username en password en kan één of meerdere rollen toegewezen krijgen. De rollen worden in de applicatie gedefinieerd en gebruikt om de toegangsrechten vast te leggen. Spring Boot biedt drie verschillende manieren om gebruikers te definiëren. In toenemende mogelijkheden (en complexiteit) zijn dat `InMemoryAuthentication`, `JdbcAuthentication` en `UserDetailsAuthentication`.

Deze methoden worden geïmplementeerd door middel van een `AuthenticationManager`-methode, zoals in onderstaande code voorbeelden zal worden weergegeven. Op het moment dat er een authenticatie methode is gedefinieerd zal Spring Boot security niet meer een standaard gebruiker met tijdelijk wachtwoord aanmaken, zoals eerder beschreven in 10.5.

`InMemoryAuthentication`

Bij `InMemoryAuthentication` worden gebruikers met wachtwoorden en rollen in de code gedefinieerd. Die kunnen dus niet worden aangepast door de gebruiker of iets dergelijks. Dit kan worden gebruikt voor een eerste proefversie van de applicatie maar zal doorgaans niet worden gebruikt in een echte applicatie.

```
@Bean
public AuthenticationManager authManager(HttpSecurity http) throws Exception {
    AuthenticationManagerBuilder authenticationManagerBuilder = http.getSharedObject(AuthenticationManagerBuilder.class);

    authenticationManagerBuilder.inMemoryAuthentication().withUser("user").password("{noop}geheim").roles("USER")
        .and().withUser("admin").password("{noop}geheim").roles("USER");
    return authenticationManagerBuilder.build();
}
```

In dit bovenstaande voorbeeld worden twee gebruikers gedefinieerd: `user` en `admin`. Beiden hebben 'geheim' als wachtwoord. Er zijn ook twee rollen gedefinieerd: 'USER' en 'ADMIN'. De gebruiker `user` heeft alleen de rol 'USER' toegewezen gekregen. De gebruiker 'admin' heeft beide rollen toegewezen. Gekeken. Je hebt misschien al het stukje "{noop}" gezien. Dit betekent dat wij de `NoopPasswordEncoder` gebruiken. De `NoopPasswordEncoder` doet praktisch niks, dus in feite gebruiken we nu "plain tekst" passwords in onze applicatie, zonder encryptie. In de volgende paragraaf gaan we de passwordencoder behandelen.

`JdbcAuthentication`

Bij `JdbcAuthentication` wordt gebruikt gemaakt van een database om gebruikers en rollen in op te slaan. Gebruikers kunnen dan door de gebruiker aan de applicatie worden toegevoegd. `JdbcAuthentication` maakt gebruik van de datasources die in de `application.properties` is geconfigureerd. Dit hebben we in hoofdstuk 9 "Repository" ook gezien.

```
private final DataSource dataSource;

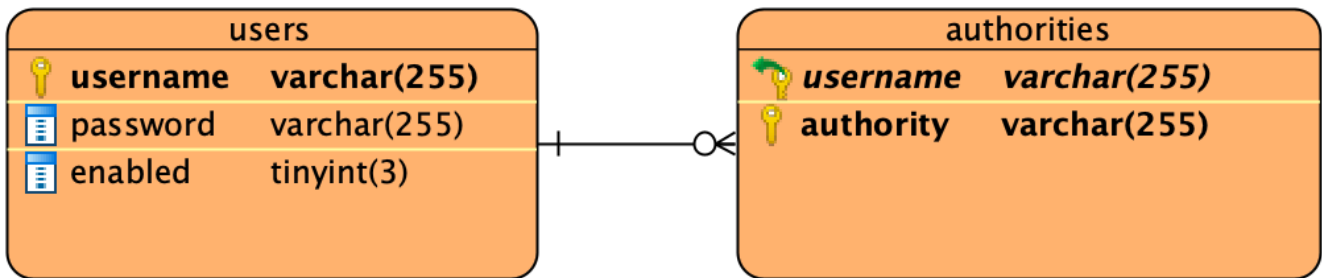
public MySecurityConfig(DataSource dataSource) {
    this.dataSource = dataSource;
}

@Bean
public AuthenticationManager authManager(HttpSecurity http) throws Exception {
    AuthenticationManagerBuilder authenticationManagerBuilder = http.getSharedObject(AuthenticationManagerBuilder.class);

    authenticationManagerBuilder.jdbcAuthentication().dataSource(dataSource).usersByUsernameQuery(
        "SELECT username, password, enabled" +
        " FROM users" +
        " WHERE username=?"
    ).authoritiesByUsernameQuery(
        "SELECT username, authority" +
        " FROM authorities" +
        " WHERE username=?"
    );

    return authenticationManagerBuilder.build();
}
```

Zoals hier is te zien, worden twee SQL queries gebruikt om de gebruikers en de rollen op te halen van de tabellen 'users' en 'authorities' in de database. Als de tabellen exact zijn geconfigureerd als hieronder beschreven dan kunnen de queries worden weggelaten en gebruikt Spring Boot de beschreven tabel- en veldnamen.



Deze tabellen kunnen op de gangbare manier worden beheerd met behulp van een controller klasse, een service klasse, een repository klasse en natuurlijk de twee model klassen `User` en `Authority`.

```
// User.java
@Entity
@Table(name = "users")
public class User {
    @Id
    @Column(nullable = false, unique = true)
    private String username;
    @Column(nullable = false)
    private String password;
    @Column(nullable = false)
    private boolean enabled = true;
    @OneToMany(
        targetEntity = Authority.class,
        mappedBy = "username",
        cascade = CascadeType.ALL,
        orphanRemoval = true,
        fetch = FetchType.EAGER
    )
    private Set<Authority> authorities = new HashSet<>();
    // getters and setters ...
}
```

```
// Authority.java

@Entity
@Table(name = "authorities")
public class Authority implements Serializable {
    @Id
    @Column(nullable = false)
    private String username;
    @Id
    @Column(nullable = false)
    private String authority;
    // getters and setters ...
}
```

userDetailsAuthentication

Een derde manier om de authentication te implementeren in Spring Boot is met behulp van `userDetailsAuthentication`. Alhoewel deze methode meer mogelijkheden biedt, zullen we deze methode voor nu niet verder bespreken. De oplettende kijker heeft ook al opgemerkt dat we in dit laatste voorbeeld `Authority` hebben gebruikt en niet `Role`!

8.8 Password encryption

Nu we de gebruiker kunnen opslaan in een database is het tijd om aandacht te besteden aan hoe de wachtwoorden zijn opgeslagen. Het is niet de bedoeling dat de wachtwoorden als gewone tekst worden opgeslagen in de database. Dit is een *bad practice* en wordt door de recente versies van Spring Boot niet meer ondersteund.

De way-to-go is om een een-richtingsversleuteling te implementeren. Het wachtwoord wordt versleuteld en het resultaat wordt opgeslagen in de database. Bij het controleren van een opgegeven wachtwoord wordt deze ook versleuteld en het resultaat hiervan wordt dan vergeleken met de versleutelde wachtwoord dat is opgeslagen.

