Werkboek TICT-V1WAC-15

Webapplication Construction

Propedeuse HBO-ICT

Cursuseigenaar: [Bart.vanEijkelenburg@hu.nl](mailto:Bart.vanEijkelenburg@hu.nl)

Auteur: Bart van Eijkelenburg (SIE)

Versie: 4.1 (13-04-2019)

Versiebeheer

|  |  |  |  |
| --- | --- | --- | --- |
| # | Wijzigingen | Door: | Datum: |
| 4.1 | Les 1-4 nagelopen voor studiejaar 2018/2019. | B. van Eijkelenburg | 13-04-2019 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Inhoud

[Inleiding & Tooling 4](#_Toc514957525)

[Les 1, practicum: Sockets, Clients & Servers 6](#_Toc514957526)

[Les 2, practicum: JEE, IDE, Tomcat & Maven 12](#_Toc514957527)

[Les 3, practicum: Webservices (Jersey & Maven) 25](#_Toc514957528)

[Les 4, practicum: JavaScript 28](#_Toc514957529)

[Les 5, practicum: OpenWeatherMap, IP-API & WorldResources 30](#_Toc514957530)

[Les 6, practicum: HTML5 Web Storage & CSS Grid Layout 34](#_Toc514957531)

[Les 7, practicum: PostgreSQL Database 36](#_Toc514957532)

[Les 8, practicum: RESTful Webservices & Relational Databases 40](#_Toc514957533)

[Les 9, practicum: Authentication, Authorization & HTTPS 41](#_Toc514957534)

[Les 10, practicum: Deployment op Heroku 46](#_Toc514957535)

[Bijlage 1: Code-snippets 53](#_Toc514957536)

[Bijlage 2: Command-line instructies 56](#_Toc514957537)

# Inleiding & Tooling

Dit werkboek bevat de opdrachten die behoren bij de cursus TCIF-V1WAC-15. De opdrachten zijn enerzijds gericht op het begrijpen van, en kunnen omgaan met een deel van het Java/Jakarta Enterprise Edition (JEE) Platform, en anderzijds om te leren gaan met concepten (o.a. REST) en tools (o.a. Tomcat, Git). Per week kan verschillen waar de nadruk op ligt.

Er zijn bij deze cursus alleen practicumopdrachten. De stof in de opdrachten maakt deel uit van de **tentamenstof**. Elke **les** kent 1 of meer (voor ieder) verplichte practicumopdrachten. Practicumopdrachten doen veelal een beroep op de bij TICT-V1OODC-15 behandelde stof.

Een voorbeeld hiervan is het toepassen van klassen en het maken van objecten. Bij de cursus TICT-V1OODC-15 is al behandeld hoe een klasse gemaakt moet worden, wat methoden zijn en hoe je objecten kunt creëren. Er zullen dan ook UML-diagrammen de revue passeren waarbij van de student verwacht mag worden dat hij of zij zelfstandig kan achterhalen hoe dat gerealiseerd moet worden.

Tijdens blok 4 lopen wederom slechts 2 cursussen naast elkaar. Daarom mag van de studenten een grotere inzet verwacht worden dan bij cursussen die doorlopen in de projectweken. De cursus beslaat 5 ECTS, wat staat voor 5 x 28 = 140 uur aan studentinzet. Over 6 weken verdeeld is dat 23,33 uur per week. Dat is niet geheel realistisch omdat er met 2 cursussen dus meer dan fulltime gestudeerd zou moeten worden. Een inzet van 18 uur mag echter wel gevraagd worden. Dan blijft er voldoende tijd over voor het individuele project en de parallel-cursus Data Processing. Een gemiddelde weekinzet is hieronder opgenomen:

* 2x lesvoorbereiding à 1,5 uur = 3 uur
* 2x college à 3 uur = 6 uur
* 2 SIE-spreekuur = 2 uur
* Huiswerk: practicum = 7 uur

Dit komt uit op een gemiddelde van 18 uur. Uiteraard zal de ene student meer of juist minder inzet moeten tonen dan de andere student om toch hetzelfde te bereiken. De spreekuren kunnen gekoppeld worden aan de werkcolleges, maar de docent kan besluiten dit anders in te richten, door bijvoorbeeld een online/virtueel spreekuur op een ander moment te houden.

## Tooling

De opdrachten van deze cursus geven regelmatig instructies om bepaalde handelingen uit te voeren in een IDE. Je bent echter niet verplicht om van de aanbevolen IDE gebruik te maken. **Kies je voor andere tools dan beschreven, dan moet je er wel rekening mee houden dat de docent je bij eventuele problemen niet altijd kan helpen.** Naast de IDE zullen er soms extra tools worden gebruikt, welke worden geïntroduceerd tijdens de lessen.

### Eclipse en alternatieven

De docent zal in ieder geval ondersteuning bieden bij het gebruik van Eclipse. Dit werkboek zal, als er sprake is van Eclipse-specifieke instructies, ook zoveel mogelijk een command-line alternatief geven zodat je ook met een andere IDE kunt werken. Ook met Eclipse ontkom je mogelijk niet altijd aan het uitvoeren van een enkel commando via de command-line, maar dat zal maar af en toe zijn.

### Tooling Opties

Kies je voor safe, ondersteuning en weinig command-line commando’s, gebruik dan Eclipse. Wil je meer ervaring opdoen met de command-line, dan kan je dat ook doen met Eclipse of andere IDE naar keuze.

**Aanbevolen**

|  |  |
| --- | --- |
| http://www.eclipse.org/downloads/images/javaee.png | Afbeeldingsresultaat voor commandline logo |
| Eclipse  +  command-line (**soms**) | Eigen IDE  +  command-line (**veel**) |

# Les 1, practicum: Sockets, Clients & Servers

## Les 1: Practicum 1: Client & Server

**Werk voor deze opdracht in tweetallen!** Student 1 laat de klasse Server uit de slides van les 1 draaien op zijn eigen laptop, en student 2 doet hetzelfde met klasse Client. **Let op:** in klasse Client moet **"localhost"** vervangen worden door het IP-adres van de server (de laptop van student 1). Zorg dat beiden op hetzelfde draadloze netwerk zijn aangesloten! Het IP-adres van je laptop kan je vinden door het commando ipconfig (Windows) of i**f**config (Linux / MAC OSX) in een terminal (Windows: ‘cmd.exe’) uit te voeren. Zoek de adapter voor je draadloze (wireless) verbinding en noteer het IP(v4)-adres.

**Let op:** het kan zijn dat op de server-laptop bijvoorbeeld een firewall ervoor zorgt dat de client geen contact kan maken. Je kunt je firewall zo instellen dat de gekozen poort (4711) wel verkeer doorlaat, of je kunt je firewall tijdelijk even uitschakelen.

🡺 Maak als client verbinding met de server en verstuur een bericht (dat de server uitprint)!

## Les 1: Practicum 2: PrintWriter & Scanner

**Werk voor deze opdracht in tweetallen!** Wijzig de code van practicum 1, zodat klasse Client een PrintWriter gebruikt om tekst te printen in plaats van (alleen) een OutputStream. Op die manier hoef je niet steeds achter elke string getBytes() aan te roepen. Zoek in de [documentatie](https://docs.oracle.com/javase/9/docs/api/java/io/PrintWriter.html) van klasse PrintWriter welke constructor je het beste kunt gebruiken om een PrintWriter te maken. Roep methode flush() aan na het printen van een tekst!

Pas de code van de klasse Server aan zodat je gebruik maakt van een Scanner in plaats van (alleen) een InputStream. Zoek in de [documentatie](https://docs.oracle.com/javase/9/docs/api/java/util/Scanner.html) van klasse Scanner welke constructor je in dit geval het beste kunt gebruiken om een Scanner te maken! Maak gebruik van de methoden hasNextLine() en nextLine() van de klasse Scanner!

🡺 Maak samen beide opdrachten en test of de applicatie nog steeds werkt!

## Les 1: Practicum 3: Gebruikersinvoer

**Werk voor deze opdracht in tweetallen!** Breid klasse Client uit zodat je de gebruiker invoer laat geven via het toetsenbord, en dit naar de server stuurt. Dit kan door een Scanner aan het toetsenbord te koppelen (OODC, les 10). Zorg dat het sturen van berichten doorgaat tot de gebruiker ‘stop’ invoert. De server print de berichten totdat de client de verbinding verbreekt.

Na het verbreken van de verbinding zal de server ook worden afgesloten. Om dit te voorkomen kan je de server-code in een oneindige while-loop plaatsen, zodat de client iedere keer opnieuw uitgevoerd kan worden, zonder dat je de server hoeft te herstarten.

🡺 Wijzig samen beide klassen zoals beschreven en test of de applicatie nog steeds werkt!

## Les 1: Practicum 4: Response!

**Deze opdracht is individueel, zorg dat je klasse Server van practicum 3 op je laptop hebt!**

De server van opdracht 3 kan ook berichten van andere clients ontvangen. Kijk wat er gebeurt als je die server start en in je browser <http://localhost:4711> opvraagt (voor deze link moet de server op poort 4711 draaien). Controleer wat je server nu uitprint!

Als het goed is zie je in de console van Eclipse iets wat lijkt op de volgende tekst:

1: GET / HTTP/1.1

2: Host: localhost:4711

3: User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Firefox/45.0

4: Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

5: Accept-Language: nl,en-US;q=0.7,en;q=0.3

6: Accept-Encoding: gzip, deflate

7: DNT: 1

8: Connection: keep-alive

9:

Dit is een HTTP-request (verzoek), waar we de volgende les dieper op ingaan. Na het versturen van het request wacht de browser op antwoord (HTTP-response). Jouw server geeft dat nog niet, en daarom lijkt het in de browser alsof de pagina niet geladen kan worden.

Een tweede probleem is dat de browser de verbinding met de server open houdt tijdens het wachten op antwoord. De server weet daardoor niet wanneer de browser klaar is met het sturen van het request. Men heeft daarom in het HTTP-protocol afgesproken dat een lege regel het einde vormt van een HTTP-request.

Dus zodra de server in een request een lege regel tegenkomt (hierboven #9), weet deze dat het request ten einde is. De server moet dan de while-loop voor het lezen van request-regels afbreken, en een response (string) in de OutputStream van de socket schrijven. Daarna kan de socket gesloten worden. Een voorbeeld-response staat hieronder; regelnummers zijn **geen** onderdeel van een response, maar newlines ("\r\n") na iedere (lege) regel tekst, **wel**!

1: HTTP/1.1 200 OK

2:

3: <h1>It works!</h1>

🡺 Zorg dat jouw klasse Server de bovenstaande response geeft, en test het in je browser.

* Java kent ‘\n’ als newline. Binnen Java-programma’s werkt dat prima. Daarbuiten ziet een newline er soms anders uit. Op Windows machines is een newline bijvoorbeeld ‘\r’ + ‘\n’ (carriage return + line feed). Voor Linux is alleen ‘\n’ al voldoende. Sommige editors kunnen met beide varianten omgaan (Notepad++), maar niet alle programma’s zijn zo flexibel (Windows Notepad). Omdat het HTTP-protocol voor communicatie tussen applicaties op diverse platformen is, heeft men een HTTP-newline gedefinieerd als carriage return + line feed. Daarom neem je in je code expliciet “\r\n” op als je een HTTP-newline wilt.

## Les 1: Practicum 5: MyServlet

De Server-klasse kan na practicum 3 en 4 meerdere requests afhandelen en beantwoorden. Echter, dat doet deze server nog steeds één-voor-één. Maar een server voor een website moet zeer veel requests tegelijk kunnen afhandelen. Daarom is het belangrijk dat de afhandeling hiervan niet sequentieel (na elkaar) gaat, maar parallel (tegelijkertijd).

We kunnen de code in klasse Server die voor elk request doorlopen moet worden, in een aparte thread laten draaien. Op deze manier kan die code meerdere keren **tegelijkertijd** worden uitgevoerd. We gaan hiervoor de klasse Server aanpassen en ook klasse MyServlet schrijven. Zie het UML-diagram:



* Klassen die requests kunnen verwerken en beantwoorden, worden in de Java-wereld vaak servlets genoemd. Dit zijn relatief kleine klassen en ‘serveren’ een antwoord op het request.

Deze Server bevat **alleen** code om nieuwe socket-verbindingen te ontvangen (met behulp van een ServerSocket). Voor ieder nieuw request, maakt de server een nieuw MyServlet-object, en geeft de nieuw geopende Socket als parameter mee aan de constructor van MyServlet.

De constructor van MyServlet slaat deze nieuwe verbinding in attribuut ‘socket’ op (zoals je van constructors in blok 3 gewend bent). Na het maken van een MyServlet-object start de server de MyServlet-thread en gaat weer wachten op nieuwe socket-verbindingen.

Methode run() van MyServlet verwerkt het nieuwe request. Alle code uit de klasse Server die je daarvoor al geschreven hebt, verplaats je dus naar deze methode!

🡺 Schrijf de klasse MyServlet, wijzig de klasse Server en test in je browser of alles nog werkt!

## Les 1: practicum 6: Paralleltest

Als je practicum 5 goed hebt uitgevoerd, kan je server meerdere requests tegelijk (parallel) verwerken. Dat moet ook, zeker als je applicatie honderden requests tegelijkertijd zou ontvangen. Dit zouden we kunnen simuleren met een andere Java-applicatie, maar dat is voor nu nog wat te ingewikkeld.

Daarom testen we de server door deze vanuit twee browservensters ‘tegelijk’ aan te roepen. Maar omdat de processor veel sneller werkt dan een mens, zouden we het verschil tussen parallelle en seriële verwerking niet merken. Immers, als je de server vanuit venster 1 aanroept, en daarna snel vanuit het venster 2, dan is de server alsnog allang klaar met het beantwoorden van het eerste request voordat je venster 2 hebt geopend. Daarom moet je voor deze test de methode run() van MyServlet vertragen, door op de laatste regel van de methode run() de code 10 seconden te laten ‘slapen’.

Als je de *server-met-vertraging* snel achter elkaar vanuit twee verschillende browservensters benadert, dan moet in beide vensters direct een resultaat op het scherm komen. In dat geval handelt de server beide requests namelijk parallel af. Als je in het tweede venster ongeveer 10 seconden moet wachten voor je reactie krijgt van de server, dan behandelt je server de requests dus serieel (achter elkaar), en is deze nog niet correct geprogrammeerd.

* Werkt je server serieel, dan voer je de methode run() mogelijk niet als aparte thread uit.

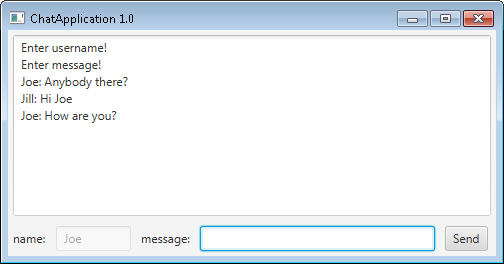
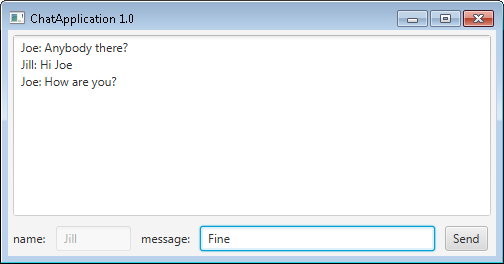
🡺 Creëer met Thread.sleep() 10 seconden vertraging in methode run() van MyServlet!

🡺 Benader de server direct achter elkaar met twee browservensters en test je server.

## Les 1: practicum 7: ChatClientApp (**optioneel**)

**Deze opdracht is niet verplicht, wel aanbevolen! Het is wel zo leuk om ook deze opdracht in tweetallen uit te voeren maar ook dat is niet verplicht.**

We maken in deze opdracht een chat-applicatie. Als je een bericht stuurt naar een andere gebruiker, dan stuur je eigenlijk een bericht naar de server, die dat bericht vervolgens weer doorstuurt naar andere gebruikers. De server is al gegeven op de volgende bladzijde. Hier moet nog een ChatClient (GUI) bij gemaakt worden. Hiervoor is een UML-diagram gegeven.

Klasse ChatClient is een JavaFX GUI en daarom een subklasse van Application. Er zijn diverse attributen, waaronder GUI-controls (TextArea, TextField etc.) en, **belangrijk**, de Socket waarmee verbinding met de server onderhouden kan worden.

De constructor maakt verbinding met de server en start een MessageListener (zie volgende bladzijde).

Methode start(..) moet het scherm opbouwen en de controls plaatsen. Zorg dat methode sendMessage() wordt uitgevoerd als je op de verzendknop klikt,.

Methode sendMessage() stuurt het ingevoerde bericht naar de server, maar alleen als er verbinding is en als er een gebruikersnaam en bericht is ingevoerd. Is hieraan niet voldaan, dan moet er een melding in de chatBox getoond worden. Na versturen van het eerste bericht mag de gebruikersnaam niet meer wijzigen, het chatName-TextField moet *disabled* worden.

Het kan natuurlijk zijn dat iemand juist een bericht verstuurt als jij nog aan het typen bent. Dit bericht moet dan direct in jouw scherm getoond worden. We hebben dus een apart proces nodig dat luistert of er van de server nieuwe berichten binnenkomen. Deze berichten moeten dan in de chatBox getoond worden. Klasse MessageListener is daarom een subklasse van Thread en het enige wat methode run() doet, is (met bijv. een while-lus en een Scanner) luisteren naar binnenkomende berichten. Een nieuw bericht wordt meteen toegevoegd aan de TextArea.

**Let op:** de MessageListener moet toegang hebben tot de chatBox en de verbinding met de server. Daarom is het handig om de MessageListener als een innerclass op te nemen in ChatClient. Zie de slides van les 1 voor een voorbeeld.