Er zijn allerlei encryptie algoritmes denkbaar. In Spring Boot wordt standaard het `bcrypt` algoritme gebruikt. Dit is een state of art one-way encryptie algoritme. Van een wachtwoord kan een hash gegenereerd worden die niet terug kan worden gevoerd tot het originele wachtwoord. Om te voorkomen dat het wachtwoord met behulp van brute-force wordt achterhaald, is in dit algoritme ook een rekenkundige vertraging ingebouwd waardoor het relatief lang duurt om een wachtwoord te encrypten. Dit is een belangrijke bescherming tegen een brute-force aanval.

Om de password encoder in Spring Boot te activeren is in het nodig deze in de web security configuratie als een component beschikbaar te maken. Dit kan door hiervan een bean te genereren.

Een `@Bean` is iets wat door Spring wordt ingeladen tijdens de startup, net als een `@Service`, een `@RestController` of een `@Component`.

```
@Configuration
public class MySecurityConfig
{
    ...
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    ...
}
```

Je kunt je eigen wachtwoord encoders volgens het BCrypt algoritme op <https://bcrypt-generator.com/>

8.9 Authorization

Als eenmaal is vastgesteld dat de gebruiker inderdaad de gebruiker is die deze zegt te zijn dan moet er worden bepaald waar die gebruiker toegang tot heeft. Met andere woorden: de autorisatie moet worden bepaald. Wie mag wat?

Bij een webservice draait autorisatie om het bepalen welke endpoints met welke HTTP methoden toegankelijk zijn voor de verschillende gebruikers. Spring Boot maakt hiervoor gebruik van de rollen die aan gebruikers zijn toegekend.

In onze eigen security configuratie class wordt de autorisatie geïmplementeerd door de `AuthorizeHttpRequests` methode van het `HttpSecurity` object.

```
@Bean
protected SecurityFilterChain filter (HttpSecurity http) throws Exception {
    http
        .csrf().disable()
        .httpBasic().disable()
        .cors().and()
        .authorizeHttpRequests()
        .requestMatchers(HttpMethod.GET, "/info").hasRole("USER")
        .requestMatchers("/users/**").hasAnyRole("ADMIN", "USER")
        .requestMatchers("/admins").hasAuthority("ROLE_ADMIN")
        .requestMatchers(HttpMethod.DELETE, "/users/{id}").hasRole("ADMIN")
        .requestMatchers("/authenticate").permitAll()

        .anyRequest().denyAll()
        .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

Zoals je in bovenstaande code ziet, worden hier allerlei zaken rondom security geregeld. Zo wordt hier het HTTP Basic Authentication protocol uitgeschakeld. Wanneer we met de `InMemoryAuthorization` werken, willen we die juist vaak aan hebben. Er worden ook al een aantal andere zaken uitgezet, omdat deze voor een webservice niet van belang zijn. Er wordt geen gebruik gemaakt van een cross-site forgery token. Het gebruik van een login-formulier wordt niet gebruikt. En ook wordt er geen session cookie verstrekt. Er wordt ook iets aangezet: namelijk de CORS. Dit is de Cross Origin Resource Sharing en is met name van belang als je met een frontend wilt kunnen communiceren.

Ten aanzien van de autorisatie zijn er een aantal regels die bepalen welke endpoints voor welke rollen toegankelijk zijn. Eerst wordt met een `requestMatcher` methode bepaald welk endpoint wordt bedoeld. Dit doen we met een string, en noemen we het path. We kunnen in het path ook de catch-all `***` gebruiken. Dit betekent "alles wat hierna komt". Naast het path kan eventueel ook een HTTP-methode worden gespecificeerd in de `requestMatcher`. Vervolgens kunnen er verschillende methoden worden gebruikt om te bepalen wie bij de endpoint wordt toegelaten.

Methodes:

- `hasRole()`;
- `hasAnyRole()`;
- `hasAuthority()`;
- `hasAnyAuthority()`;
- `authenticated()`;
- `permitAll()`;
- `denyAll()`;

Dit is dus de plek waar de toegang tot de webapplicatie wordt geregeld. Voor alle beschikbare endpoints moet hier dus worden bepaald wie erbij mag. Het is verstandig om je security configuratie altijd te eindigen met `denyAll()`. Zo weet je zeker dat endpoints die je per ongeluk vergeten bent te configureren, geen security bedreiging zullen vormen.

Compleet:

```
@Configuration
public class MySecurityConfig {
    private final DataSource dataSource;
    public MySecurityConfig(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Bean
    PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }
}
```



```
}

@Bean
protected SecurityFilterChain filter (HttpSecurity http) throws Exception {

    http
        .csrf().disable()
        .httpBasic().disable()
        .cors().and()
        .authorizeHttpRequests()
        .requestMatchers(HttpMethod.GET, "/info").hasRole("USER")
        .requestMatchers("/users/**").hasAnyRole("ADMIN", "USER")
        .requestMatchers("/admins").hasAuthority("ROLE_ADMIN")
        .requestMatchers(HttpMethod.DELETE, "/users/{id}").hasRole("ADMIN")
        .requestMatchers("/authenticate").permitAll()

        .anyRequest().denyAll()
        .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    return http.build();
}

@Bean
public AuthenticationManager authManager(HttpSecurity http) throws Exception {
    AuthenticationManagerBuilder authenticationManagerBuilder = http.getSharedObject(AuthenticationManagerBuilder.class);
    authenticationManagerBuilder.jdbcAuthentication().dataSource(dataSource)
        .usersByUsernameQuery("SELECT username, password, enabled" +
            " FROM users" +
            " WHERE username=?" )
        .authoritiesByUsernameQuery("SELECT username, authority" +
            " FROM authorities" +
            " WHERE username=?");
    return authenticationManagerBuilder.build();
}
```

8.10 JSON Web Token Authentication

Tot nu toe zijn we ervan uitgegaan dat de authenticatie van de communicatie met de webservice is beveiligd met het HTTP basic protocol. Hierbij wordt de gebruikersnaam en het wachtwoord met base-64 versleuteld en bij ieder request meegestuurd in de authorization header van de request.

HTTP-request Header Key	Value
Authorization	Basic dXNlcjpwYXNzd29yZA==

Zoals we gezien hebben is de base-64 hash code eenvoudig te decoderen. Dit betekent dat gebruikersnamen en bijbehorende wachtwoorden eenvoudig te achterhalen zijn. Dit is niet wenselijk.

Een mogelijkheid om in ieder geval geen wachtwoorden heen en weer te sturen is om gebruik te maken van een JSON Web Token. Zo'n token wordt door de server aangemaakt voor een gebruiker. Vervolgens kan en moet de gebruiker deze token gebruiken om toegang te krijgen tot de webservice. De token wordt ook in de HTTP-request authorization header.

HTTP-request Header Key	Value
Authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWwiOiJ1c2VyliwiZXhwIjoxNjMzMTczMDcwLCJpYXQiOiJlY2MzIiwiaWF0Ij0yZy6RS3Z2Gln3tkP99SaMlrHla2EagpuAU

In de token is onder andere de gebruikersnaam versleuteld. Samen met een secret die alleen bij de back-end server bekend is. Alleen de webservice kan een ontvangen token decoderen en daarmee bepalen welke gebruiker de token heeft verzonden. In de token is ook geldigheidsperiode versleuteld. Een token kan dus na verloop van tijd verlopen. De gebruiker kan dan een nieuwe token aanvragen.



Hier onder is de code gegeven waarmee een .JWT-token kan worden gecreëerd

26-5-2023 09:57