* Doordat je een apart proces start, blijft er verbinding met de server bestaan, ook al sluit je de GUI. Als je wilt dat jouw applicatie dan ook geheel afsluit, moet je zorgen dat een close-request (een klik op de sluitknop van je scherm) ook de verbinding met server sluit. Je kunt hiervoor de methode setOnCloseRequest(..) van klasse Stage gebruiken.

**import** java.io.\*; **import** java.net.\*; **import** java.util.\*;

**public** **class** Server {

**public** **static** **void** main(String[] args) **throws** IOException {

ArrayList<PrintWriter> allClients = **new** ArrayList<PrintWriter>();

**while** (**true**) {

**try** (ServerSocket ss = **new** ServerSocket(8080)) {

Socket s = ss.accept();

**new** Thread() {

**public** **void** run() {

PrintWriter pw = **null**;

**try** (Scanner scanner = **new** Scanner(s.getInputStream())) {

allClients.add(pw = **new** PrintWriter(s.getOutputStream()));

**while** (scanner.hasNextLine()) {

String message = scanner.nextLine();

**for** (PrintWriter printer : allClients) {

printer.println(message);

printer.flush();

}

}

} **catch** (IOException ioe) {

ioe.printStackTrace();

} **finally** { allClients.remove(pw); }

}

}.start();

}

}

}

}

🡺 Implementeer het gegeven UML-klassendiagram volgens de beschrijving!

# Les 2, practicum: JEE, IDE, Tomcat & Maven

In deze les werk je in de eerste vier practicumopdrachten stap voor-stap het Dynamic Example uit welke in de les is toegelicht. Vervolgens zal deze applicatie op Tomcat gedeployed worden. Door deze applicatie niet kant en klaar aan te bieden, maar zelf deze stappen uit te voeren zal je beter begrijpen hoe een webapplicatie in elkaar steekt.

**Practicum 1: Installatie en configuratie van IDE & Server  
Practicum 2: Creëren van een Maven project  
Practicum 3: Bouw van Dynamic Example  
Practicum 4: Deployment op Tomcat**

In dit werkboek wordt steeds het gebruik van de [aanbevolen](#_Tooling_Opties)tooling (Eclipse EE) beschreven. Kies je voor andere tools, dan zal regelmatig worden verwezen naar de bijlagen waarin beschreven is wat je moet doen als je geen Eclipse EE gebruikt. Het kan soms zo zijn dat hiervoor zelf meer moet uit- of opzoeken.

* Liever je eigen IDE? Installeer die IDE en [Tomcat 9](http://tomcat.apache.org/download-80.cgi). Ga daarna verder met **Practicum 2**.

Voor sommige opdrachten zal na practicum 1 regelmatig een command-line alternatief gegeven worden. **Let op;** soms is het een alternatief voor een enkele stap, soms voor een heel practicum. Dit staat steeds aangegeven.

## Les 2: practicum 1: Tooling - Installatie & Configuratie

**Practicum preconditie:** Geïnstalleerde Java SDK / JDK 1.8 of hoger

**Eclipse homepage:** <http://www.eclipse.org/> **Tomcat homepage:** <http://tomcat.apache.org/>

**Stap 1:**

Als je Eclipse EE nog niet hebt, download dan [hier](http://www.eclipse.org/downloads/eclipse-packages/) de “IDE for Java EE Developers” package. Pak het zip-bestand uit op een voor jou handige locatie (bij voorkeur de root-directory).

**Stap 2:**

Ga naar de Tomcat 9 [downloadpagina](https://tomcat.apache.org/download-90.cgi), en download van de “Binary distributions” de “Core” variant in zip-formaat. Pak het zip bestand uit op een voor jou handige locatie. Het is handig om Tomcat niet helemaal weg te werken in de negende subdirectory van een directory op de C:\ schijf. Je moet er namelijk af en toe naar toe verwijzen. In de voorbeelden in dit document staat Tomcat direct op de C:\ schijf: C:\apache-tomcat-9.0.6.

**Stap 3**

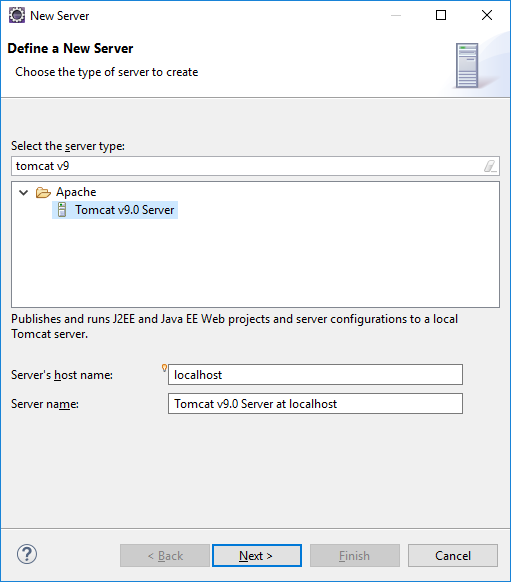
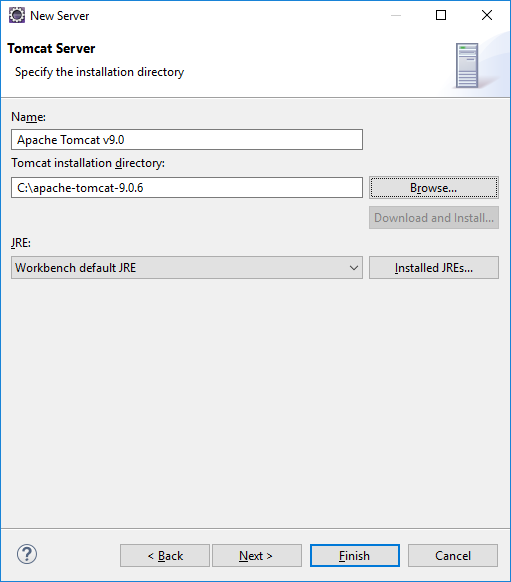
Open Eclipse **(**start eclipse.exe). Als je Eclipse EE net hebt geïnstalleerd, zal deze vragen welke “Workspace” je wilt gebruiken. Maak op een handige plaats een nieuwe directory aan voor al je toekomstige projecten of kies de map die Eclipse voorstelt. Klik **OK**.

**Stap 4**

Webapplicaties moeten straks via een server voor de buitenwereld te bereiken zijn. Die server moeten we eerst instellen. We gebruiken daarvoor dus de Tomcat-server. Open via menu **Window**, **Show View**, **Other** de “Servers” view (zoek evt. op “servers” in de wizard).

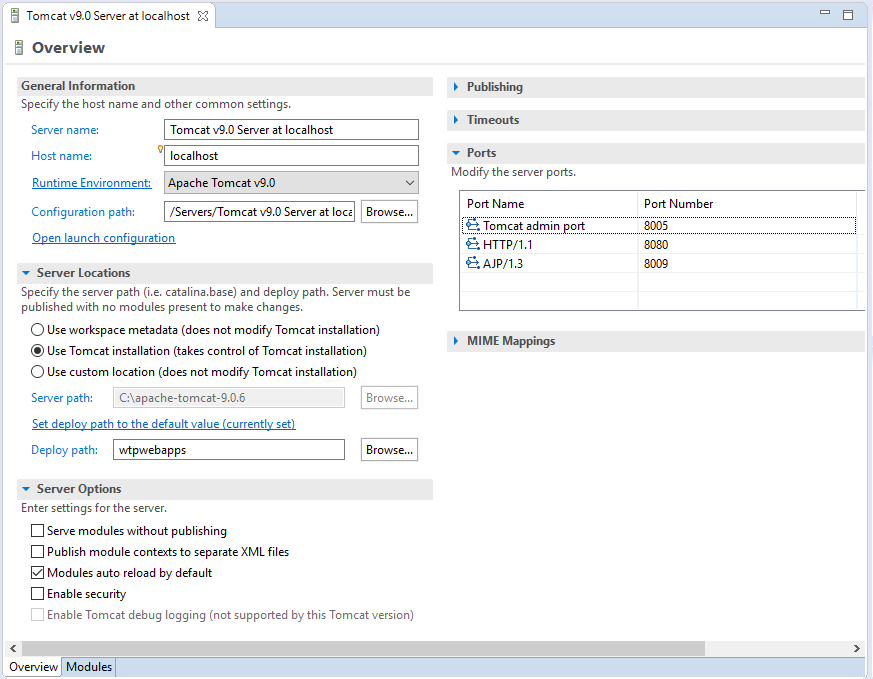
**Stap 5**

Rechtermuisklik in de nieuwe view en kies voor **New**, **Server**. Kies voor een Tomcat v9.0 server en kies **Next**. Selecteer daarna de directory waar je Tomcat hebt uitgepakt en kies nogmaals **Finish**. Zie de screenshots hieronder:

**Stap 6**

Voordat we de applicatie gaan bouwen wijzigen we eerst hoe Eclipse gebruik maakt van Tomcat. Dubbelklik op de zojuist toegevoegde server in de Servers-view. Selecteer onder “Server Locations” de optie “Use Tomcat installation (takes control of Tomcat installation)”.

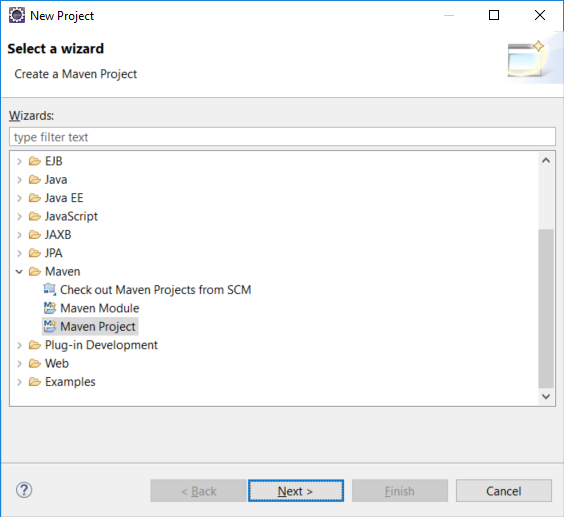
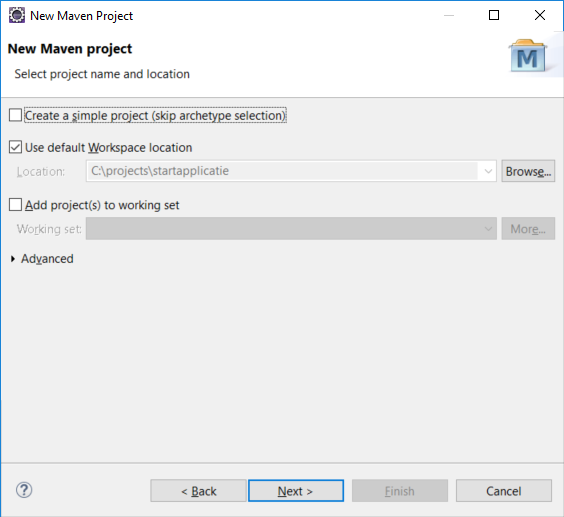


Sla de wijzigingen op en sluit het serverconfiguratie-venster.

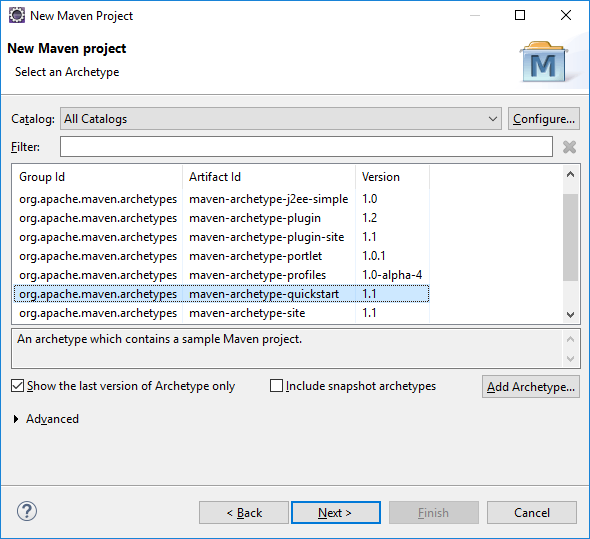
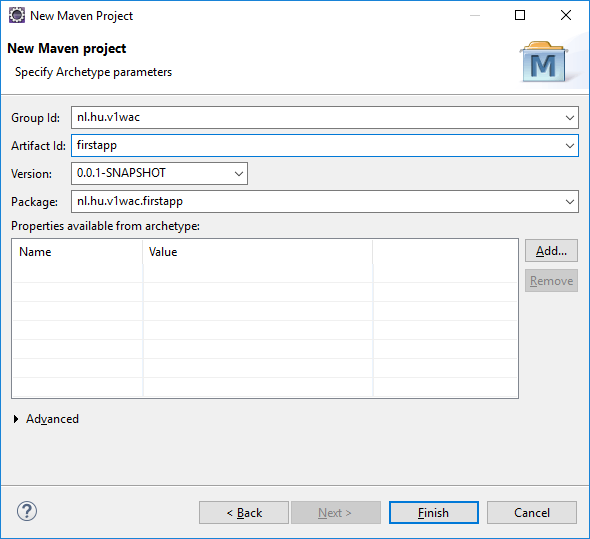
## Les 2: practicum 2: Creëren van een Maven project

**Stap 1** (**command-line alternatief *voor deze stap*: klik** [**hier**](#_Creëren_van_een))

Maak een nieuw project via het menu **File** > **New** > **Project**. Klap daarna het mapje **Maven** open en kies voor **Maven Project**, klik **Next**. Klik in het volgende scherm nogmaals **Next**:

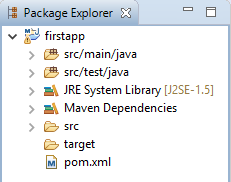
 

Hierdoor zal project aangemaakt worden in de Eclipse-workspace. In het volgende scherm van de wizard (screenshot linksonder) geef je het Maven archetype op. Dat bepaalt o.a. de projectstructuur (directories etc.) van het project. Selecteer het quickstart-archetype met Artifact Id: **maven-archetype-quickstart** en kies **Next**.

In het laatste scherm geef je het Group- en Artifact id op (hier **nl.hu.v1wac** en **firstapp**). Dit beschrijft je applicatie en vormt de packagestructuur voor jouw Java-bestanden. Klik **Finish**.

* De Java-SE [documentatie](http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html) beschrijft conventies voor package-namen. Aanvullende Nederlandstalige uitleg is te vinden op [Wikipedia](http://nl.wikipedia.org/wiki/Java_package#Naamconventie_voor_een_package).

**Stap 2**

De structuur van het project is hiernaast weergegeven.

jouw Java-broncode bestanden

jouw testcode

compiler die Eclipse gebruikt

libraries van anderen die je in jouw applicatie gebruikt of nodig hebt

alle bronbestanden (naast Java-bestanden ook HTML/Javascript/images etc.)

bevat het resultaat van het build-proces (compileren/testen etc.)

project build configuratie

Het bestand **pom.xml** bevat informatie over jouw project. Je kunt het project zo ook buiten Eclipse compileren. Eclipse gebruikt voor compilatie de M2Eclipse Plugin. Deze is ingesteld op Java 1.5 compatibility; we wijzigen dat naar **Java 1.8** (alle tools ondersteunen deze versie)!

Open hiervoor pom.xml (klik in Eclipse EE daarna onderin de editor op “pom.xml” om de XML-tekst te zien). Voeg vervolgens de onderstaande compiler-properties toe:

<project ...>

...

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

**<maven.compiler.source>1.8</maven.compiler.source>**

**<maven.compiler.target>1.8</maven.compiler.target>**

</properties>

De vorm waarin de applicatie gedistribueerd wordt, staat nog op ‘jar’ (Java ARchive), maar moet ‘war’ (Web ARchive) worden. Wijzig ook meteen de versie van je project:

<project ...>...

<version>1.0</version>

<packaging>war</packaging>

Sla pom.xml op, rechtermuisklik op de projectnaam (firstapp); kies **Maven** > **Update Project**.

**Stap 3**

Het project mist nu nog het bestand **web.xml** (Eclipse geeft een foutmelding). Je kunt Maven zo configureren dat dit bestand niet nodig is, maar we hebben web.xml later nog nodig. Rechtermuisklik in de directory src op de folder **main**. Kies **New** > **File**. Kies als parent folder **‘firstapp/src/main/webapp/WEB-INF’**, en als bestandsnaam **web.xml**. De directorystructuur **MOET EXACT** zijn zoals aangegeven (m.u.v.) projectnaam ‘firstapp’!

Kopieer onderstaande voorbeeld in web.xml:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee web-app\_4\_0.xsd"*

version=*"4.0"*>

</web-app>

## Les 2: practicum 3: Bouw van Dynamic Example

**Stap 1**

Maak in de src/main/webapp folder een nieuw HTML-bestand aan. Rechtermuisklik op “webapp” en kies **New**, **File**. Kies als bestandsnaam “index.html”. Voeg daar vervolgens de volgende HTML code aan toe:

<!DOCTYPE html>

<html>

<title>Dynamic Example</title>

<body>

<form action=*"DynamicServlet.do"* method=*"get"*>

<div>

<input type=*"text"* name=*"username"* />

<input type=*"submit"* value=*"Go!"* />

</div>

</form>

</body>

</html>

Zoals je in de form-tag kunt zien zal de informatie die de gebruiker kan invoeren in het tekstveld met name “username”, via een GET-request verzonden worden naar de URL “**DynamicServlet.do**”. Die servlet bestaat nog niet, maar gaan we in stap 2 maken.

**Stap 2**

Verwijder klasse **App.java** in de package **nl.hu.v1wac.firstapp**. Rechtermuisklik vervolgens op deze package enklik **New** > **Class**. Noem de klasse “DynamicServlet” en wijzig de package in nl.hu.v1wac.firstapp.**servlets**, klik **Finish**. Kopieer deze code in de nieuwe klasse:

**package** nl.hu.v1wac.firstapp.servlets;

**import** java.io.\*;

**import** javax.servlet.ServletException;

**import** javax.servlet.http.\*;

**public** **class** DynamicServlet **extends** HttpServlet {

**protected** **void** doGet(HttpServletRequest req, HttpServletResponse resp)

**throws** ServletException, IOException {

String name = req.getParameter("username");

PrintWriter out = resp.getWriter();

resp.setContentType("text/html");

out.println("<!DOCTYPE html>");

out.println("<html>");

out.println(" <title>Dynamic Example</title>");

out.println(" <body>");

out.println(" <h2>Dynamic webapplication example</h2>");

out.println(" <h2>Hello " + name + "!</h2>");

out.println(" </body>");

out.println("</html>");

}

}

**Stap 3**

Eclipse geeft nu compiler-fouten. Dat komt omdat de klassen die worden geïmporteerd, geen onderdeel zijn van Java SE. We hebben daarvoor extra Java EE libraries nodig. Die kan je aan je project koppelen door een dependency aan de pom.xml toe te voegen.

<project ...>

...

<dependencies>

**<dependency>**

**<groupId>javax.servlet</groupId>**

**<artifactId>javax.servlet-api</artifactId>**

**<version>4.0.0</version>**

**<scope>provided</scope>**

**</dependency>**

Tomcat heeft deze library al. Met de scope ‘provided’ geef je aan dat deze library daarom niet in het uiteindelijke war-bestand hoeft te worden gekopieerd. Als je pom.xml opslaat wordt de library gedownload van een Maven [repository](https://mvnrepository.com/). De foutmeldingen verdwijnen.

**Stap 4**

Nu moeten we ervoor gaan zorgen dat de URL “/DynamicServlet.do” ook daadwerkelijk bij onze servlet uitkomt. Dat kan door een annotatie toe te voegen aan de klasse…:

@WebServlet(urlPatterns = "/DynamicServlet.do")

**public** **class** DynamicServlet ...

Vergeet hierbij niet om de annotatie te importeren (CTRL+SHIFT+O)! Een **alternatief** is om in het bestand web.xml aan de web-app tag de volgende mapping toevoegen:

<web-app ...>

<servlet>

<servlet-name>Dynamic Example Servlet</servlet-name>

<servlet-class>nl.hu.v1wac.firstapp.servlets.DynamicServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Dynamic Example Servlet</servlet-name>

<url-pattern>/DynamicServlet.do</url-pattern>

</servlet-mapping>

</web-app>

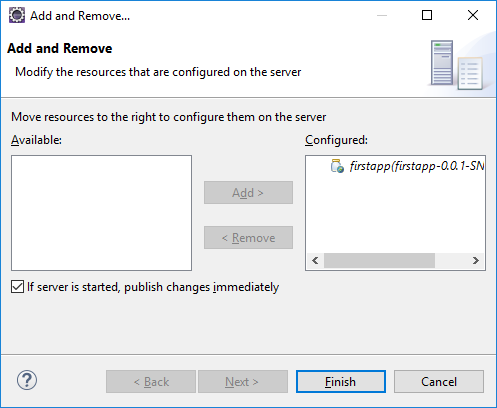
Bij eigen klassen is de annotatie altijd mogelijk, maar gebruik je servlets van derden, dan moet je soms een servlet-mapping aanmaken. Kies zelf welke aanpak je nu het beste acht!

**Stap 5**

Om API-documentatie van JEE klassen te kunnen zien (bijv. als je met je muis over een JEE-klasse gaat) stellen we de Javadoc-locatie in. Rechtermuisklik in map **Maven Dependencies** op de **javax.servlet-api-4.0.0.jar**. Kies **Properties**, **Javadoc Location**. Voer bij “Javadoc location path” deze URL in: <https://javaee.github.io/javaee-spec/javadocs/> , klik **Apply and Close**.

Les 2: practicum 4: Deployment op Tomcat

**Stap 1 (command-line alternatief *voor dit practicum*: klik** [hier](#_Deployment_op_Tomcat)**)**

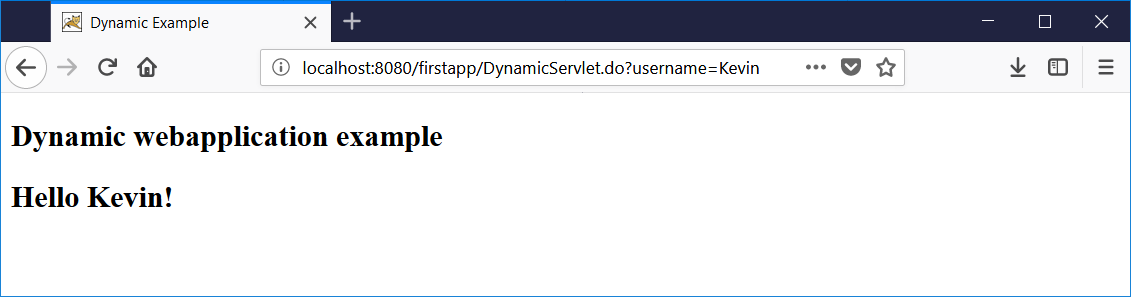
Je bent nu klaar om de webapplicatie aan Tomcat te koppelen. Open de Servers-view (zie evt. practicum 1, stap 4 hoe dat ook alweer moet), en rechtermuisklik op "Tomcat v9.0 Server…". Kies vervolgens **Add and Remove…**

Voeg (**afb. hiernaast**) het project toe aan de server. Klik **Finish**.

**Stap 2**

Rechtermuisklik nu op de Tomcat server en start deze. Je kunt in de Eclipse-console de Tomcat-output zien. Hier zijn ook eventuele exceptions in je webapplicatie terug te vinden. Open via <http://localhost:8080/firstapp> de index.html pagina in je browser (de browser zal automatisch proberen om index.html op te vragen als je geen html-pagina opgeeft).

Voer een naam in en klik <Go>. Je zou dan dit resultaat moeten zien:

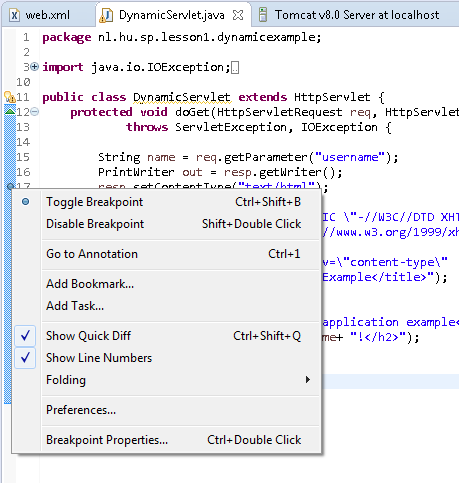


In de directory C:\apache-tomcat-9.0.6\wtpwebapps\firstapp zijn nu de bestanden terug te vinden die bij jouw applicatie horen. Dit kan soms handig zijn bij het opsporen van fouten.

Volg stap 5 als je de applicatie niet kunt openen.

**Stap 3 (optioneel -- Debuggen van Tomcat in Eclipse)**

Je kunt jouw code die in Tomcat draait ook debuggen in Eclipse. Daarvoor moet je voor de regel code waar je de controle wilt beginnen, een breakpoint plaatsen middels dubbelklikken of rechtermuisklik in de kantlijn:



Debuggen kan alleen als je de Tomcat server start in Debug-mode (rechtermuisklik op de Tomcat-server, **Debug**. Zodra je nu in de browser een actie uitvoert waardoor je langs de betreffende regel Java-code komt, zal Eclipse het programma pauzeren en vragen of je wil overschakelen naar Debug-perspective. Dan krijg je de beschikking over de volgende knoppen en views om door je programma heen te stappen:





Welke mogelijkheden er allemaal in de debugger zitten en hoe je deze dient te gebruiken vallen buiten de scope van dit vak. Door rechtsboven de Java-perspective te selecteren kom je weer terug in het perspectief dat je gewend bent om je code in te schrijven. Hier kan je het debuggen beëindigen door in de servers-view de Tomcat server te stoppen.

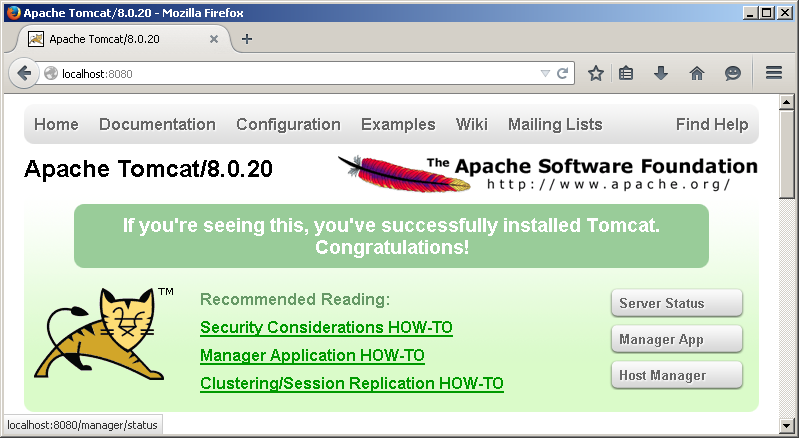
**Stap 4 (optioneel -- Tomcat starten buiten Eclipse)**

Als je Tomcat wilt starten buiten Eclipse, dan moet er op jouw systeem de globale variabele JAVA\_HOME zijn ingesteld. Als je dat nog nooit hebt gedaan, dan kan dat in Windows met: WINDOWS-KEY + PAUSE (shortcut naar System Properties), kies <Advanced system settings>, <Environment variables> en voeg de volgende System variable toe:

Variable name: JAVA\_HOME  
Variable value: C:\Program Files\Java\jdk-11.0.2

Uiteraard hangt de value af van de configuratie van jouw laptop of PC!! Je kunt Tomcat nu starten met startup.bat (dit bestand staat in de “bin” directory van Tomcat).

**Stap 5 (optioneel -- wijzigen van Tomcat HTTP-poort)**

Als je de applicatie niet kunt openen, verzeker je er dan van dat Tomcat echt draait door <http://localhost:8080/> in je browser te openen. Je ziet dan een pagina die lijkt op afbeelding hiernaast. Draait Tomcat **WEL**, kijk dan in de “wtpwebapps” directory van Tomcat of jouw “firstapp” directory daar staat en of de inhoud klopt.

Draait Tomcat **NIET**, dan komt dat vaak omdat de standaardpoort (8080) al in gebruik is door een ander programma. Open dan het configuratiescherm van practicum 1, stap 6. Wijzig onder “Ports” de HTTP/1.1 poort naar bijvoorbeeld 4711. Restart de Tomcat server.

Test nu jouw webapplicatie via: [http://localhost:4711/firstapp/](http://localhost:4711/firstapp/testpage.html)

## Les 2: Practicum 5: Rekenmachine

🡺 Breid het voorbeeld van practicum 1 uit zodat je twee getallen kunt invoeren waarmee gerekend moet worden. Maak daarvoor een extra formulier (form-tag) met twee invoervelden (input-tags) voor getallen. Maak daarnaast vier knoppen om te kunnen af- en optellen, delen en vermenigvuldigen. Ook hiervoor kun je een input-tag gebruiken.

Maak aan de serverkant een simpele Calculator-servlet die beide getallen uit een request kan halen en het resultaat kan berekenen. Dit resultaat is uiteraard afhankelijk van de knop die op de HTML-pagina werd ingedrukt. Het resultaat moet naar de browser geprint worden!

**Nog niet helemaal zeker van je HTML-skills?**

**w3schools HTML tutorial:** <https://www.w3schools.com/html/default.asp>

**w3schools input-tag:** <https://www.w3schools.com/tags/tag_input.asp>

**codecademy HTML tutorial:** <https://www.codecademy.com/courses/web-beginner-en-HZA3b/0/5>

Les 2: practicum 6: Git & GitHub

Bij de cursus Python Programming in blok 1 heb je al een GitHub account gemaakt. Zo niet, doe dat dan alsnog. Vraag daarna het [Student Developer Pack](https://education.github.com/pack/join) via GitHub aan zodat je ook de beschikking krijgt over **private repositories**. Gebruik hierbij je HU-mailaccount! Over het algemeen worden de verzoeken snel ingewilligd. Met een private repository kan je jouw code voor IPASS en WAC beschermen. **Voor de cursus WAC is een private repository verplicht**!

Log in bij GitHub en klik op de homepage **New repository**! Kies een repository-naam, maak de repository **private** en laat de overige instellingen ongewijzigd. Klik daarna **Create repository**. Je ziet vervolgens instructies om een project in GitHub te plaatsen. Dat gaan we nu doen.

Git is geïntegreerd in de meeste IDE’s, waardoor we deze stap dus zouden kunnen uitvoeren zonder de command-line. Vaak zie je echter online Git-commando’s die je moet uitvoeren om projecten te clonen of te committen etc. Daarnaast is de command-line tool ook erg krachtig vanwege de vele mogelijke opties. We maken daarom nu kennis met de command-line.

**Stap 1: Git installeren**

Test met een command-prompt / terminal of bij jou Git is geïnstalleerd met het onderstaande commando. Zo niet, [download](https://git-scm.com/download/) en installeer Git dan voor jouw besturingssysteem. Open de Git-bash (of OS-terminal) en test:

git --version

Als het goed is, zie je nu de door jou geïnstalleerde versie van Git geprint worden. Als je een foutmelding krijgt zit je ofwel in een verkeerde terminal, of Git is niet correct geïnstalleerd!

**Stap 2: Repository aanmaken**

Navigeer nu in de terminal naar de directory waar jouw project staat. Als je dat niet weet, kan je dat opzoeken in je IDE. Bekijk hiervoor de properties van je project.

* Navigeren met de command-line gaat op de meeste besturingssystemen op een vergelijkbare wijze. Je gebruikt hiervoor het commando **cd** (change directory), gevolgd door de directory waar je naar toe wilt. Voorbeeld op Windows: **cd c:\projecten\firstapp**.

Als je in de directory van je project bent aangekomen, voer dan de volgende commando’s uit:

**echo Projectinfo >> README.md**

**git init**

* Het eerste commando creëert een readme-bestand. Hierin kan je project-informatie zetten.
* Het tweede commando initieert een Git-repository in de huidige directory (.git directory).

**Stap 3: Repository configureren**

De repository is nu gemaakt. Voer nu ook het volgende commandos uit:

**echo /target/ >> .gitignore**

* Dit commando zorgt de target-directory NIET in Git komt. Deze directory wordt gegenereerd door Maven. Iedereen die het project kloont (kopieert), kan dat opnieuw door Maven laten doen. Te negeren directories & bestanden staan in het bestand **.gitignore**!

Gebruikers van Eclipse voeren ook de volgende commando’s uit om Eclipse-configuratie uit de repository te houden. Dit is namelijk geen onderdeel van het project zelf:

**echo /.settings/ >> .gitignore**

**echo .project >> .gitignore**

**echo .classpath >> .gitignore**

* Gebruikers van IntelliJ kunnen de **.idea** directory aan de **.gitignore** file toevoegen!
* De site <https://www.gitignore.io/> kan je helpen bij het maken van een goede .gitignore

**Stap 4: Initiële commit**

Nu moeten alle overige bestanden en directories aan Git toegevoegd worden, en dat moet bevestigd worden (commit). Dit doe je door de volgende twee commando’s uit te voeren:

**git add .**

**git commit -m "firstapp commit"**

Met het commando **git status** kan je eventueel de status van je repository controleren! Na de commit staat alles in de repository op jouw laptop, maar nog niet in GitHub. Dat gaan we in de volgende stap doen.

**Stap 5: GitHub push**

Tot slot **pushen** we nu ook de lokale repository naar GitHub met de volgende commando’s:

**git remote add origin https://github.com/<JOUW REPO>.git**

**git push -u origin master**

**Let op:** **<JOUW REPO>** moet je vervangen door je <GitHub-username>/<repositorynaam>. De exacte HTTPS-URL kan je vinden in GitHub, op het tabblad **Code** van je repository.

* Met het eerste commando geef je een **remote** repository op. Deze repository staat dus op een andere locatie dan de repository die je zojuist gemaakt hebt. De remote repository heeft hierboven de naam **origin** gekregen. Daar **push** je jouw lokale repository naar toe met het tweede commando!

Controleer in GitHub of jouw project correct is ge-commit!

## Les 2: Practicum 7: FileBrowser (**Optioneel**)

In bijlage 3 staat klasse [FileBrowserServlet](#_FileBrowserServlet.java). Deze geeft op een GET-request op de URL-mapping "/filebrowser", een response met het overzicht van subdirectories en bestanden in een directory. Subdirectories kan je bekijken door een parameter toe te voegen aan de URL: "/filebrowser?path=/Mijn muziek". Kopieer deze klasse naast de DynamicServlet.