```

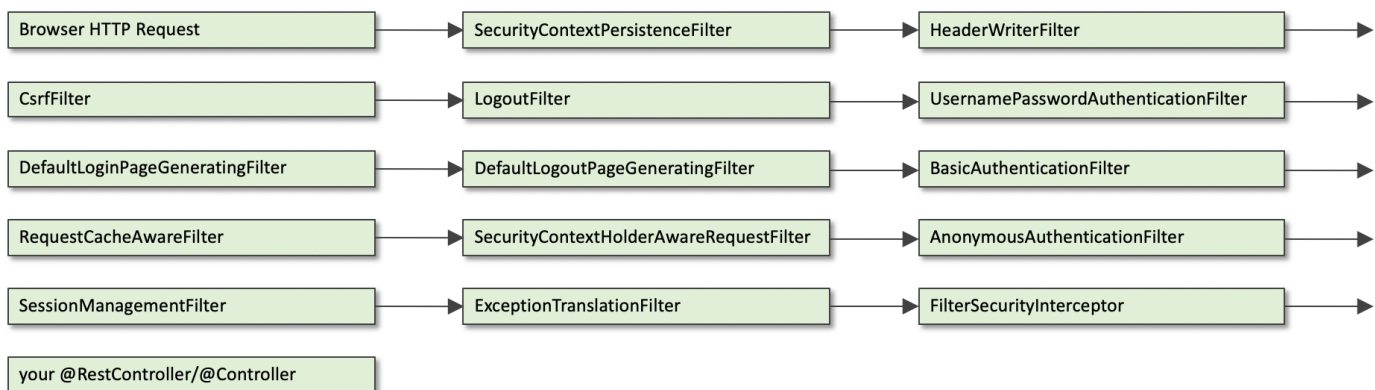
        .setClaims(claims)
        .setSubject(subject)
        .setIssuedAt(new Date(System.currentTimeMillis()))
        .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60 * 24 * 10))
        .signWith(getSigningKey(), SignatureAlgorithm.HS256)
        .compact();
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return username.equals(userDetails.getUsername()) && !isTokenExpired(token);
    }
}

```

Let er in bovenstaand voorbeeld op dat de SECRET_KEY minimaal 256 bits lang moet zijn, of grofweg 45 tekens.

Als de gebruiker de JWT-token heeft opgevraagd en ontvangen dan kan deze worden gebruikt als authenticatie in plaats van het versturen van een gebruikersnaam en een wachtwoord. Bij Spring Boot wordt de authenticatie en de daarmee samenhangende autorisatie afgehandeld voordat de HTTP-request bij de geïmplementeerde controllers binnenkomt. Hiervoor hanteert Spring Boot een zogenaamde filter chain.



De onderdelen van de filter chain worden één voor één afgelopen. Ieder onderdeel heeft de mogelijkheid om het HTTP-request te weigeren waardoor de filter chain worden verbroken. Pas als het request door alle onderdelen van de filter chain wordt doorgelaten, komt het bij de controllers aan. Het verifiëren van een JWT-token dient ook in de filter chain te worden opgenomen. Dat kan door een RequestFilter te implementeren en die aan de filter chain toe te voegen.

```

@Component
public class JwtRequestFilter extends OncePerRequestFilter {

    private final CustomUserDetailsService userDetailsService;

    private final JwtUtil jwtUtil;

    public JwtRequestFilter(CustomUserDetailsService userDetailsService, JwtUtil jwtUtil) {
        this.userDetailsService = userDetailsService;
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {

        final String authorizationHeader = request.getHeader("Authorization");

        String username = null;
        String jwt = null;

        if (authorizationHeader != null && authorizationHeader.startsWith("Bearer ")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUsername(jwt);
        }

        if (username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);

            if (jwtUtil.validateToken(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken usernamePasswordAuthenticationToken = new UsernamePasswordAuthenticationToken(
                    userDetails, null, userDetails.getAuthorities()
                );
                usernamePasswordAuthenticationToken.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            }
        }
        filterChain.doFilter(request, response);
    }
}

```

Tenslotte moet deze RequestFilter worden toegevoegd aan de filter chain in MySecurityConfig.java:

```

@Bean
protected SecurityFilterChain filter(HttpSecurity http) throws Exception {

    http
        .csrf().disable()
        .httpBasic().disable()
        .cors().and()
        .authorizeHttpRequests()

    ...