🡺 Bestudeer de code en maak de volgende aanpassingen:

1. Stel de resourceBase in op de directory waar in je (bijvoorbeeld) je muziek opslaat.
2. Zorg dat bestanden ook als link worden getoond in de browser (zoals directories).
3. Schrijf een else-statement zodat een bestand als response geretourneerd zal worden als deze wordt opgevraagd. Bestudeer welke methoden van klasse [Files](https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html)[[1]](#footnote-1) je helpen bij regel 1, 3, 4 en 5 van de hieronder beschreven 5 benodigde regels code:
   1. Controleer of het gevraagde Path een bestand is.
   2. Vraag het reële[[2]](#footnote-2) Path op: requestedPath = requestedPath.toRealPath();
   3. Stel het response content-type in aan de hand van de bestands-extensie.
   4. Stel de response content-length in op de grootte van het bestand (in bytes).
   5. Kopieer het bestand naar de outputstream van het response-object.
4. Geef nu ook een melding als het gevraagde path geen bestand of de directory is. Zorg dat symbolic links naar bestanden buiten de resourceBase niet meer werken (het begin van het reële path is dan niet gelijk aan de resourceBase). Dit veroorzaakt anders beveiligingsrisico’s.

# Les 3, practicum: Webservices (Jersey & Maven)

## Les 3: practicum 1: Jersey instellen

Om services te programmeren, gebruiken we de Java API for RESTful Web Services (JAX-RS). In het Java EE platform is (met interfaces) wel beschreven hoe deze libraries werken, maar de implementatie (uitwerking in klassen) is niet meegeleverd. Sommige webservers, zoals JBoss, GlassFish of Wildfly leveren standaard al een (eigen) JAX-RS implementatie mee. Tomcat doet dat niet, maar je kunt wel zelf een JAX-RS implementatie aan Tomcat toevoegen zodat je toch webservices kunt maken!

Er zijn meerdere JAX-RS implementaties zoals Apache CFX, Jersey, RESTeasy etc. Allen hebben ze zo hun voor- en nadelen. Wij gebruiken de vrij eenvoudige Jersey-implementatie. Om webservices te kunnen compileren is ons project daarom afhankelijk (dependent) van de libraries van Jersey. Daarom moeten we Jersey aan ons project toevoegen.

**Stap 1: pom.xml**

We gaan Jersey niet zelf downloaden, maar we laten Maven dit doen. Zoals gezegd is Maven een tool om je project te beheren. Je hoeft daarvoor enkel de juiste dependency (afhankelijkheid) voor op te geven. Open daarom (in Eclipse) **‘pom.xml’**. Dit bestand kun je vinden in de root-directory van je project.

De juiste dependency kun je meestal vinden op de website van de aanbieder van libraries, zo ook bij [Jersey](https://jersey.github.io/documentation/latest/modules-and-dependencies.html#servlet-app-general). Jersey is op zijn beurt ook weer afhankelijk van andere (JEE 8) libraries. Voor de duidelijkheid zijn hieronder alle benodigde dependencies voor Jersey opgenomen:

<dependency>

<groupId>org.glassfish.jersey.containers</groupId>

<artifactId>jersey-container-servlet</artifactId>

<version>2.26</version>

</dependency>

<dependency>

<groupId>org.glassfish.jersey.inject</groupId>

<artifactId>jersey-hk2</artifactId>

<version>2.26</version>

</dependency>

<dependency>

<groupId>javax.xml.bind</groupId>

<artifactId>jaxb-api</artifactId>

<version>2.2.11</version>

</dependency>

<dependency>

<groupId>javax.activation</groupId>

<artifactId>activation</artifactId>

<version>1.1.1</version>

</dependency>

Het toevoegen van een dependency kan als volgt: open **pom.xml** (en klik in Eclipse onderin de editor op pom.xml). Zoek de <dependencies> tag. Hier staan al enkele andere dependencies. Voeg ook de zojuist gegeven dependencies toe en sla het bestand op. Eclipse zal Maven nu automatisch de juiste libraries laten binnenhalen. Je kunt dit zien als in je project de **‘Maven Dependencies’** openklapt (kan heel even duren).

* Als je al eens deze library hebt gedownload, dan moet je soms je project updaten omdat Maven foutmeldingen geeft dat de library niet gevonden kan worden (in jouw lokale repository). In Eclipse: rechtermuisklik op de projectnaam > **Maven** > **Update Project…** en vink vervolgens de checkbox **Force Update of Snapshots/Releases** aan en klik **OK**. Met de commandline kan het middels de opdracht: **mvn clean compile -U**.

**Stap 2: web.xml**

Je kunt nu jouw services met Jersey gaan ontwikkelen. Daarvoor moet je echter nog één onderdeel correct instellen. Open daarvoor het bestand **‘web.xml’** en voeg de volgende servlet en servlet-mapping toe:

<servlet>

<servlet-name>Jersey REST Service</servlet-name>

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>

<init-param>

<param-name>jersey.config.server.provider.packages</param-name>

<param-value>nl.hu.v1wac.firstapp.webservices</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>Jersey REST Service</servlet-name>

<url-pattern>/restservices/\*</url-pattern>

</servlet-mapping>

De opgegeven servlet is van Jersey zelf en is de enige die je hoeft te registreren. Dit maakt dat de package die als <init-param> is opgegeven, wordt gescand op webservice-klassen. Door de <load-on-startup> tag wordt de servlet geladen zodra Tomcat jouw applicatie opstart. Daardoor wordt methode init() (en dus de scan) uitgevoerd. Al jouw services komen dan beschikbaar onder de prefix **‘/restservices/’** zoals bij de <servlet-mapping> is ingesteld.

## Les 3: practicum 2: Webservice: WorldResource

Bij dit practicum maak je jouw eerste REST-webservice aan (in een nieuwe klasse **WorldResource**). Deze klasse heeft vier publieke URI’s, welke informatie over landen moeten opleveren in JSON-formaat. Om de JSON-API te kunnen gebruiken moet je de volgende dependency aan pom.xml toevoegen (zie voor het toevoegen van dependencies practicum 1):

<dependency>

<groupId>org.glassfish</groupId>

<artifactId>javax.json</artifactId>

<version>1.0.4</version>  
</dependency>

Het resultaat is afhankelijk van welke service iemand aangeroepen heeft. Het moet mogelijk zijn om de volgende informatie op te vragen:

1. Een lijst (array) van alle landen waar informatie van is, TYPE: GET, URI:

<http://localhost:8080/firstapp/restservices/countries>

1. Landinformatie (op basis van landcode), TYPE: GET, URI (bijv. voor China):

<http://localhost:8080/firstapp/restservices/countries/CN>

1. Een lijst (array) met de 10 grootste landen, gemeten naar oppervlakte, TYPE: GET, URI:

<http://localhost:8080/firstapp/restservices/countries/largestsurfaces>

1. Een array met de 10 grootste landen, gemeten naar inwoneraantal, TYPE: GET, URI:

<http://localhost:8080/firstapp/restservices/countries/largestpopulations>

De URL kan bij jou natuurlijk anders zijn (andere poort / applicatienaam)! Per land worden steeds de onderstaande gegevens opgeleverd. Een voorbeeld van het resultaat van URI 2:

{

"code":"CN", "iso3":"CHN",  
"name":"China", "continent":"Asia",

"capital":"Beijing", "region":"Eastern Asia",

"surface":9572900.0, "population":1277558000,

"government":"People'sRepublic",

"lat":39.55, "lng":116.23

}

Het is de bedoeling dat je leert werken met webservices. Je hoeft daarom niet alle informatie m.b.t. landen zelf op te zoeken. Gebruik daarvoor de klassen ServiceProvider, WorldService en Country die op [Canvas](https://canvas.hu.nl/courses/2767/files/170764/download?wrap=1) zijn gegeven. Benader de WorldService via de ServiceProvider. Aan jou de taak om de informatie om te zetten naar JSON.

🡺 Maak WorldResource (in package **‘nl.hu.v1wac.firstapp.webservices’**).

🡺 Controleer alle URI’s met HttpRequester, Postman of een andere browser-tool!

## Les 3: practicum 3: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository, en push deze naar GitHub.

# Les 4, practicum: JavaScript

## Les 4: Practicum 1: JavaScript ConsolE & Delayed Functions **(OPTIONEEL)**

Creëer in het firstapp-project van de vorige les (in de directory **webapp**) een nieuwe HTML-pagina met daarin een HTML-[textarea](https://www.w3schools.com/tags/tag_textarea.asp). Geef deze textarea een uniek id. Maak daarna ook een JavaScript bestand aan, en koppel deze aan de zojuist gemaakte HTML-pagina (met behulp van een script-tag).

Gebruik nu de **setInterval(func, delay)** functie van Javascript om elke 5 seconden de inhoud van de textarea in de JavaScript console van je browser te printen. Schrijf hiervoor zelf een functie die je aan de setInterval functie als parameter meegeeft. Deze functie print de inhoud van de textarea in de console, maar alleen als er iets in de textarea gewijzigd is.

* **Let op:** o.a. bij HTML-input en textarea elementen moet je niet de innerHTML-property uitlezen, maar de value-property. Hierin staat de tekst die de gebruiker heeft ingevoerd. Zie w3schools.com voor meer informatie over het [HTML DOM Textarea Object](https://www.w3schools.com/jsref/dom_obj_textarea.asp).

🡺 Realiseer de beschreven functionaliteit!

## Les 4: Practicum 2: Debugger **(OPTIONEEL)**

* Debugging is een belangrijk onderdeel van het software-ontwikkelen. Een debugger stelt je in staat om ‘draaiende’ code te onderbreken om te controleren of bepaalde stukjes code wel doen wat ze moeten doen. Het voordeel van een debugger t.o.v. een IDE is dat alle variabelen op dat moment een waarde hebben gekregen. Je kunt dus controleren of die ene variabele wel de waarde heeft die jij verwacht. Je kunt ook controleren waar de code een ander pad volgt dan je bij het ontwikkelen in je hoofd had.

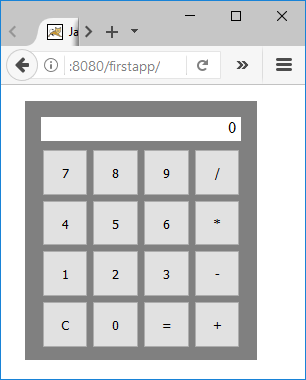
Je kunt code onderbreken door er, heel toepasselijk, een breakpoint te plaatsen. In Eclipse kan dat door bijvoorbeeld te dubbelklikken in de kantlijn voor de betreffende regel code. Bij het uitvoeren zal Eclipse (in Debug-mode) stoppen als hij bij die regel aankomt. Je kunt dan vervolgens variabelen inspecteren en/of stap voor stap de code laten uitvoeren (jij bepaalt dan wanneer een volgende regel uitgevoerd gaat worden.

Tegenwoordig hebben de meeste browsers ook debug-functionaliteit voor JavaScript (aangezien wij JavaScript als client-side taal gebruiken, kan je jouw JavaScript code ook niet in Eclipse debuggen). M.b.v. de developer tools van je browser (F12 in Chrome/Firefox) kan je de debugger openen en breakpoints plaatsen.

🡺 Plaats op de eerste regel van de functie van practicum 1 een breakpoint. Wat is het effect?

🡺 Probeer de code stap voor stap te laten uitvoeren en bekijk steeds de variabele-waarden!

## Les 4: Practicum 3: JavaScript Rekenmachine

In les 2 heb je een servlet gemaakt die voor elke berekening geraadpleegd werd. We kunnen nu met JavaScript we de browser hetzelfde laten doen, **zonder** dat de server hiervoor nodig is.

Je mag zelf met HTML en CSS de hiernaast getoonde rekenmachine maken (of iets wat erop lijkt), maar omdat dit practicum over JavaScript gaat, mag je ook de [gegeven code](#_Rekenmachine.html) gebruiken. Voorwaarde is dat je de inline-CSS wijzigt naar een [external stylesheet](https://www.w3schools.com/css/css_howto.asp)!

De rekenmachine moet in staat zijn om alle getoonde berekeningen (+,-,\*,/) uit te voeren.

Om je op weg te helpen: het is handig om in ieder geval globale variabelen op te nemen voor:

1. een eerder ingevoerd getal (of het resultaat van de laatste berekening)
2. het bijhouden van de ingevoerde operator (+,-,\*,/)
3. het getal in het display (dit groeit na iedere klik op een getal)

Schrijf daarna event-handlers voor alle knoppen. **Let op:** de knoppen met getallen schrijven allemaal een (ander) getal in het display. Kan je hiervoor wellicht één handler maken?

🡺 Creëer een JavaScript bestand, koppel deze aan de rekenmachine-HTML en schrijf de code!

🡺 Dit kan in Eclipse, maar je kunt hiervoor ook Visual Studio Code, Atom o.i.d. gebruiken!

* Eclipse ‘ziet’ wijzigingen die je in andere editors maakt standaard **niet**. Hierdoor moet je steeds het project in Eclipse refreshen om de wijzigingen naar Tomcat te laten deployen. Als je optie **Refresh using native hooks or polling** aanzet, ziet Eclipse **wel** wijzigingen van andere editors. Dat kan via menu Window > Preferences, categorie General > Workspace.

## Les 4: practicum 4: Git & GitHub

Commit je wijzigingen naar je lokale repository en push naar GitHub!

## Les 4: Practicum 5: Advanced Rekenmachine (**optioneel**)

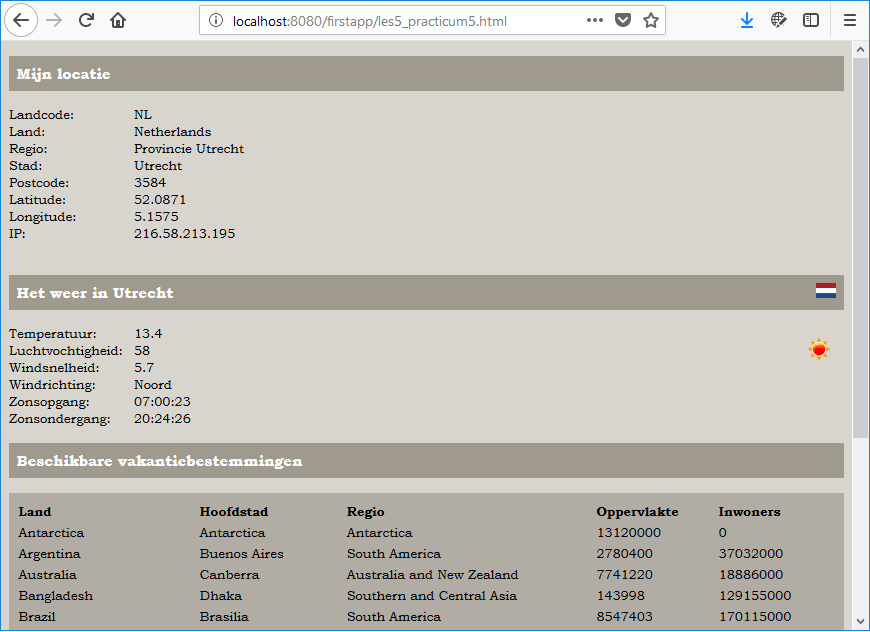
🡺 Voeg aan de rekenmachine knoppen toe voor;

1. CE (clear entry) om het display te resetten.
2. +/- inverter (positieve getallen worden negatief & vice versa).
3. X2 om het getal in het display te kwadrateren.
4. √(x) om de wortel van het getal in het display te berekenen.

# Les 5, practicum: OpenWeatherMap, IP-API & WorldResources

In dit practicum maak je onderstaande pagina, waarin je de WorldResource-webservice van les 3 combineert met de [IP-API](https://ipapi.co/json) en de [OpenWeatherMap-API](https://openweathermap.org/). De pagina toont:

* jouw locatiegegevens (verkregen via de IP-API – zie [docs](https://ipapi.co/api/)).
* het weer op de locatie waar jij bent (via de OpenWeatherMap-API – zie [docs](https://openweathermap.org/current)).
* een deel van de informatie van de landen die jouw eigen webservice teruggeeft:



Als je de pagina opent, moet het weer van jouw locatie getoond worden. Zodra je op een land klikt, moet het weer in de betreffende hoofdstad getoond worden.

Om je op weg te helpen is de opdracht opgedeeld in verschillende practicumopdrachten:

* Practicum 1: Aanroepen van IP-API
* Practicum 2: Aanroepen van de OpenWeatherMap-API
* Practicum 3: Aanroepen van je eigen WorldResource-webservice
* Practicum 4: Afbeeldingen plaatsen (optioneel)
* Practicum 5: Afrondende, losse opdrachten
* Practicum 6: Optionele (uitdagende) opdracht
* **In de volgende les pas je de layout van de pagina verder aan. Het is handig om de pagina te maken zoals de opdrachten beschrijven om een vergelijkbaar uiterlijk te krijgen!**

## Les 5: practicum 1: IP-API

Creëer een HTML-pagina met daarin in ieder geval de volgende body:

<body>

<div id=*"frame"*>

<div id=*"myLocation"*></div>

<div id=*"weatherInfo"*></div>

<div id=*"countryList"*></div>

</div>

<!-- Hier komen jouw script-tags -->

</body>

Maak binnen div met id *"myLocation"* HTML-elementen ([labels](https://www.w3schools.com/tags/tag_label.asp)) waarin je de locatiegegevens gaat tonen. Geef ze ook een id, zodat je de inhoud kunt wijzigen met JavaScript. Kijk in de [documentatie](https://ipapi.co/api/) van de IP-API wat de response is van de <https://ipapi.co/json/> URI. Schrijf een (JS-) functie initPage() die de URI aanroept. Plaats de response-data in de HTML-elementen. Zorg dat functie initPage() wordt uitgevoerd als de browser het script laadt.

Koppel een stijl aan je pagina middels een externe CSS-stylesheet. Zorg voor een:

* [background-color](https://www.w3schools.com/cssref/pr_background-color.asp) van de gehele pagina!
* uitgelijnd overzicht met locatie-gegevens (bijv: [float](https://www.w3schools.com/cssref/pr_class_float.asp) de labels en geef een [width](https://www.w3schools.com/cssref/pr_dim_width.asp) op).

🡺 Realiseer het locatie-overzicht! **Let op:** virusscanners/AdBlockers kunnen de API blokkeren!

## Les 5: practicum 2: OpenWeatherMap-API

* Creëer nu binnen de div met id ”*weatherInfo*” ook labels, maar dan voor het tonen van de weersinformatie van een bepaalde locatie (plaatjes mag je even overslaan).
* Voor de OpenWeatherMap-API heb je een key nodig die je bij elk request moet meesturen. Maak hiervoor een account aan en probeer een aantal [voorbeelden](https://openweathermap.org/current) uit om het huidige weer op te vragen. Bekijk welke informatie de API retourneert. Dit kan in HttpRequester of Postman, waar je de response leesbaar kunt tonen. Zoek uit welke API-URI je nodig hebt om weersinformatie op te vragen op basis van geografische coördinaten. **Let op:** gebruik http**s**:// om met OpenWeatherMap te communiceren. Dit is nodig voor latere opdrachten.
* Schrijf de (JS-) functie showWeather(latitude, longitude, city) die het weer opvraagt voor de gegeven parameters (breedtegraad en lengtegraad). Toon de weerinfo zoals in het voorbeeld (en zet de plaatsnaam, city, in een h3-header)! Roep deze functie aan vanuit initPage(), en geef daarbij als parameter de latitude en longitude van jouw locatie op!

**Let op:** standaard zijn de response-waarden in andere eenheden gegeven dan wij gewend zijn (o.a. Kelvin i.p.v. Celcius). De API biedt mogelijkheden om andere eenheden te laten retourneren, zodat je de uitvoer kunt realiseren zoals in de screenshot gegeven is. De windrichting en tijdstippen zal je wel zelf moeten omzetten van getallen naar strings.

## Les 5: practicum 3: WorldResource-API (webservice)

Plaats in de div met id ”*countryList*” een tabel (<table>). Zorg dat deze tabel alvast een rij heeft (<tr>), met headers (<th>) voor land, hoofdstad, regio, oppervlakte en inwoners. De overige rijen maak je aan vanuit JavaScript.

Maak daarna de (JS-) functie loadCountries() die je eigen rest-service aanroept. Je krijgt dan een lijst met landen terug. Schrijf per land een **nieuwe** tabel-rij en plaats de benodigde gegevens elk in een table-data (<td>) tag. Gebruik hierbij de [for-of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) van JavaScript. **Let op:** gebruik **let** of **const** in de for-of (zie documentatie).

De webservice levert per land ook de geografische coördinaten van de hoofdstad mee. Koppel aan elke tabelrij een event-handler, zodat als je op een rij klikt, de getoonde weersinformatie vervangen wordt door het weer van de hoofdstad van dat land. Roep daarvoor de functie showWeather(..) aan, waarbij je de coördinaten en hoofdstad als parameters meegeeft!

🡺 Maak de div met tabel voor het landenoverzicht.

🡺 Schrijf de functie loadCountries() en roep deze aan vanuit initPage().

## Les 5: practicum 4: Images (**OPTIONEEL**)

Het tonen van een vlag is eenvoudig. Koppel de volgende CSS-stylesheet aan je pagina:

<https://cdnjs.cloudflare.com/ajax/libs/flag-icon-css/2.8.0/css/flag-icon.min.css>

(...of download deze file in je eigen project). Plaats nu met een [span](https://www.w3schools.com/tags/tag_span.asp) een vlag op je pagina:

<span id=*"flag"* class=*"flag-icon flag-icon-gr"*></span>

Dit element toont de Griekse vlag (*-gr*), maar door de CSS-class te veranderen in *flag-icon flag-icon-nl* krijg je de Nederlandse vlag te zien. Zie ook de [documentatie](https://github.com/lipis/flag-icon-css)! Breid functie showWeather(..) uit met parameter countryCode, en zorg dat bij het tonen van de weersinfo ook de span-class wijzigt. Gebruik hiervoor de JS-functie [setAttribute(name, value)](https://developer.mozilla.org/en-US/docs/Web/API/Element/setAttribute).

🡺 Toon steeds de vlag van het land waarvan weersinformatie in het scherm te zien is.

In het response-bericht van OpenWeatherMap staat ook een icon-property:

... { ..., "icon": "10n",

OpenWeatherMap biedt [icoontjes](https://openweathermap.org/weather-conditions) aan op de URL https://openweathermap.org/img/w/. Door de icon (en png-extensie) erachter te plakken ([vb](http://openweathermap.org/img/w/10n.png)), kan je dit icoontje tonen:

var imgTag = "<img src='https://openweathermap.org/img/w/"+icon+".png'/>";

document.querySelector("#weatherConditionSpan").innerHTML = imgTag;

🡺 Plaats het icoontje op een handige plek bij de weersinformatie.

## Les 5: practicum 5: Afrondende opdrachten

In deze opdracht nog enkele (niet perse samenhangende) opdrachten:

* Je hebt de afgelopen lessen verschillende HTML-pagina’s geschreven. Het is lastig om telkens de URL hiervan in de browser in te tikken. Als je een index.html hebt, hernoem deze dan en maak een nieuwe index.html. Maak op de nieuwe index.html met anchor (<a>) tags (hyperlinks) een overzicht waarmee je verwijst naar alle eerdere uitwerkingen.
* Koppel ook een event-handler aan je eigen locatie-plaatsnaam zodat als je erop klikt, je het weer van je eigen locatie (weer) geladen wordt.
* Zorg dat wanneer je met de muis over de tabel-rijen of jouw locatie-plaats heen beweegt, deze rijen (of de plaatsnaam) wit wordt. Dit kan je doen met de CSS-pseudoclass [:HOVER](https://www.w3schools.com/cssref/sel_hover.asp). Zorg hierbij voor een duidelijke muisaanwijzer ([cursor](https://www.w3schools.com/cssref/pr_class_cursor.asp)).

## Les 5: practicum 6: Sorteren en Filteren (**OPTIONEEL**)

* Zorg dat de tabel op alle kolommen te sorteren is op alfabetische volgorde. Bedenk zelf wat hiervoor de beste oplossing is. Je kunt de webservice hiervoor uitbreiden, maar het kan het ook geheel in JavaScript.

## Les 5: practicum 7: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository en push naar GitHub!

# Les 6, practicum: HTML5 Web Storage & CSS Grid Layout

## Les 6: practicum 1: Synchronisatie **(OPTIONEEL)**

* Schrijf een eenvoudige HTML-pagina met daarin een input-textfield. Zorg dat bij elke wijziging (keyup-event) de waarde van het invoerveld in het localStorage object terechtkomt.
* Schrijf een tweede pagina met alleen een label. Zorg dat de inhoud van het label bijgewerkt wordt als de inhoud van het invoerveld op de andere pagina wijzigt. Pas hiervoor een event-handler toe die een StorageEvent verwerkt!
* Open beide pagina’s en controleer of het correct werkt!

## Les 6: practicum 2: HTML5 Storage

Als je in de uitwerking van les 5 op een land klikt, zal jouw code nu telkens opnieuw het weer van dat betreffende land bij OpenWeatherMap opvragen. Dit is volgens de documentatie van de API echter niet zinvol, omdat het weer niet zo vaak wijzigt. Liever hebben ze dat je hooguit [om de 10 minuten](https://openweathermap.org/appid#use) een update opvraagt, natuurlijk ook om de server te ontlasten.

Zorg dat je elke succesvolle OpenWeatherMap-response niet alleen toont, maar ook opslaat in de browser-storage. Bepaal zelf of dat de Local of Session Storage moet zijn. Uiteraard moet je ook het tijdstip onthouden waarop je het betreffende response-bericht hebt ontvangen.

* Zorg in je applicatie voor maximaal 1 OpenWeatherMap-request per land, per 10 minuten!

## Les 6: practicum 3: Responsive Design

Controleer met je smartphone (deel Wifi met je laptop), door de Responsive-Design modus van je browser of gewoon door de browser te re-sizen hoe responsive je uitwerking van les 5 is. Waarschijnlijk zal de pagina erg klein getoond worden, omdat de viewport niet is ingesteld.

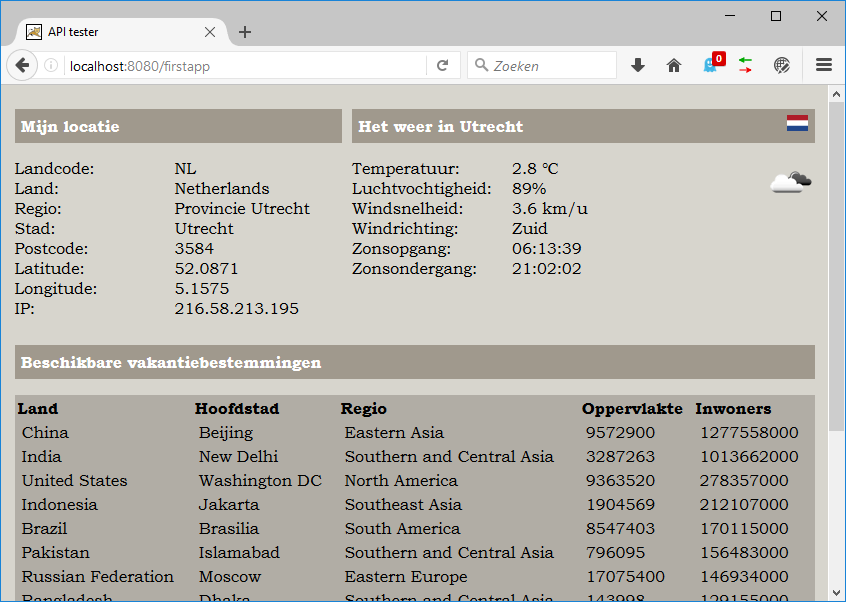
* Stel de viewport in.

De pagina zal nu al redelijk op een smal scherm weergegeven worden, maar het kan zijn dat bijvoorbeeld labels te breed zijn. En afhankelijk van de schermgrootte zal de tabel deels buiten het scherm vallen. Zie [hier](https://www.w3schools.com/howto/howto_css_table_responsive.asp) hoe je de tabel met landen responsive kunt maken!

* Maak eventueel aanpassingen aan de pagina zodat deze goed op een klein scherm (ga uit van zo’n 400 pixels breed) getoond wordt. Zorg ook dat de tabel met landen naar links en/of rechts gescrold kan worden als deze te breed is voor het scherm.
* Bij een smalle tabel wordt soms na [white-space](https://www.w3schools.com/cssref/pr_text_white-space.asp)s een enter geplaatst. Dat is te voorkomen!

## Les 6: practicum 4: CSS Grid Layout

De applicatie wordt nu goed op een klein scherm getoond. Maar op een breed scherm kunnen de locatiegegevens en de weersinformatie prima ***naast*** elkaar getoond worden:



Dit kan goed gedaan worden met een CSS Grid Layout. Hiervoor hoef je de HTML-pagina in principe niet aan te passen! Zorg met een CSS Media Query dat alleen in schermen die breder zijn dan 750 pixels, de beide div’s naast elkaar getoond in een Grid Layout getoond worden.

* Omdat de smalle schermen al correct getoond worden, hoef je daar dus geen aanpassing voor te maken, maar beschrijf je alleen maar aanvullende regels voor bredere schermen.
* Realiseer de beschreven Grid Layout voor schermen groter dan 700 pixels.

## Les 6: practicum 5: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository en push naar GitHub!

# Les 7, practicum: PostgreSQL Database

## Les 7: practicum 1: Installatie & configuratie

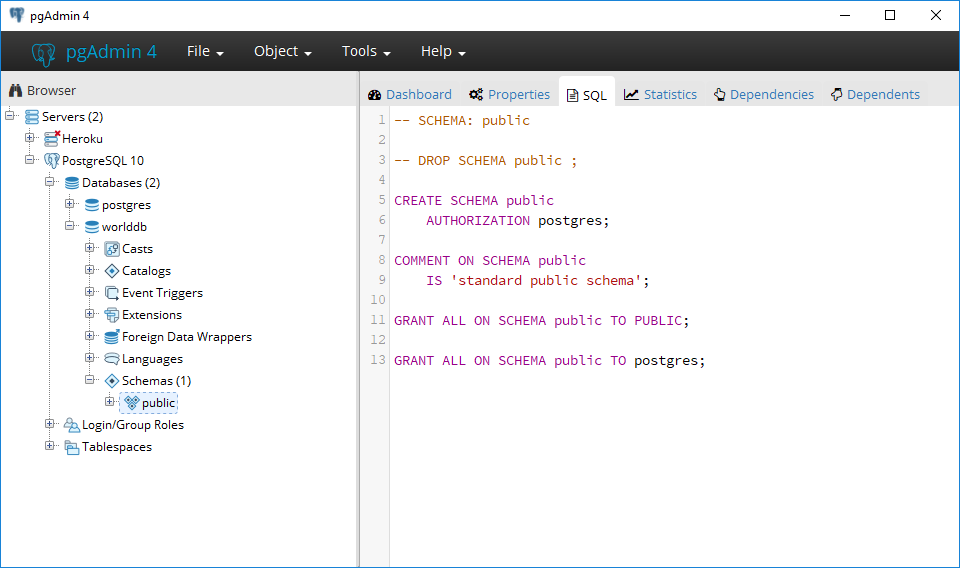
De WorldService die je in les 3 en 5 hebt gebruikt, heeft nu nog gewoon de op te leveren data hardcoded in de Java-code staan. Het is natuurlijk realistischer dat deze informatie uit een database komt die ook door anderen gebruikt en geüpdatet kan worden. Deze les koppel je een **database** aan de eerder gebruikte WorldService.

Omdat we in les 10 ook een applicatie gaan hosten op Heroku, gebruiken we tijdens deze les een database die ook bij Heroku beschikbaar is. We gebruiken daarom **geen** Oracle database, maar de **PostgreSQL** database die je bij de cursus TICT-V1MOD-15 hebt geïnstalleerd. Voor het database-beheer gebruiken we **pgAdmin IV**, die je ook al hebt leren kennen.

Tijdens het ontwikkelen van een applicatie gebruik je meestal een andere database dan die klanten gebruiken op de live-omgeving. Anders is er het risico dat klantgegevens verloren gaan of dat de applicatie crasht. Het is niet fijn als eindgebruikers daar last van ondervinden. Deze les maken we dus alleen voor de ontwikkelomgeving een database. Die kan gewoon op jouw laptop geïnstalleerd worden. Daarom wordt deze ook wel **lokale** database genoemd!

**Stap 1: installatie**

Als je PostgreSQL 10 en **pgAdmin IV** niet meer hebt, [download](https://www.postgresql.org/download/) en installeer deze dan (gebruik eventueel de [handleiding](https://cursussen.sharepoint.hu.nl/fnt/51/TICT-V1MOD-15/Studiemateriaal/05%20Introductie%20databases%20en%20SQL%20DML/SQL%20Installatie%20handleiding%20postgreSQL%20versie%201_0.pdf) van TICT-V1MOD-15). Open daarna pgAdmin en creëer op de database-server op je laptop een nieuwe database. Geef deze de naam ‘**worlddb’** en klik Save. PostgreSQL creëert nu een public schema (set van tabellen en objecten):



**Stap 2: database vullen met pgAdmin**

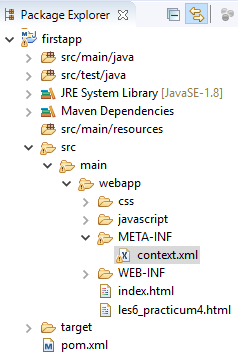
De database moet nu gevuld worden met een aantal tabellen en data. Dit kun je zelf natuurlijk doen, maar op [Sharepoint](https://cursussen.sharepoint.hu.nl/fnt/53/TICT-V1WAC-15/Studiemateriaal/Java-sources/V1WAC-15_les07_world.sql) kun je een script vinden om dat voor dit practicum te doen. Open dat script met een teksteditor en plak de inhoud van het script in de SQL-editor van pgAdmin (rechtermuisklik op het nieuwe **public** schema, kies voor Query Tool).

Voer het script uit, en refresh het schema. Controleer welke tabel er is toegevoegd!

**Stap 3: Tomcat voorzien van PostgreSQL driver**

[Download](https://jdbc.postgresql.org/download.html) nu ook de (meest recente) Java connector. Plaats het bestand in directory **‘lib’** in de **Tomcat**-installatiemap. Hiermee kan Tomcat met PostgreSQL-databases communiceren.

**Stap 4: Tomcat configureren**

Om te zorgen dat de code op Tomcat **èn** straks op Heroku werkt, gaan we de URL naar de database **niet** hardcoded in Java-code opnemen, maar in de configuratie van het project! Daarvoor moet je het bestand **context.xml** aanmaken.

Creëer in de map **webapp** (zie screenshot hiernaast), Kies **New**, **Folder**. Geef deze de exact de naam **META-INF** (hoofdlettergevoelig).

Rechtermuisklik op de nieuwe map en kies **New**, **File** en geef dit bestand de naam **context.xml**. Voeg daar de onderstaande XML-code toe. **Vervang** **password** en eventueel **username.** Als je de database een andere naam dan **‘worlddb’** hebt gegeven moet je dat ook wijzigen:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<Context>

<Resource name=*"jdbc/PostgresDS"*

url=*"jdbc:postgresql://localhost:5432/****worlddb****"*

driverClassName=*"org.postgresql.Driver"*

auth=*"Container"*

type=*"javax.sql.DataSource"*

username=*"****postgres****"*

password=*"****secret****"* />

</Context>

Tomcat maakt hierdoor bij het starten van de server een **databaseverbinding** **(resource)** met als naam **‘jdbc/PostgresDS’** beschikbaar. De resource kun je in de code opvragen!