    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}

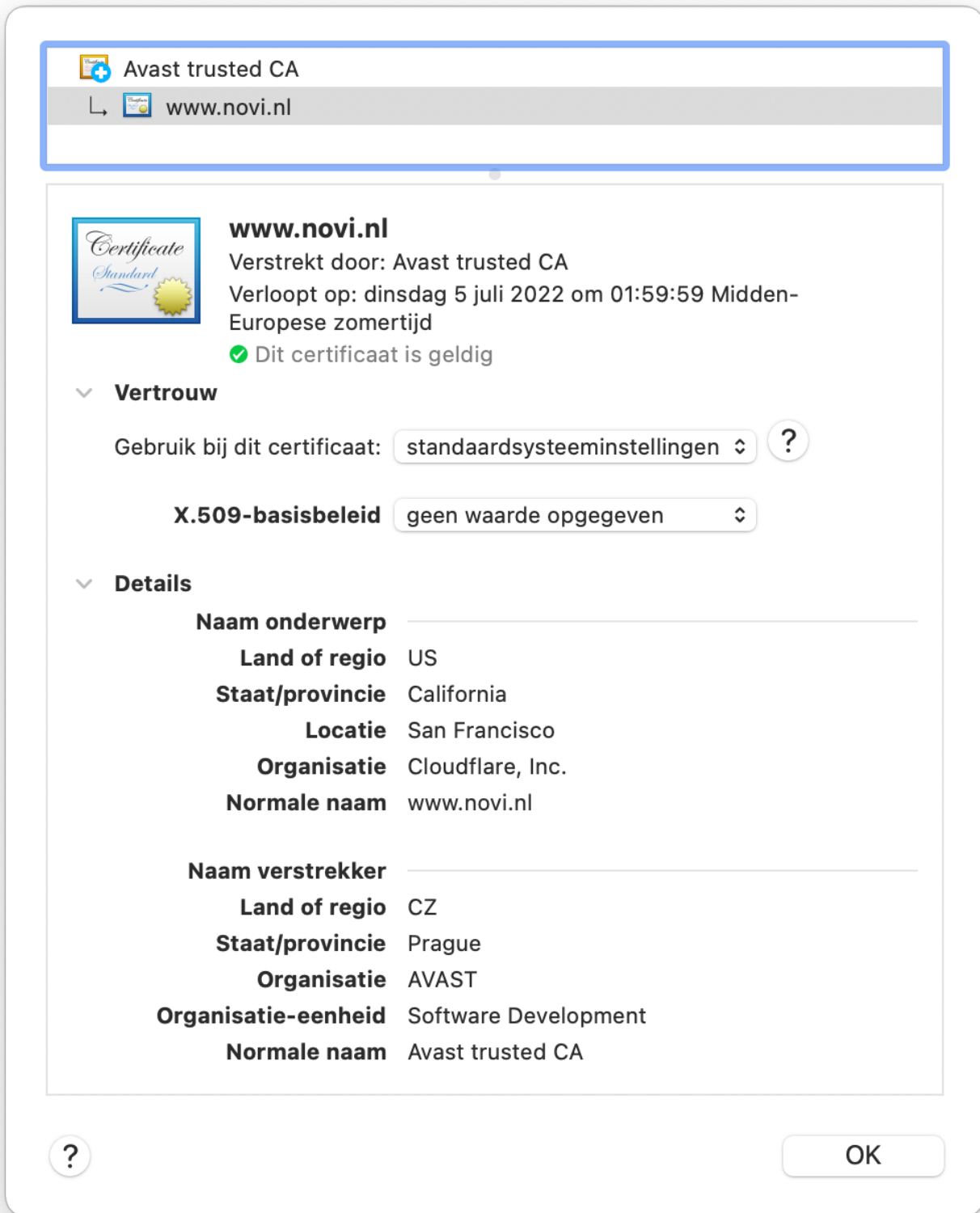
```

}

8.11 HTTPS

Het belang van het gebruik van HTTPS is al eerder genoemd. Als je geen HTTPS gebruikt gaan alle gegevens onversleuteld over het internet is is daar eigen door Jan en alleman zichtbaar. Dit is natuurlijk zeer ongewenst. Ook de uitwisseling van gegevens met de webservice die we met Spring Boot ontwikkelen zou op basis van HTTPS moeten plaatsvinden.

Bij HTTPS wordt al het verkeer tussen de server en de client versleuteld. Door deze versleuteling zijn de gegevens niet meer voor iedereen zichtbaar. Voor de versleuteling wordt een SSL beveiligingscertificaat gebruikt. Zo'n certificaat wordt toegekend aan de server. In een webbrowser wordt in het adres aangegeven dat een website is beveiligd met een SSL-certificaat en dat het verkeer met de website dus plaatsvindt op basis van HTTPS. Het adres van de website wordt dan vooraf gegaan door https zoals bijvoorbeeld de website van het NOVI: <https://www.novi.nl/>. Als je op het slotje klikt kun je de SSL-certificaat inzien.



In Spring Boot kan ook gebruik worden gemaakt van HTTPS. Hiervoor moeten aan aantal acties worden gedaan. Allereerst hebben we een SSL certificaat nodig. Tijdens het ontwikkelen zullen we gebruik maken van een self-signed certificaat. Dit is een certificaat die zelf maken. Voer in een command venster het volgende in:

```
keytool -genkey -keyalg RSA -alias certificate -keystore  
certificate.jks -storepass p4ssw0rd -validity 365 -keysize 4096 -  
storetype pkcs12
```

Na een aantal informatieve vragen wordt hiermee wordt een certificaat gegenereerd in het bestand certificate.jks. Plaats dit bestaand in het project.

Bijvoorbeeld in de map resources. In de application.properties worden de volgende instellingen toegevoegd:

```
server.ssl.key-store=classpath:certificate.jks
server.ssl.key-store-type=pkcs12
server.ssl.key-store-password=password
server.ssl.key-password=password
server.ssl.alias=certificate
server.port=8443
```

De webservice is nu alleen bereikbaar via HTTP en op port 8443.

<https://localhost:8443/>

8.12 CORS

Cors staat voor **Cross Origin Resource Sharing** en is belangrijk om je resources beschikbaar te maken voor een frontend. In de backend testen wij onze endpoints meestal met Postman. Postman is een erg slimme applicatie die zich voor kan doen alsof deze van dezelfde origin afkomstig is als de applicatie. Een frontend applicatie geschreven in React, kan dat niet.

Wat we bedoelen met de origin is "localhost:8080". Dit is het standaard adres van een backend applicatie en tevens de origin. De standaard origin van een React applicatie is "localhost:3000". Dit komt niet met elkaar overeen en daarom kunnen ze niet goed met elkaar communiceren. In de frontend zul je een cross-origin-error ontvangen als je probeert endpoints van jouw backend aan te spreken.

We kunnen dit oplossen door in de backend de CORS te regelen. De eerste stap hiervoor hebben we al gezien. We moeten in onze security configuratie de `.cors()`-functie toevoegen. Ook moeten we aangeven voor welke endpoints, methodes en origins we de cors willen toestaan. We kunnen dit op klasse of methode niveau doen, door boven een klasse of methode de `@CrossOrigin` annotatie toe te voegen. We kunnen het echter ook globaal regelen met een cross origin configuratie klasse die voor alle endpoints geldt:

```
@Configuration
public class GlobalCorsConfiguration
{
    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                    .allowedOrigins("*")
                    .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE", "OPTIONS");
            }
        };
    }
}
```

8.13 Zelftest

9. Testing

In dit hoofdstuk wordt besproken hoe de applicatie kan worden getest met unittests en integratietesten. Zonder bijbehorende testen is een applicatie niet af.

9.1 Unittesting

Testing is cruciaal voor het ontwikkelen van betrouwbare software. Als ontwikkelaar probeer je natuurlijk alles wat je maakt tijdens het ontwikkelen te testen. De code wordt opgestart, je voert de nodige acties uit en bekijkt het resultaat. Als het overeenkomt met wat je verwacht dan ben je tevreden en ga je door met het ontwikkelen.

In feite doet een unittest hetzelfde. Een stukje code wordt uitgevoerd en de uitkomst wordt vergeleken met de verwachte uitkomst. Als die hetzelfde zijn dat is de test geslaagd.

Met het ontwikkelen van unittests borg je de werking van je code. Als je eenmaal een aantal unittests hebt geschreven die verschillende aspecten van jouw software testen dan kunnen die keer op keer opnieuw worden uitgevoerd om er zo zeker van te zijn dat de code nog steeds doet wat er van verwacht wordt.

Bij unittesting wordt er vanuit gegaan dat het stukje code dat wordt getest op zichzelf staat en niet afhankelijk is van andere delen van de code. Een goed ontworpen functie of klasse, dus zonder afhankelijkheden is hiervoor uitermate geschikt.

In het onderstaand voorbeeld is een klasse Counter geschreven met een methode.

```
public class Counter {

    private int count = 0;

    public void add() {
        count += 1;
    }

    public int getCount() {
        return count;
    }

}
```

Voor deze klasse kan worden een unittest worden geschreven met JUnit.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class CounterTest {

    @Test
```

```
void testCounterAddOne() {  
    // Arrange  
    Counter counter = new Counter();  
  
    // Act  
    counter.add();  
  
    // Assert  
    int expected = 1;  
    int actual = counter.getCount();  
    assertEquals(expected, actual);  
}  
  
}
```

In het bovenstaand voorbeeld wordt een unittest met behulp van de bibliotheek [JUnit](#) geïmplementeerd.