## Les 7: practicum 2: DAO implementatie

We gebruiken een **DAO** (Data Access Object) om de database aan te spreken. We hebben de interface **CountryDao**, en de klasse die de implementatie hiervan voor PostgreSQL bevat, is **CountryPostgresDaoImpl**. Deze DAO haalt informatie van landen op uit de database, en heeft klasse **PostgresBaseDao** als superklasse. De superklasse dient om een Connection-object naar de PostgreSQL database op te vragen waarmee je een SQL-statement kunt uitvoeren. Een voorbeeld van deze constructie is gegeven in de powerpoint-slides van deze les!



Klasse **Country** heb je tijdens les 3 al gedownload. In het UML zijn de getters, en constructor weggelaten. Interface **CountryDao** kent zeven methoden om via queries verschillende gegevens uit de database te halen, te verwijderen of te wijzigen. In deze methoden vraag je een connectie op (met super.getConnection()) en voer je een query uit.

* Je zult merken dat de ‘find’-methoden een vergelijkbaar patroon volgen. Je voert een statement uit, doorloopt een while-lus en plaatst het record in een object. Je kunt dit slimmer aanpakken door delen van het ‘dubbele’ werk in **private methoden** uit te voeren!

🡺 Maak de interface en de 2 nieuwe klassen (in package **‘nl.hu.v1wac.firstapp.persistence’**)!

## Les 7: practicum 3: WorldService aanpassen

De klasse WorldService moet nu gewijzigd worden zodat deze de informatie ook echt uit de database haalt. Verwijder daarvoor de **constructor** en attribuut **‘allCountries’**. Pas de methoden zo aan dat de DAO wordt aangesproken! De methoden save, update en delete hoef je nog niet te verwerken in de WorldService klasse (wel in de DAO), dat komt de volgende les!

* De klasse WorldService wordt nu min of meer een doorgeefluik omdat de DAO de benodigde informatie uit de database kan leveren. De service hoeft niet veel meer te doen. Het is echter niet altijd wenselijk en mogelijk om alle informatie zo uit de database te halen. Zodra er meer ingewikkelde bewerkingen uitgevoerd moeten worden waarbij bijvoorbeeld meerdere DAO’s betrokken zijn, ‘groeit’ de WorldService weer!

🡺 Voer de beschreven aanpassingen uit en test of jouw webservices nog steeds werken!

## Les 7: practicum 4: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository en push naar GitHub!

# Les 8, practicum: RESTful Webservices & Relational Databases

Tijdens les 3 heb je een voor een viertal URI’s webservices gebouwd, waaronder twee URI’s voor het opvragen van land(en)-informatie:

1. <http://localhost:8080/firstapp/restservices/countries>
2. [http://localhost:8080/firstapp/restservices/countries/<landcode>](http://localhost:8080/firstapp/restservices/countries/%3clandcode%3e)

In deze opdracht maak je het mogelijk om op diezelfde URI’s niet alleen land(en)-informatie op te vragen, maar ook toe te voegen, wijzigen en verwijderen.

## Les 8: practicum 1: PUT (wijzigen) en DELETE (verwijderen) Requests

* **Let op:** de Response-voorbeelden in de slides vereisen de **Jackson-dependency** in pom.xml!

**De GUI (clientside):**

* Voeg in de eerder gemaakte tabel met landen **per land** een verwijder- en wijzigknop toe.
* Verzend bij een klik op de verwijderknop een DELETE-request; update de tabel!
* Bepaal hoe je het wijzigen mogelijk wilt maken. **Suggesties** bij een klik op de wijzigknop:
  + Toon landinfo in een formulier boven de tabel (geef het formulier [focus](https://www.w3schools.com/jsref/met_html_focus.asp)).
  + Toon landinfo in een formulier dat ‘uitklapt’ onder een tabelrij.
  + Maak de tabelrij editable (plaats de gegevens van de aangeklikte rij in input-tags).
  + Toon een div-element met invoerformulier als popup/overlay. Dat kan o.a. met CSS-properties [position](https://www.w3schools.com/cssref/pr_class_position.asp) (fixed) en [display](https://www.w3schools.com/cssref/pr_class_display.asp) die je [wijzigt met JavaScript](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style).
* Alle zichtbare tabelgegevens moeten te wijzigen zijn. Verzend daarna een PUT-request.
* Toon eventuele foutmeldingen van de webservice aan de gebruiker.

**De webservice (serverside):**

* Beoordeel zelf of het nodig is om een aparte Resource-klasse (webservice) te maken voor URI 2, of dat je alles in WorldResource gaat programmeren.
* Maak op URI 2 DELETE- en PUT-requests mogelijk om landen te wijzigen/verwijderen.
* Maak gebruik van de UPDATE en DELETE methoden van je CountryDAO van les 7!
* Return een foutmelding (in JSON) als het wijzigen niet lukt.

## Les 8: practicum 2: POST (toevoegen) requests

* Maak voor het toevoegen van een nieuw land een formulier. Bepaal zelf waar en hoe je dat doet. Vergeet de velden voor verplichte database-waarden niet (o.a. een landcode).
* Maak op URI 1 POST-requests mogelijk, om een nieuw land toe te voegen.
* Return een foutmelding (in JSON) als het toevoegen niet lukt.

## Les 8: practicum 3: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository en push naar GitHub!

# Les 9, practicum: Authentication, Authorization & HTTPS

In deze les zorg je dat er ingelogd moet/kan worden voor het gebruiken van de webservices van les 8. In practicum 1 schakel je HTTPS in voor Tomcat. In practicum 2 werk je stapsgewijs naar een beveiligde REST-API toe, waarvoor je in practicum 3 een login-formulier maakt.

Practicum 4 en 5 zijn optioneel, maar wel handig om te maken, omdat je hier oefent met een registratieformulier. Mogelijk heb je zoiets ook nodig bij de IPASS opdracht.

## Les 9: practicum 1: Tomcat & HTTPS

**Let op: lukt deze configuratiestap niet, ga dan gewoon verder met de volgende opdrachten!**

Steeds meer webapplicaties communiceren alleen via https. Op Heroku is zowel http als https mogelijk. Lokaal kan je ook https mogelijk maken met een zelfgemaakt certificaat. Voer daarvoor het onderstaande commando uit op jouw besturingssysteem. Als op jouw systeem JAVA\_HOME niet is ingesteld, werkt dit commando niet. Dan kan je alles voor /bin (of \bin) vervangen door de locatie van de Java-installatie op jouw systeem.

Windows: **"%JAVA\_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA**

Unix: **$JAVA\_HOME/bin/keytool -genkey -alias tomcat -keyalg RSA**

Voer als password **changeit** in en beantwoord de persoonlijke vragen. Geef tot slot opnieuw het password **changeit** op. Dit commando zorgt voor een .keystore bestand in jouw persoonlijke directory. Hierin is het zojuist gegenereerde, zelfgemaakt certificaat opgeslagen.

Stop Tomcat als deze nog draait. Gebruik je Eclipse, open dan in de Package Explorer het bestand **servers.xml** van het Servers-project.

* Gebruik je Eclipse niet, open **servers.xml** dan in de ‘conf’ map van de Tomcat-installatie.

Voeg binnen de Service-tag de volgende regels toe:

<!-- Define a SSL Coyote HTTP/1.1 Connector on port 8443 -->

<Connector

protocol="org.apache.coyote.http11.Http11NioProtocol"

port="8443" maxThreads="200"

scheme="https" secure="true" SSLEnabled="true"

keystoreFile="${user.home}/.keystore" keystorePass="changeit"

clientAuth="false" sslProtocol="TLS"/>

Na starten van Tomcat kan je jouw applicatie openen via <https://localhost:8443/firstapp>. Tomcat gebruikt standaard poort 8443 voor HTTPS. Meer info kan je [hier](https://tomcat.apache.org/tomcat-8.0-doc/ssl-howto.html) vinden. Maar zoals poort 80 de standaardpoort voor HTTP is, zo is 443 dat voor HTTPS. Verander je de poort hierboven in 443, dan is de applicatie bereikbaar op <https://localhost/firstapp>.

**Let op:** de browser waarschuwt wel dat dit geen officieel certificaat is! Je moet dus expliciet aangeven dat de browser toch naar de betreffende pagina moet gaan.

## Les 9: practicum 2: Authentication & Authorization

**Stap 1: tabel useraccount**

Om te kunnen inloggen, hebben we in de database een tabel met gebruikers nodig. Creëer in dezelfde database waarin de landentabellen staan, ook een tabel **useraccount** met 3 kolommen: **username**, **password** en **role**. Allen van het type character varying(20). Gebruik de kolom username als **primary key**, en creëer vast enkele gebruikers met de rol **‘user’**.

* **Let op;** we gaan nu weliswaar de verbinding tussen client en server met https beveiligen, maar het opslaan van wachtwoorden als plain text in de database vormt ook een risico. Als iemand erin slaagt het wachtwoord van de database te achterhalen, dan zijn alle inlog-gegevens in te zien. Dit kan voorkomen worden door een hash van een wachtwoord op te slaan, in plaats van de oorspronkelijke tekst. Dit valt enigszins buiten de scope van deze cursus, maar [dit artikel](https://crackstation.net/hashing-security.htm) is een goed startpunt om hier meer over te leren. Er zijn ook voldoende [aanknopingspunten](https://www.owasp.org/index.php/Hashing_Java) te vinden hoe je dit in Java zou kunnen toepassen.

**Stap 2: de UserPostgresDaoImpl**

Maak een DAO voor de tabel useraccount. Neem onderstaande model als uitgangspunt. De methode van de DAO moet voor de parameters username en password de bijbehorende rol van de gebruiker ophalen uit de database. Bij ongeldige gegevens is de return-waarde null.



Schrijf UserDao en UserPostgresDaoImpl (in package nl.hu.v1wac.firstapp.persistence).

**Stap 3: JSON webtoken-dependency**

Voeg de volgende dependency aan je pom.xml toe. Met deze library gaan we in stap 4 unieke JSON webtokens genereren:

<dependency>

<groupId>io.jsonwebtoken</groupId>

<artifactId>jjwt</artifactId>

<version>0.9.0</version>

</dependency>

**Stap 4: de AuthenticationResource**

Om een beveiligde service te kunnen gebruiken, moet de browser eerst een JWT hebben. Schrijf een webservice (of neem ‘m over van de slides van les 9) die de inlog-gegevens van een gebruiker controleert en een JWT genereert. Plaats de klasse in een package die Jersey scant, bijvoorbeeld nl.hu.v1wac.firstapp.webservices.

* **Let op;** waarschijnlijk heb je in les 8 al de Jackson-dependency (zie slides les 8) toegevoegd. Zo niet, doe dat dan alsnog als je de AuthenticationResource van de slides van les 9 gebruikt. Doe je dat niet, dan krijg je in de Tomcat-console de volgende foutmelding:

SEVERE: MessageBodyWriter not found for media type=application/json, type=class <class>.

**Stap 5: de SecurityContext**

Als iemand met een JWT een beveiligde webservice wil gebruiken, zal de server kijken of de JWT geldig is, en welke user de service wil gebruiken. De gegevens van de user worden daarna in een **SecurityContext** opgeslagen, en Jersey gebruikt deze informatie bij de autorisatie.

Door een SecurityContext te implementeren kan een applicatie zelf bepalen welke gegevens worden bijgehouden. Voeg daarom de volgende klasse toe (in dezelfde package als de AuthenticationResource van stap 4):

**import** java.security.Principal;

**import** javax.ws.rs.core.SecurityContext;

**public** **class** MySecurityContext **implements** SecurityContext {

**private** String name;

**private** String role;

**private** **boolean** isSecure;

**public** MySecurityContext(String name, String role, **boolean** isSecure) {

**this**.name = name;

**this**.role = role;

}

**public** Principal getUserPrincipal() {

**return** **new** Principal() {

**public** String getName() { **return** name; }

};

}

**public** **boolean** isUserInRole(String role) { **return** role.equals(**this**.role); }

**public** **boolean** isSecure() { **return** isSecure; }

**public** String getAuthenticationScheme() { **return** "Bearer"; }

}

Deze klasse implementeert de interface SecurityContext. Dit is een JEE-interface, van Java zelf dus. Met de constructor kunnen de drie gegevens (username, user-role en of de verbinding veilig is) ingesteld worden. De overige methoden zijn methoden van de interface, die de betreffende attributen weer returnen.

**Stap 6: het AuthenticationFilter**

Nu er een SecurityContext is, kan ook het filter gemaakt worden dat de JWT controleert en de SecurityContext vult met informatie. Schrijf dit filter zelf, of neem ‘m over van de slides van les 9. Plaats de klasse in **dezelfde package** als de AuthenticationResource van stap 4.

**Stap 7: Dynamic Roles**

Om nu de webservices met annotaties te kunnen koppelen aan een rol, moet dit nog in Jersey ingesteld worden met een init-param van de Jersey-servlet. Zoek in je **web.xml** de servlet met naam Jersey REST Service op, en voeg de hieronder vetgedrukte init-param toe:

<servlet>

<servlet-name>Jersey REST Service</servlet-name>

......

**<init-param>**

**<param-name>jersey.config.server.provider.classnames</param-name>**

**<param-value>org.glassfish.jersey.server.filter.RolesAllowedDynamicFeature</param-value>**

**</init-param>**

</servlet>

Doe je dit niet, dan zal Jersey de @RolesAllow annotatie van de volgende stap negeren!

**Stap 8: REST-API autorisatie**

Maak de webservices die je voor les 8 hebt gemaakt om landen te wijzigen, te verwijderen en toe te voegen, alleen beschikbaar voor gebruikers met de rol **"user"**:

@RolesAllowed("user")

Wanneer je meerdere rollen toegang wilt geven kan je dat als volgt doen:

@RolesAllowed({"user", "admin"})

* Webservices **zonder** annotatie zijn voor iedereen toegankelijk!

**Stap 9: code inspectie**

Je hebt nu mogelijk veel code in je project geplakt die je niet (helemaal) zelf hebt geschreven. Probeer nogmaals de stappen te doorlopen en probeer jezelf of iemand anders uit te leggen waarvoor de betreffende stappen nodig zijn, en wat de code doet die je hebt overgenomen.

## Les 9: practicum 3: Login

* Maak een login-formulier (op de pagina die je voor les 8 hebt gemaakt, of op een andere pagina, dat maakt niet zoveel uit), en zorg dat hier een username en password ingevuld kunnen worden. Zorg ook voor een knop om het formulier te kunnen verzenden.
* Roep bij indrukken van de knop de AuthenticationResource (practicum 2, stap 4) aan met de ingevulde gegevens. Sla het token op in de JavaScript Session-Storage onder de naam ‘sessionToken’. Test dit voor een gebruiker die je in de database hebt geplaatst.
* Geef bij aanroepen van de beveiligde webservices het token uit de Session-Storage mee als request-header. Zie de slides van les 9 voor een voorbeeld! Geef een nette foutmelding als de webservice een 401 (unauthorized) oplevert.

## Les 9: practicum 4: Registratie **(OPTIONEEL)**

* Maak een registratie-formulier (bepaal zelf op welke pagina) om nieuwe gebruikers te registreren. Nieuwe gebruikers moeten standaard de rol **user** krijgen, dus hiervoor hoeft geen invoerveld aanwezig te zijn!
* Maak in de UserDao ook een methode om een nieuwe gebruiker toe te voegen.
* Maak een nieuwe webservice (bijv. UsersResource) om een nieuwe gebruiker te registreren (met een POST-request). Aanmaken van gebruikers mag alleen maar door gebruikers met de rol admin gedaan worden.

## Les 9: practicum 5: Gepersonaliseerde API **(OPTIONEEL)**

* Breid het registratie-formulier uit met extra gegevens (bijvoorbeeld e-mailadres, fysiek adres, telefoonnummer etc.). Sla deze gegevens ook op in de database.
* Maak een extra HTML-pagina (of alleen de webservice, die moet je dan met Postman of een andere tool testen) waarin de gebruiker zijn eigen gegevens kan inzien. Hiervoor heb je dus een extra webservice-methode nodig waarin je de SecurityContext laat injecteren door Jersey. Hiermee kan je de gegevens van de betreffende gebruiker als JSON retourneren.

## Les 9: practicum 6: Git & GitHub

Commit je wijzigingen van deze les naar je lokale repository en push naar GitHub!

# Les 10, practicum: Deployment op Heroku

Tot nu toe heb je steeds applicaties gedraaid op je eigen laptop. Dat is prima tijdens het ontwikkelen, maar je wilt natuurlijk ook dat andere mensen jouw applicatie kunnen gebruiken zodra deze klaar is! Het ***firstapp***-project gaan we daarom hosten op **Heroku**. Heroku is een online platform dat servers beschikbaar stelt voor gebruik door particulieren en bedrijven. Het voordeel is dat je zelf geen servers hoeft te onderhouden. Je hoeft je ook niet druk te maken of je internetverbinding wel snel genoeg is etc.

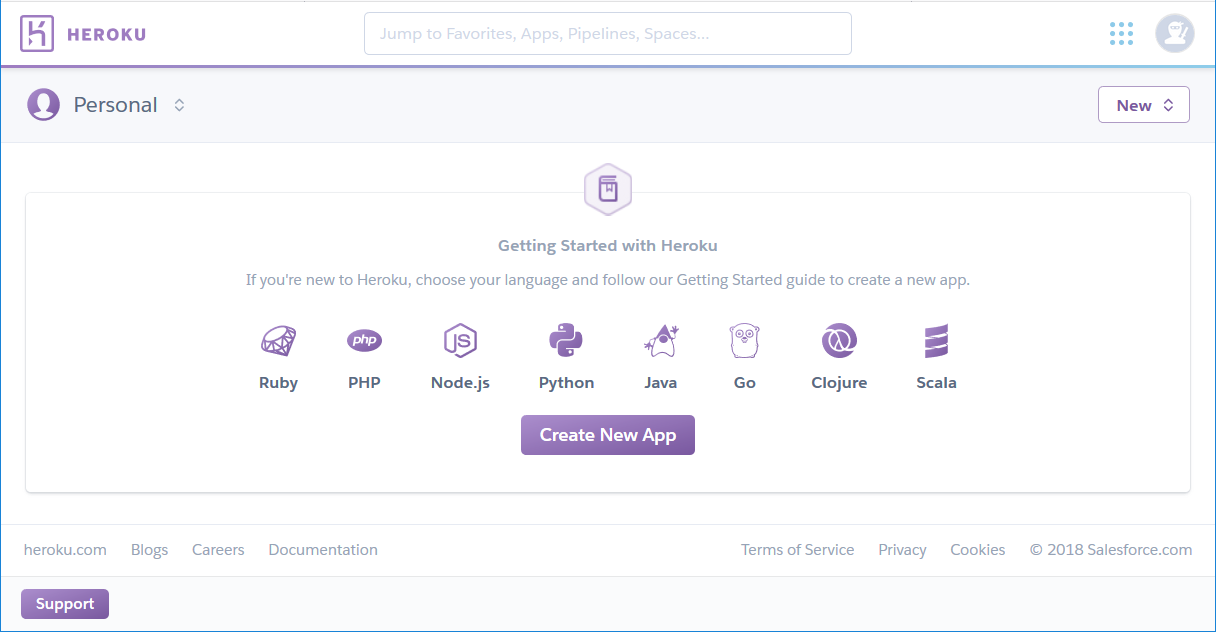
* In deze les deploy je de gehele applicatie vanaf GitHub op Heroku. Het is ook mogelijk om je project niet op GitHub te plaatsen en direct naar Heroku te kopiëren met de Heroku Command Line Interface (CLI). Echter, omdat je voor IPASS ook een code-repository moet opleveren kiezen we voor de route via GitHub. Daarnaast is deze werkwijze erg handig als je in een team werkt (er is dan 1 centrale GitHub-repository) en je eventueel geautomatiseerde tests wilt uitvoeren middels Continuous Integration. Dat laatste behandelen we echter niet in deze les.

## Les 10: practicum 1: Heroku

In dit practicum deploy je het firstapp-project op Heroku. Daarvoor zijn een Heroku-account en -applicatie nodig. Daarna stel je de Heroku-database in zijn er een paar kleine aanpassingen nodig zodat je project op Heroku draait. Tot slot kloon je de applicatie uit GitHub naar Heroku.

**Stap 1: Heroku-account aanmaken en applicatie creëren**

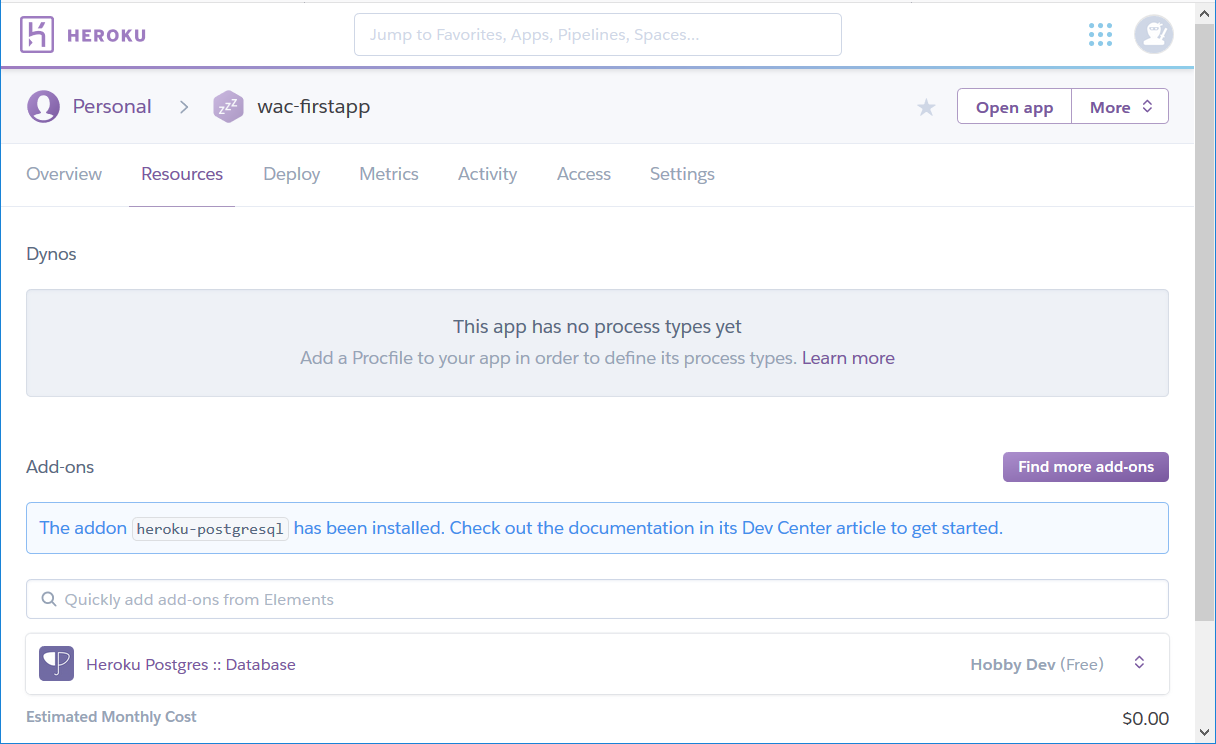
Ga naar <https://www.heroku.com>; maak een account aan. Na verificatie zie je het dashboard:



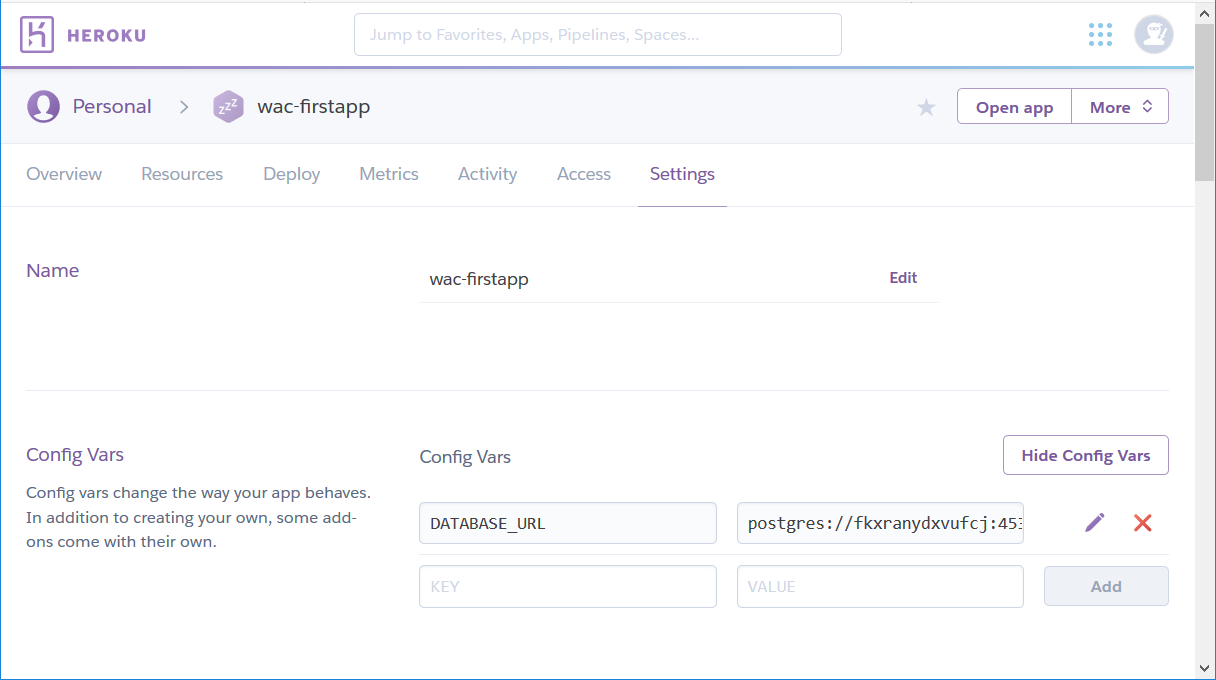
Klik hier op de knop **Create New App**. Kies op de volgende pagina een applicatienaam, selecteer de regio Europa en klik de knop **Create App**.

**Stap 2: Heroku-database aanmaken**

Je bent nu op de deployment-pagina gekomen, maar er moet eerst nog een database voor jouw applicatie gemaakt worden. Zoek op het tabblad **Resources** naar de **Heroku Postgres add-on.** Klik hierop en klik dan op **Provision** om de (gratis) add-on aan je project toe te voegen:



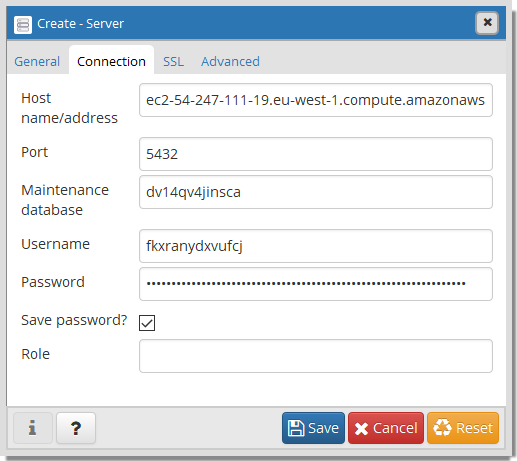
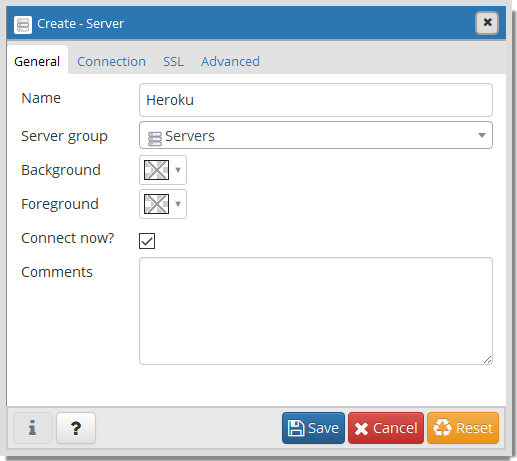
Op de settings-tab van je project kan je de database-URL als [environment variabele](https://en.wikipedia.org/wiki/Environment_variable) vinden:



De URL bevat de server-gegevens. Kopieer de URL en noteer deze in bijvoorbeeld Notepad.

Open pgAdmin 4 en rechtsklik op ‘Servers’. Kies **Create > Server…** om de server te benaderen. Voer de gegevens uit de URL in de wizard in (zie screenshots). De URL is als volgt opgebouwd:

**postgres://<user>:<password>@server:port/databasename**



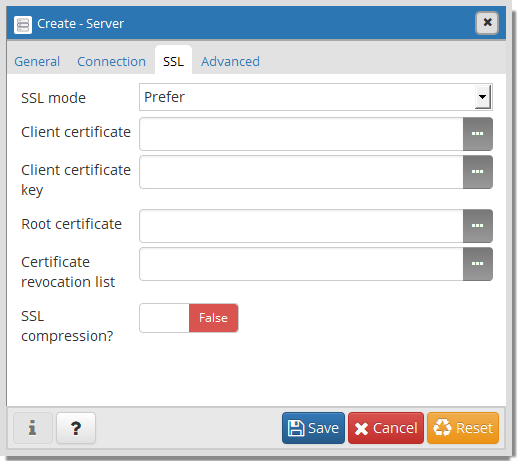
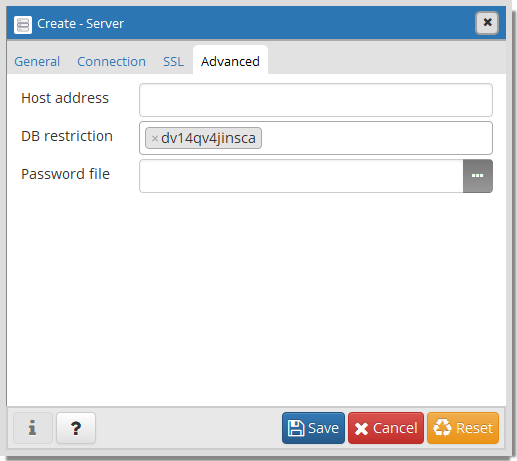
**server**

**port**

**DBname**

**user**

**password**



**DBname**

Laat je ‘DB restriction’ leeg, dan zie je alle databases op de server. Klik **Save**. De Heroku database moet nu gevuld worden. We maken een kopie (dump) van de lokale database. Klap in je eigen worlddb-database **schemas** open. Rechtsklik op **public** en kies [**Backup**](https://www.pgadmin.org/docs/pgadmin4/1.x/backup_dialog.html). Dan:

1. Geef een bestandsnaam (en locatie) op voor het dump-script.
2. Kies als Format: PLAIN, en gebruik als Encoding: UTF8. Open tabblad *Dump Options*.
3. Selecteer bij *Sections* **Pre-data, Post-data** en **Data**.Kies bij *Do not save* (de waarde **Yes**)voor **Owner**. Selecteer tot slot bij *Queries* **Use Insert Commands.**
4. Klik **Backup**.

* Als pgAdmin de installatiedirectory van jouw PostgreSQL-database niet kent, krijg je foutmeldingen. Stel dit eventueel in: File > Preferences > Paths > Binary Paths**.** Voer bij ‘PostgreSQL Binary Path’ de bin-directory in. Bijv. **C:\Program Files\PostgreSQL\10\bin**.

Open de Query Tool voor de **Heroku database**, en voer het dump-script hierin uit. De database is nu gemaakt en gevuld. Als zich problemen voordoen kan je eventueel het public schema van de Heroku database verwijderen en opnieuw aanmaken. Backup eventueel wel je data!

**Stap 3: Applicatie configuratie**

Heroku is een platform waar je allerlei (Java) applicaties op kunt draaien. Tomcat is een Java applicatie, en [kan dus op Heroku](https://devcenter.heroku.com/categories/java-support) draaien. Op Heroku is het gebruikelijk om de [Webapp Runner](https://github.com/jsimone/webapp-runner/tree/tomcat9) (een soort gezipte, single-file Tomcat-variant) te gebruiken. Lokaal is zo’n single-file meestal niet handig, maar op Heroku hoeven we Tomcat verder niet aan te passen. En Maven kan de Webapp Runner automatisch voor je downloaden. Verdere installatie is niet nodig.

Ook hebben we op Heroku nog de PostgreSQL driver nodig, en een library om datasources te kunnen opgeven in context.xml. Deze library zit niet standaard in de Webapp Runner. We vragen Maven om deze files te downloaden. Niet via dependencies, want ze maken geen deel uit van je project zelf, het zijn hulpmiddelen. Voeg de volgende build-tag aan je pom.xml toe:

<project>

...

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-dependency-plugin</artifactId>

<version>2.3</version>

<executions>

<execution>

<phase>**package**</phase>

<goals><goal>copy</goal></goals>

<configuration>

<artifactItems>

<artifactItem>

<groupId>com.github.jsimone</groupId>

<artifactId>webapp-runner</artifactId>

<version>9.0.8.0</version>

<destFileName>webapp-runner.jar</destFileName>

</artifactItem>

<artifactItem>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-dbcp</artifactId>

<version>9.0.8</version>

</artifactItem>

<artifactItem>

<groupId>org.postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>42.2.2</version>

</artifactItem>

</artifactItems>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

Als je het project op Heroku zet, [detecteert Heroku dat het een Maven-project is](https://devcenter.heroku.com/articles/java-support#activation). Heroku voert het commando **mvn** [**install**](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Build_Lifecycle_Basics)uit, download dan dus ook de files en plaatst deze in de map **target/dependency**. Ook maakt Maven in de map **target** een war-bestand van je project.

**Stap 4: PostgreSQL Datasource en Procfile**

De context.xml bevat nu de toegangsgegevens van jouw lokale database. Daar kan je op Heroku niets mee. Voeg in map META-INF een tweede context-file (**context-heroku.xml**) toe:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<Context>

<Resource name=*"jdbc/PostgresDS"*

url=*"${DB\_URL}"*

driverClassName=*"org.postgresql.Driver"*

auth=*"Container"*

type=*"javax.sql.DataSource"* />

</Context>

Zoals je ziet staan hier geen toegangsgegevens en URL in, maar wel een variabele: ***${DB\_URL}***. Heroku maakt naast de database-URL uit stap 2 van Heroku ook een tweede environment-variabele aan: de [JDBC\_DATABASE\_URL](https://devcenter.heroku.com/articles/connecting-to-relational-databases-on-heroku-with-java#using-the-jdbc_database_url). Deze bevat de volledige [connectie-string](https://jdbc.postgresql.org/documentation/80/connect.html) voor een PostgreSQL database, inclusief username en password. Helaas kan deze environment-variabele van het besturingssysteem (OS) niet direct gebruikt worden in de context.xml, dus daarom zullen we ‘m hieronder binnen Java beschikbaar maken onder de naam **DB\_URL**.