De unittest is opgebouwd volgens het AAA patroon: Arrange, Act en Assert. Oftwel: klaarzetten, actie uitvoeren en bevestigen.

In een webservice zoals die wordt opgebouwd met Spring Boot kunnen de domeinklassen met unittests worden getest. Deze domeinklassen zijn echter van POJO (Plain Old Java Object) klassen met vaak niet veel meer dan attributen, constructors, getters and setters.

9.2 Mocking

Het testen van een Spring Boot webservice met unittests is lastig, omdat de verschillende lagen afhankelijk zijn van onderliggende lagen. Zo zijn de controller klassen afhankelijk van de service klassen. En de service klassen zijn op hun beurt weer afhankelijk van de repository.

Om toch automatische tests te kunnen gebruiken wordt er gebruik gemaakt van mocks. Met een mock kan het gedrag van een klasse worden geïmiteerd. Een mock gedraagt zich als de originele klasse en worden gebruikt in plaats van de echte klasse. Hiermee is het dan toch mogelijk unittests uit te voeren zonder afhankelijk te zijn van de ... afhankelijkheden.

Een voorbeeld is het gebruik van een database die door de repository wordt gebruikt om gegevens op te halen. Als hiervoor een unittest zou worden opgezet dan zou het gebruik van een database voor problemen kunnen zorgen. Het is traag en het telkens klaarzetten van een gecontroleerde set records is lastig. Met het gebruik van een mock is worden deze problemen onzeild.

Voor een Spring Boot webservice zullen we de verschillende lagen van verschillende mocks moeten voorzien voordat we unittests kunnen uitvoeren. Een ander aspect waarmee we rekening moeten houden, is de autoconfiguratie van Spring Boot. Tijdens het opstarten van de webservice doorloopt Spring Boot een uitgebreide reeks configuratie instellingen die resulteren in allerlei vooraf geïnstantieerde componenten. Bij het testen zullen we er voor moeten zorgen dat deze autoconfiguratie aansluit bij unittests die we willen uitvoeren.

De dependencies die we nodig hebben zijn al opgenomen in het POM.XML bestand in de Spring Boot Starter Projecten.

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

9.3 Mockito

Voor het werken met mocks maken we gebruik van de bibliotheek [Mockito](#). Het testen met mocks van Mockito volgt het Given - When - Then patroon. Dit is een variatie op het eerder genoemde AAA-patroon.

Een voorbeeld van het gebruik van Mockito waarbij een mock wordt gemaakt van een klasse MyList gaat als volgt: het gedrag van de de mock wordt bepaald met een when thenReturn regel. Dit is typisch voor Mockito. Als er een bepaald gedrag wordt gevraagd dat zal er een vooraf aangegeven response optreden. In dit geval zal de methode add een false teruggeven als return waarde. Vervolgens wordt de methode uitgevoerd en het resultaat vergeleken met de verwachte uitkomst.

```
import java.util.AbstractList;

class MyList extends AbstractList<String> {
}...

```java
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;

public class TestMyList {

 @Test
 public void simpleReturnBehaviour() {

 // given
 MyList listMock = Mockito.mock(MyList.class);
 when(listMock.add(anyString())).thenReturn(false);

 // when
 boolean added = listMock.add("abc");

 // then
 assertFalse(added);
 }

}
```

#### Testing een repository

Hieronder is een voorbeeld van een unittest voor een repository.

```
import org.junit.Test;
import static org.junit.jupiter.api.Assertions.*;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;

@DataJpaTest
@ContextConfiguration(classes={MyApplication.class})
class CustomerRepositoryTest {

 @Autowired
 private TestEntityManager entityManager;

 @Autowired
 private CustomerRepository customerRepository;

 @Test
 void testFindByLastName() {

 // given
 Customer customer = new Customer("Albert", "Einstein");
 entityManager.persist(customer);
 entityManager.flush();

 // when
 Customer found = customerRepository.findByLastName("Einstein");

 // then
 String expected = "Albert Einstein";
 String actual = found.getFullName();
 assertEquals(expected, actual);
 }

}
```

#### Testing een service klasse

Hieronder is een voorbeeld van een unittest voor een service klasse.

```
import org.junit.jupiter.api.Test;

import org.mockito.Mock;
import org.mockito.Mockito;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.mock.mockito.MockBean;
import org.springframework.test.context.ContextConfiguration;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
@ContextConfiguration(classes={MyApplication.class})
public class CustomerServiceTest {

 @Autowired
 private CustomerService customerService;

 @MockBean
 private CustomerRepository customerRepository;

 @Mock
 Customer customer;

 @Test
 public void testGetCustomerByLastName() {
 customer = new Customer("Albert", "Einstein");
```



```

Mockito
 .when(customerRepository.findByLastName(customer.getLastName()))
 .thenReturn(customer);

String name = "Einstein";
String expected = "Albert Einstein";

Customer found = customerService.getCustomerByLastName(name);

assertEquals(expected, found.getFullName());
}

```

#### Testing een controller

Hieronder is een voorbeeld van een unittest voor een controller.

```

import org.junit.jupiter.api.Test;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.http.MediaType;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.web.servlet.MockMvc;

import java.util.Arrays;
import java.util.List;

import static org.hamcrest.Matchers.hasSize;
import static org.hamcrest.Matchers.is;
import static org.mockito.BDDMockito.given;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@WebMvcTest
@ContextConfiguration(classes={MyApplication.class})
public class CustomerControllerTest {

 @Autowired
 private MockMvc mvc;

 @MockBean
 private CustomerService service;

 @Test
 public void testEndpointGetAllCustomers() throws Exception {

 Customer customer = new Customer("Albert", "Einstein");
 List<Customer> allCustomers = Arrays.asList(customer);

 given(service.getAllCustomers()).willReturn(allCustomers);

 mvc.perform(get("/customers")
 .contentType(MediaType.APPLICATION_JSON))
 .andExpect(status().isOk())
 .andExpect(jsonPath("$.", hasSize(1)))
 .andExpect(jsonPath("$.length()", is(1)))
 .andExpect(jsonPath("$.lastName", is("Einstein")));

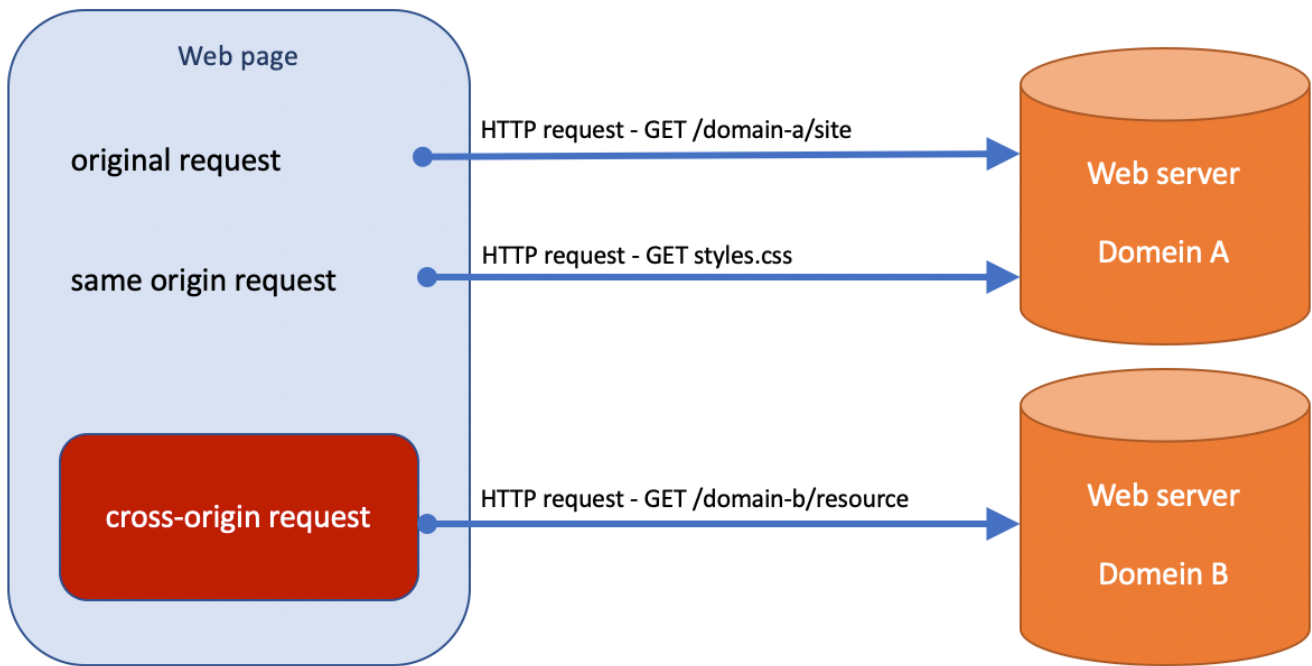
 }
}

```

## 10. Tenslotte

### 10.1 CORS

Cross Origin Resource Sharing (CORS) is een protocol dat gebruikmaakt van HTTP headers om een browser te laten weten dat een web applicatie die afkomstig is van domein A, toestemming heeft om resources te gebruiken van een server die op domein B staan.



De server van het tweede domein geeft toestemming aan een cross-origin request door middel van een 'Access-Control-Allow-Origin' header in de response. Een '\*' geeft aan dat alle domeinen een cross-origin request mogen doen. Hier kan ook een domein in worden aangegeven.

Header Key	Value
Access-Control-Allow-Origin	*

Het CORS protocol maakt ook gebruik van een 'preflight' request waarbij er een OPTION-request wordt verstuurd (vóór het eigenlijke request) om vooraf te informeren wat de toegestane communicaties mogelijkheden zijn. Zo kan de server beoordelen of een verzoek veilig is of niet en een verzoek blokkeren waar nodig. Hieronder wordt in een sequencediagram weergegeven hoe dit in zijn werk gaat.

## Client

## Server

Preflight request

```

OPTIONS /doc HTTP/1.1
Origin: https://foo.example
Access-Control-Request-Method: POST
Access-Control-Request-Headers: X-PINGOTHER, Content-type
...

```

```

HTTP/1.1 204 No Content
Access-Control-Allow-Origin: https://foo.example
Access-Control-Allow-Methods: POST, GET, OPTIONS
Access-Control-Allow-Headers: X-PINGOTHER, Content-Type
Access-Control-Max-Age: 86400
...

```

Main request

```

POST /doc HTTP/1.1
X-PINGOTHER: pongpong
Content-Type: text/xml; charset=UTF-8
Origin: http://foo.example
...

```

```

HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.example
Vary: Accept-Encoding, Origin
Content-Encoding: gzip
Content-Length: 235
...

```

Het implementeren van CORS headers is ook van belang voor de webservice. Er zijn verschillende mogelijkheden om dit in Spring Boot te voor elkaar te krijgen. Het is mogelijk de CORS headers afhankelijk van de request endpoint een request methode te implementeren. Wij kiezen er hier voor om dit in één keer globaal te regelen voor alle requests. Dit wordt gedaan in de configuratie fase van Spring Boot met behulp van de `@Configuration` annotate.

```

@Configuration
public class GlobalCorsConfiguration
{
 @Bean
 public WebMvcConfigurer corsConfigurer()
 {
 return new WebMvcConfigurer() {
 @Override
 public void addCorsMappings(CorsRegistry registry) {
 registry.addMapping("/**")
 .allowedOrigins("*")
 .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE");
 }
 };
 }
}

```

CORS speelt ook een rol bij het ontwikkelen van een frontend applicatie die gebruik maakt van een webservice om gegevens in op te slaan of op te halen.

## 10.2 Documentatie

Nu de webservice klaar is is het belangrijk documentatie over het gebruik van de webservice op te stellen. Dit komt neer op een lijst van endpoints met mogelijke HTTP-methoden en de benodigde autorisatie. Van de requests zijn de mogelijke query parameters van belang en de vereiste structuur van de body. Van een response is het met name van belang om de inhoud van de response toe te lichten en ook de mogelijke status codes.

Postman kan hierbij een goed hulpmiddel zijn. Door alle mogelijke requests in een collectie op te nemen kan hiervan een complete API documentatie worden opgesteld.

# NOVI

Make things easier for your teammates with a complete collection description.

## Notes - Users

Make things easier for your teammates with a complete folder description.

### GET users

[Open Request →](#)

```
http://localhost:8080/api/v1/users
```

Make things easier for your teammates with a complete request description.

#### Authorization Basic Auth

Username	<username>
----------	------------

Password	<password>
----------	------------

### GET notes

[Open Request →](#)

```
http://localhost:8080/api/v1/notes
```

Make things easier for your teammates with a complete request description.

#### Authorization Basic Auth

Username	<username>
----------	------------

Password	<password>
----------	------------

### POST add user

[Open Request →](#)

```
http://localhost:8080/api/v1/users
```

Make things easier for your teammates with a complete request description.

#### Authorization Basic Auth

Er zijn ook tools zoals [Swagger](#) die een API documentatie opbouwen op basis van een verzameling unittest.

Tenlotte is het bekijken van [HATEAOS](#) de moeite waard. HATEAOS is een architectuur boven op een RESTful webservice. Hierin zijn allerlei regels opgenomen over hoe een RESTful webservice er volgens HATEAOS er uit zou moeten zien. Al is het voor onze webservice zeker niet nodig om alles volgens HATEAOS te implementeren, een aantal concepten hieruit zijn zeker de moeite waard. Zo wordt bijvoorbeeld in een response naast de domein gegevens ook een lijst van relevante links meegestuurd.

```
HTTP/1.1 200 OK

{
 "account": {
 "account_number": 12345,
 "balance": {
 "currency": "usd",
 "value": 100.00
 },
 "links": {
 "deposits": "/accounts/12345/deposits",
 "withdrawals": "/accounts/12345/withdrawals",
 "transfers": "/accounts/12345/transfers",
 "close-requests": "/accounts/12345/close-requests"
 }
 }
}
```

### 10.3 Deployment

Een Spring Boot Webservice kan op vele manieren worden uitgerold. Tot nu toe maakten we gebruik van de ingebouwde Tomcat server. Dit is niet geschikt voor een serieuze productieomgeving. Maar er zijn veel andere mogelijkheden. [Hier](#) vind je een overzicht.

### 10.4 Tot slot

De in deze cursus geschetste opbouw van een RESTful webservice met Spring Boot is redelijk gangbaar. Toch biedt Spring Boot in combinatie met andere bibliotheken nog vele andere toevoegingen. The sky is the limit. Succes met het verder ontdekken.

## 11. Aanvullend leesmateriaal: Spring Boot 2

Wil je nog meer weten over het Spring Boot framework? Lees dan het boek 'Spring Boot 2'.

## 12. Aanvullend materiaal: Relaties

Na het doorlopen van dit hoofdstuk kun je relaties tussen entiteiten leggen in een Spring Boot applicatie. We kennen drie soorten relaties, namelijk de OneToOne, OneToMany (en ManyToOne, deze twee horen bij elkaar) en de ManyToMany. In de eerste paragraaf leren we over de OneToOne relatie en de annotaties die daarbij horen. In de tweede paragraaf leren we over de OneToMany relatie en hoe deze afwijkt van de OneToOne. In de derde en vierde paragraaf leren we over de ManyToMany relatie en dat deze op twee manieren geïmplementeerd kan worden. Aan het einde van het hoofdstuk vind je nog enkele notities waar je op kunt letten bij het implementeren van de verschillende relaties.

### 12.1 OnetoOne

Een OneToOne relatie leggen we tussen twee entiteiten die elk één attribuut of klasse variabele van de ander heeft. Als voorbeeld gebruiken we een relatie tussen een voetballer en diens schoenen.

```
@Entity
public class Player {
 @Id
 int id;

 @OneToOne(cascade = CascadeType.ALL)
 @JoinColumn(name = "sneakers", nullable = false)
 Shoes shoes;
}

@Entity
public class Shoes {
 @Id
 int id;

 @OneToOne(mappedBy = "shoes")
 Player player;
}
```

De Player heeft één paar Shoes. De belangrijke annotaties zijn:

- @Entity

Deze annotatie geeft aan dat het om een entiteit gaat die in de database opgeslagen kan worden. We kunnen alleen relaties leggen tussen entiteiten. Let er ook op dat beide entiteiten reeds in de database opgeslagen zijn, voordat we de relatie opslaan, anders krijgen we een 'EntityNotFoundException'.

- @Id

Elke entiteit heeft verplicht een primary key nodig. De primary key van Shoes zal gebruikt worden als foreign key in de tabel van Player.

- @OneToOne

Deze annotatie zorgt er voor dat er een foreign key kolom wordt aangemaakt in de klasse tabel. Voor onze Player houdt dat in dat er een "shoes" kolom wordt aangemaakt. De kolom krijgt automatisch de naam van de variabele waar de OneToOne annotatie boven staat. In dit geval wordt de naam uiteindelijk niet "shoes", maar daar komen we zo op.

Verder zien we in de Shoes class dat er (mappedBy = "shoes") achter de annotatie staat. Dit attribuut kan slechts bij één van beide kanten van de relatie staan en geeft aan dat de Shoes klasse de target kant is en Player de owner kant. De owner kant is de kant waar de foreign key in de database wordt opgeslagen. Voor een OneToOne relatie is het arbitrair welke kant de owner kant is.

Als het "mappedBy" attribuut vergeten wordt, worden de twee kanten van de relatie niet aan elkaar gekoppeld en krijgen we dus twee *uni directionele* relaties in plaats van één *bi directionele* relatie.

- @JoinColumn

Deze annotatie is optioneel. De annotatie is bedoeld om de kolom specifiek aan te passen. In ons geval zien we dat we de "name" attribuut en de "nullable" attribuut hebben ingevuld. De "name" attribuut zorgt dat de kolom niet de standaard naam krijgt, maar de string die we aan deze attribuut meegeven. De "nullable" attribuut hebben we op false gezet en daarmee zeggen we dat er geen Player entiteit in de database opgeslagen mag worden zonder diens Shoes (dit is dus database-validatie en is wat anders dan DTO-validatie).

### 12.2 OneToMany – ManyToOne

De @OneToMany en @ManyToOne annotaties zijn horen bij elkaar. Voor het gemak spreken we vanaf hier over een OneToMany relatie, in plaats van een "one-to-many-en-many-to-one-relatie". Een OneToMany relatie is een relatie waarbij een entiteit meerdere relaties met een andere entiteit kan hebben, maar die andere kent maar één van de eerste. Laten we dit weer verduidelijken met een voorbeeld:

```
@Entity
public class Player {
 @Id
 int id;

 @ManyToOne
 @JoinColumn(name = "team_id")
 Team myTeam;
}

@Entity
public class Team {
 @Id
 int id;
```

```
@OneToMany(mappedBy = "team")
List<Player> members;
}
```

We gebruiken hier weer de Player (voetballer) en we maken een relatie met Team (voetbalelftal). We zien hier bijna dezelfde annotaties als bij een OneToOne relatie, alleen gebruikt de Player class nu @ManyToOne en gebruikt de Team class nu @OneToMany.

Verder zien we ook dat de Player een enkel "Team myTeam" heeft met @ManyToOne en dat het Team een "List<Player> members" heeft met @OneToMany. Dit betekent dat een Team meerdere members (Players) kan hebben, maar een Player kan maar één Team hebben. Een OneToMany annotatie staat altijd boven een Collectie (List, Set, Map) en een ManyToOne staat altijd boven een enkele entiteit. Een andere manier om dit te onthouden is dat de annotatie die begint met One staat in de class waar er maar één van is in de relatie en de annotatie die begint met Many staat in de class waar er meerdere van zijn in de relatie.

Een tweede verschil is dat de (mappedBy = "team"), die we hier in de Team class zien staan, niet arbitrair is in deze relatie (bij OneToOne was dat wel arbitrair). Dit betekent dat ManyToOne altijd de owner is en OneToMany altijd de target. De mappedBy moet namelijk altijd bij de OneToMany staan, oftewel boven de List. Het is namelijk niet mogelijk om een lijst op te slaan in een tabel. Een cel in een tabel kan maar één waarde bevatten (een foreign key in dit geval) en dus geen lijst van waardes. Vandaar dat de foreign key in de tabel van de ManyToOne-kant moet worden opgeslagen, daar staat er namelijk maar één in plaats van een lijst.

## 12.3 ManyToMany

Een ManyToMany relatie kunnen we op twee manieren uitvoeren, namelijk met twee @ManyToMany annotaties of met een tussentabel.

In de vorige paragraaf hebben we geleerd dat we een lijst niet kunnen opslaan in de database, maar nu hebben we een relatie met twee lijsten. In deze paragraaf leren we twee manieren hoe we dit toch in de database kunnen opslaan.

```
@Entity
public class Player {
 @Id
 int id;
 @ManyToOne
 @JoinTable(
 name = "player_teams",
 joinColumns = @JoinColumn(name = "team_id"),
 inverseJoinColumns = @JoinColumn(name = "player_id")
)
 List<Club> clubs;
}

@Entity
public class Club {
 @Id
 int id;

 @ManyToMany(mappedBy = "clubs")
 List<Player> players;
}
```

We zien in bovenstaande code de Player class (voetballer), een "List<Club> clubs" met een @ManyToMany annotatie en in de Club class (voetbalvereniging) een "List<Player> players" met, wederom, een @ManyToMany relatie.

De ManyToMany relatie heeft, net als de OneToOne relatie, twee keer dezelfde annotatie. Dat betekent dat het een gelijkwaardige relatie betreft en de mappedBy dus arbitrair geplaatst kan worden. Oftewel: we mogen zelf weten welke class de owner is en welke class de target is. De class waar we de mappedBy zetten, is de target class.

De ManyToMany annotatie is echter niet hetzelfde als de OneToOne annotatie. Waar bij de OneToOne een kolom in de database werd aangemaakt voor de foreign key, is dat bij een ManyToMany relatie natuurlijk niet mogelijk omdat we twee lijsten hebben. Wanneer we een @ManyToMany annotatie gebruiken, maakt Spring achter de schermen voor ons een complete tabel aan: een tussentabel. Deze tussentabel bevat twee kolommen: de foreign key van de owner en de foreign key van de target.

Een voorbeeld van een tussentabel:  
[TABEL INVOEGEN]

Je ziet in deze tabel het ManyToMany aspect terug in club 3005 en player 1004. In de "player\_id" kolom staat twee keer 1004 en in de "team\_id" kolom staat twee keer 3005. Eén van die 1004 en 3005 staan in dezelfde rij en vormen samen dus een relatie met elkaar. De andere 3005 en 1004 staan beide in een rij met een ander id (1002-3005 en 1004-3006). Zo heeft dus zowel 1004 als 3005 meerdere relaties, oftewel many to many.

Als laatste zien we in de Player class nog het volgende stukje code:

```
@JoinTable(
 name = "player_teams",
 joinColumns = @JoinColumn(name = "player_id"),
 inverseJoinColumns = @JoinColumn(name = "team_id")
)
```

Deze @JoinTable annotatie is optioneel. We vullen in deze annotatie drie attributen in: het name-attriboot, de joinColumns en de inverseJoinColumns.

- Met het name-attriboot kunnen we de naam van de tussentabel bepalen. Dit is met name handig voor je data.sql bestand, waarin we deze tabel bij naam moeten aanspreken om het te kunnen vullen. De standaard naam is "tabel1naam\_tabel2naam".
- Met het joinColumns-attriboot kunnen we de kolom specificaties van de owner kolom regelen. We gebruiken hiervoor de eerder besproken @JoinColumn annotatie.
- Met het inverseJoinColumns-attriboot kunnen we de kolom specificaties van de target kolom regelen. We gebruiken hiervoor de eerder besproken @JoinColumn annotatie.

De @JoinTable annotatie staat altijd aan de owner kant.

## 12.4 Tussentabel

In sommige gevallen maakt Spring geen automatische tabel aan. In dat geval maak je gebruik van de tussentabel. Hierover leer je in deze paragraaf meer.

We kunnen een ManyToMany ook implementeren met een expliciete tussentabel. Dit is met name nuttig als je een extra attribuut aan de tussentabel wilt toevoegen, zoals bij een rating. In onderstaand voorbeeld zien we hoe we dit kunnen implementeren als we spelers de mogelijkheid willen geven om een club (hun eigen, die van de tegenstander, of een willekeurige club) een cijfer te geven.

```
@Entity
public class Club {
 @Id
 int id;

 @OneToMany(mappedBy = "club")
 Set<ClubRating> ratings;
}

@Entity
public class Player {
 @Id
 int id;

 @OneToMany(mappedBy = "player")
 Set<ClubRating> ratings;
}

@Entity
public class ClubRating {

 @Id
 long id;

 @ManyToOne
 @JoinColumn(name="club_id")
 Club club;

 @ManyToOne
 @JoinColumn(name = "player_id")
 Player player;
}
```

```
int rating;
}
```

We zien hier dat we drie klassen nodig hebben. We hebben weer een Club en een Player, maar nu ook een ClubRating. In Club en Player zien we beide een "List<ClubRating> ratings" met @OneToMany er boven. We zien dat voor beide klassen de tegenhanger, de @ManyToOne, in de ClubRating class ligt. We hebben hier dus een vergelijkbare tussentabel gemaakt als wat de @ManyToMany annotatie voor ons doet. Het enige verschil is dat onze eigen tussentabel vier kolommen heeft, waar de automatisch gegenereerde tussentabel slechts twee kolommen had. Onze eigen tussentabel is namelijk een entiteit (en heeft ook een @Entity annotatie), dus moet het ook een primary key hebben. Ook hebben we een "rating" kolom toegevoegd aan onze eigen tabel, zodat we het cijfer van een speler voor deze club kunnen opslaan in deze kolom.

## 12.5 Notities

Bij het toepassen van relaties zijn een paar dingen belangrijk om op te letten.

- Een relatie hoeft niet altijd ingevuld te worden. Met bijvoorbeeld een @OneToMany annotatie, leg je de mogelijkheid tot een relatie. Het kan zo zijn dat niet alle instanties van je entiteit deze specifieke relatie nodig hebben.
- Het kan zijn dat je een relatie correct geïmplementeerd hebt, maar bij je GET response toch niet te zien krijgt dat het object de relatie bevat. Je verwacht bij player bijvoorbeeld een shoes attribuut in de JSON, maar je ziet enkel een id-attribuut. In dat geval heb je waarschijnlijk in PlayerDto geen variabele van ShoesDto toegevoegd.
  - a. Dit is de code die je verwacht:

```
{
 "id" : 1001,
 "shoes" : {
 "id" : 2001
 }
}
```

a. Maar dit is de code die je werkelijk krijgt:

```
{
 "id" : 1001
}
```

- Als je wel het attribuut in je JSON response te zien krijgt, maar de waarde is null, dan heb je waarschijnlijk vergeten om in je service class de waarde van de relatie in je DTO in te vullen:
  - a. Dit is de code die je verwacht:

```
{
 "id" : 1001,
 "shoes" : {
 "id" : 2001
 }
}
```

a. Maar dit is de code die je werkelijk krijgt:

```
{
 "id" : 1001,
 "shoes" : null
}
```

- Het is gebruikelijk om een aparte PUT mapping te maken voor het toevoegen van een relatie.
- Zorg dat je in de service methode altijd de entity opslaat na het toevoegen van de relatie.
- Zorg dat de relatie die je toevoegt ook opgeslagen is in de database, anders krijg je een 'EntityNotFoundException'.

## 13. Cursusevaluatie - Spring Boot

Je hebt zojuist de cursus Spring Boot afgerond. We zijn benieuwd wat je ervan vond! Om ervoor te zorgen dat wij continue kunnen blijven verbeteren, vragen we jou om deze cursus (de content, lessen en docent) te evalueren. Dit kan via:

[Evaluatie - Cursus Spring Boot](#)

Het invullen van de vragenlijst kost je 1 à 2 minuten en is anoniem. Dankjewel!