Als laatste maken we een [Procfile](https://devcenter.heroku.com/articles/procfile) waarin het commando komt te staan waarmee de Webapp Runner gestart moet worden, en waarin de juiste instellingen worden meegegeven (o.a. dus DB\_URL). Maak daarvoor in de hoofddirectory van je project het bestand Procfile aan (**zonder extensie, met hoofdletter ‘P’**) en plaats daarin het volgende commando (**alles op 1 regel!!**):

web: java -cp "target/dependency/\*" $JAVA\_OPTS -DDB\_URL=$JDBC\_DATABASE\_URL webapp.runner.launch.Main --enable-naming --context-xml src/main/webapp/META-INF/context-heroku.xml --port $PORT target/\*.war

**Wat doet dit commando?**

web: Heroku gebruikt instellingen voor een [webapplicatie](https://devcenter.heroku.com/articles/procfile#the-web-process-type)

java Java.exe wordt gestart met de volgende opties:

-cp "target/dependency/\*" [Classpath](https://docs.oracle.com/javase/9/tools/java.htm#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE__GUID-8CAE5D46-28F3-49B8-AAF5-EE79DC092E5D) waar Java zoekt naar klassen en libraries

$JAVA\_OPTS Environment variabele met [Heroku’s Java instellingen](https://devcenter.heroku.com/articles/java-support#environment)

-DDB\_URL=$JDBC\_DATABASE\_URL DB\_URL wordt beschikbaar als [Java property](https://docs.oracle.com/javase/9/tools/java.htm#GUID-3B1CE181-CD30-4178-9602-230B800D4FAE__GUID-F6A19E5C-0E9C-4376-BA5C-11571082F509)

webapp.runner.launch.Main Webapp Runner Main-klasse die gestart moet worden

--enable-naming Opgeven van resources in context.xml [inschakelen](https://github.com/jsimone/webapp-runner#options)

--context-xml src/main/webapp/META-INF/context-heroku.xml

De file die gebruikt moet worden i.p.v. [context](https://github.com/jsimone/webapp-runner#options).xml

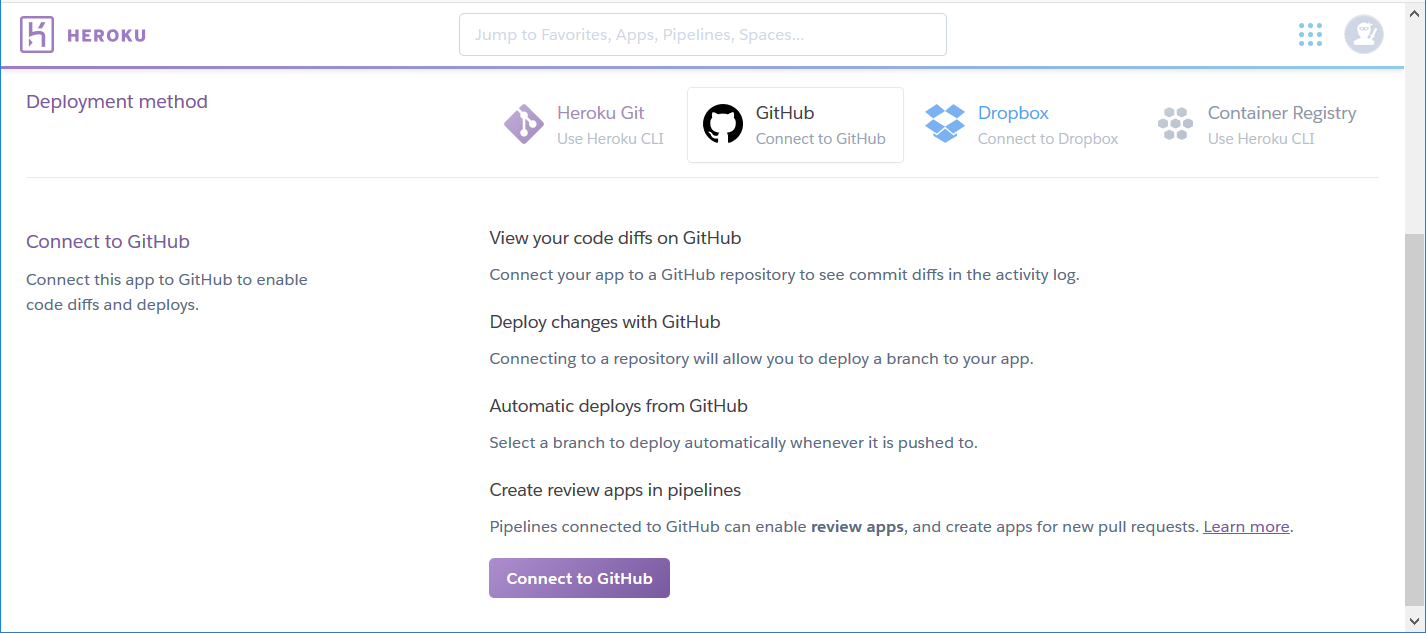
--port $PORT De applicatie moet beschikbaar worden op [deze](https://devcenter.heroku.com/articles/java-support#environment) [poort](https://github.com/jsimone/webapp-runner#options)

target/\*.war De webapplicatie die geladen moet worden.

Zodra Heroku met Maven het project heeft ge-build, wordt dit Procfile-commando uitgevoerd.

**Stap 5: Deployment via GitHub**

Controleer of je applicatie lokaal nog steeds goed werkt, add en commit in dat geval de wijzigingen lokaal. Push vervolgens ook de wijzigingen naar GitHub en open daarna het Heroku dashboard, tabblad Deploy. Kies **GitHub** als Deployment method en klik **Connect to GitHub**:



Autoriseer Heroku om toegang te krijgen tot jouw GitHub account. Zoek daarna in het Heroku dashboard de repository waarin jouw code staat, en kies **Connect**. Scroll tot slot in het dashboard naar onderen, en kies bij **Manual Deploy** om de master-branch te deployen.

Heroku haalt jouw code op uit GitHub en build het project met Maven. Volg of de Maven-build goed gaat. Daarna wordt de applicatie gestart. Niet altijd gaat alles meteen goed; build logs zijn terug te vinden via tab Activity, voor live logs van je applicatie: kies rechtsboven in het dashboard **More > View logs**. Gaat alles goed? Klik bovenin het dashboard dan op ‘**Open app**’.

## Les 10: practicum 2: Wijzigingen aanbrengen, commit & PUSH

**Let op:** misschien heb je in HTML- en/of JavaScript code nog directe paden naar REST-services of pagina’s opgenomen (zoals: localhost:8080 of localhost:8080/firstapp). Deze werken niet op Heroku. Gebruik relatieve paden!

🡺 Zorg dat alle pagina’s ook op Heroku correct werken!

## Les 10: practicum 3: Continuous Deployment **(OPTIONEEL)**

Maak lokaal een aparte [branch](https://www.atlassian.com/git/tutorials/using-branches) met de naam ‘**live**’. [Push](https://www.atlassian.com/git/tutorials/syncing/git-push) deze branch ook naar GitHub. Zorg in de **Deployment tab** van Heroku dat commits op deze GitHub-branch **automatisch** op Heroku gedeployed worden. Werk lokaal verder op de master-branch. Hierop kan je programmeren, testen, committen en pushen naar GitHub zonder dat gebruikers er last van hebben. Zodra je iets ‘live’ wilt zetten, switch je naar de live branch, en git [merge](https://www.atlassian.com/git/tutorials/using-branches/git-merge) master. Push de branch dan naar GitHub en deployment gaat verder automatisch!

## Les 10: practicum 4: Alleen HTTPS! **(OPTIONEEL)**

Als vanuit Heroku de applicatie opent, dan gebruik je standaard https. Certificaten zijn al ingesteld door Heroku. Maar, verander je in je browser https in http, dan kan je de pagina’s ook onbeveiligd opvragen. Dat zou eigenlijk niet mogelijk moeten zijn. Dat gaan we oplossen.

Elk request voor je applicatie [komt eerst bij de Heroku-servers terecht](https://devcenter.heroku.com/articles/http-routing#routing). Deze sturen het request intern onbeveiligd (http) door naar jouw applicatie, maar voegen er [extra headers](https://devcenter.heroku.com/articles/http-routing#heroku-headers) aan toe. O.a. een header (X-Forwarded-Proto) waarin staat of oorspronkelijk https is gebruikt.

Aan een webapplicatie kan je filters toevoegen. Requests kunnen dan door het filter gestuurd worden. Je kunt zo ook een filter maken dat kijkt of het gebruikte protocol https is. Zo niet, dan moet het filter een https-redirect naar de browser sturen. Een redirect stuurt de browser naar een andere URL. In ons geval hetzelfde adres, maar met https ervoor in plaats van http.

Een filter maken om https af te dwingen is typisch een voorbeeld van wat waarschijnlijk al eens door iemand anders is opgelost. Zie dit [voorbeeld](https://gist.github.com/qerub/8975333) of gebruik de simpeler (JEE8) variant in de [bijlage](#_HttpsFilter.java) van dit werkboek. Het filter uit het werkboek kan je net als servlets in een willekeurige package van je applicatie plaatsen; kies of maak een logisch package!

🡺 Pas een filter toe, en test of je de applicatie inderdaad niet meer via http kunt opvragen.

## Les 10: practicum 5: Werkplaats-deelvraag

De afgelopen weken heb je veel aspecten gezien van het ontwikkelen van webapplicaties. Er is behalve les 9 echter relatief weinig aandacht geweest voor security. Dit echter wel een erg belangrijk onderdeel. Je kunt op vele manieren iets doen aan de beveiliging van je applicatie. Is https bijvoorbeeld wel zo betrouwbaar, of zijn de JWT’s wel veilig voor [XSS](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))-attacks?

Voor de laatste opdracht ga je op zoek naar een verbetering op het gebied van security in de applicatie van de laatste weken. Het is de bedoeling dat je hier een werkplaats-onderzoek op los laat. Meer informatie daarover kun je [hier](http://onderzoek.hbo-i.nl/index.php/Categorie:Werkplaats) vinden. Geef antwoord op de volgende vraag:

**“Welke veiligheidsrisico’s bedreigen je applicatie en wat is de impact van tegenmaatregelen?”**

Zoek uit welke dreigingen er bij webapplicaties een rol spelen. Beschrijf:

* Welke soorten risico’s er spelen.
* Wat eventuele tegenmaatregelen kunnen zijn.
* Hoe omvangrijk het implementeren van die maatregelen is.
* Ga daarbij in op jouw concrete situatie (Tomcat, Java EE etc.).

Beschrijf dit gestructureerd in 1 tot 2 A4’tjes. Geef aan welke bronnen je hebt gebruikt en doe aanbevelingen voor welke maatregelen zo snel mogelijk uitgewerkt zouden moeten worden!

🡺 Voer dit onderzoek uit en lever het resultaat in bij je docent.

# Bijlage 1: Code-snippets

## FileBrowserServlet.java

**import** java.io.\*; **import** javax.servlet.http.\*; **import** javax.servlet.ServletException;

**import** java.net.\*; **import** java.nio.file.\*;  **import** javax.servlet.annotation.WebServlet;

@WebServlet(urlPatterns = "/filebrowser")

**public** **class** FileBrowserServlet **extends** HttpServlet {

**private** **static** **final** String ***htmlTemplate*** = "<!DOCTYPE html><html><title>FileBrowser 1.0</title><body>%s</body></html>";

**private** **static** **final** String ***dirTemplate*** = "<label><a href=\"?path=%s\">%s</a></label><br/>";

**private** **static** **final** String ***fileTemplate*** = "<label>%s</label><br/>";

**private** **static** **final** Path ***resourceBase*** = Paths.*get*("c:/Users/.../Documents/");

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

String pathParam = request.getParameter("path");

String decodedParam = (pathParam == **null** ? "" : URLDecoder.*decode*(pathParam, "UTF-8"));

Path requestedPath = Paths.*get*(***resourceBase*** + decodedParam);

**if** (Files.*isDirectory*(requestedPath)) {

requestedPath = requestedPath.toRealPath();

StringBuilder responseBuilder = **new** StringBuilder();

**for** (File dir : requestedPath.toFile().listFiles(file -> file.isDirectory())) {

String encodedPath = URLEncoder.*encode*(decodedParam + "/" + dir.getName(), "UTF-8");

responseBuilder.append(String.*format*(***dirTemplate***, encodedPath, dir.getName()));

}

**for** (File file : requestedPath.toFile().listFiles(file -> file.isFile()))

responseBuilder.append(String.*format*(***fileTemplate***, file.getName()));

response.setContentType("text/html");

PrintWriter pw = response.getWriter();

pw.write(String.*format*(***htmlTemplate***, responseBuilder.toString()));

}

}

}

## Rekenmachine.html

<!DOCTYPE html>

<html>

<title>JavaScript Rekenmachine</title>

<style>

**div***#frame* {

background-color: *gray*;

margin: *1em*;

padding: *0.5em*;

text-align: *center*;

width: *13.5em*;

}

**div***#display* {

background-color: *white*;

height: *1.5em*;

margin: *0.5em*;

padding-right: *0.3em*;

text-align: *right*;

}

**button** {

height: *3.5em*;

margin-bottom: *0.3em*;

width: *3.5em*;

}

</style>

<body>

<div id=*"frame"*>

<div id=*"display"*>0</div>

<button id=*"btn\_7"*>7</button>

<button id=*"btn\_8"*>8</button>

<button id=*"btn\_9"*>9</button>

<button id=*"btn\_div"*>/</button>

<button id=*"btn\_4"*>4</button>

<button id=*"btn\_5"*>5</button>

<button id=*"btn\_6"*>6</button>

<button id=*"btn\_prod"*>\*</button>

<button id=*"btn\_1"*>1</button>

<button id=*"btn\_2"*>2</button>

<button id=*"btn\_3"*>3</button>

<button id=*"btn\_min"*>-</button>

<button id=*"btn\_clear"*>C</button>

<button id=*"btn\_0"*>0</button>

<button id=*"btn\_eq"*>=</button>

<button id=*"btn\_plus"*>+</button>

</div>

</body>

</html>

## HttpsFilter.java

**!! WIJZIG HIERVOOR IN POM.XML DE JAVAX.SERVLET-API DEPENDENCY VERSIE NAAR 4.0.0 !!**

**import** java.io.IOException;

**import** javax.servlet.FilterChain;

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebFilter;

**import** javax.servlet.http.HttpFilter;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

@WebFilter(urlPatterns = "/\*")

**public** **class** HttpsFilter **extends** HttpFilter {

**public** **static** **final** String ***X\_FORWARDED\_PROTO*** = "X-Forwarded-Proto";

**public** **static** **final** String ***HSTS*** = "Strict-Transport-Security";

@Override

**public** **void** doFilter(HttpServletRequest request, HttpServletResponse response,

FilterChain filterChain) **throws** IOException, ServletException {

**if** (request.getHeader(***X\_FORWARDED\_PROTO***) != **null**) {

**if** (request.getHeader(***X\_FORWARDED\_PROTO***).equals("http")) {

String requestedURL = request.getRequestURL().toString();

response.sendRedirect(requestedURL.replace("http://", "https://"));

} **else** {

response.setHeader(***HSTS***, "max-age=31536000; includeSubDomains");

}

}

filterChain.doFilter(request, response);

}

}

**!! WIJZIG HIERVOOR IN POM.XML DE JAVAX.SERVLET-API DEPENDENCY VERSIE NAAR 4.0.0 !!**

* Het filter heeft de @WebFilter annotatie. Daarbij is het url-patroon opgegeven waarop dit filter van toepassing is: ‘**/\***’. Door de wildcard ‘\*’ wordt het filter toegepast op alle requests. Ook op localhost, maar daar doet het niets aangezien de [‘X-Forwarded-Proto’-header](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-Proto) alleen door proxy-servers als die van Heroku wordt ingesteld. Op localhost is afdwingen van HTTPS ook onnodig.
* De klasse HttpFilter die hier ge-extend wordt, bestaat pas vanaf JEE8 (servlet-api 4.0.0).
* Als een request op Heroku via HTTPS binnenkomt, stuurt het filter de [HSTS-header](https://developer.mozilla.org/nl/docs/Web/HTTP/Headers/Strict-Transport-Security) terug. Dit voorkomt dat gebruikers per ongeluk toch de onbeveiligde verbinding (http) gebruiken. Gebruikers typen namelijk meestal niet **https://** voor een url, maar bijvoorbeeld alleen ‘website.com’. De browser maakt daar zelf **http://**www.website.com van, maar de HSTS-header zorgt dat de browser onthoudt dat deze website ALTIJD alleen via HTTPS geladen moet worden. De allereerste keer dat een browser de website opvraagt is eenmalig wel een initieel http-request mogelijk. Dit voorkom je met de [preload property](https://developer.mozilla.org/nl/docs/Web/HTTP/Headers/Strict-Transport-Security#Preloading_Strict_Transport_Security).
* Er zijn nog meer HTTP-headers om je applicatie te beveiligen. Het [Mozilla Observatory](https://observatory.mozilla.org/) kan je Heroku-applicatie scannen en geeft suggesties welke headers je kunt toepassen.

# Bijlage 2: Command-line instructies

## Creëren van een Maven project

Het aanmaken van een Maven project kan via de commandline, maar dan moet je Maven wel apart [downloaden](http://maven.apache.org/download.cgi) en [installeren](http://maven.apache.org/install.html). Navigeer daarna met de commandline naar de directory waar jouw project moet komen te staan (bijvoorbeeld **/projects/**). Voer dan het volgende commando uit:

mvn -B archetype:generate -DarchetypeGroupId=**org.apache.maven.archetypes** -DarchetypeArtifactId=**maven-archetype-quickstart** -DgroupId=**nl.hu.v1wac** -DartifactId=**firstapp** -Dpackage=**nl.hu.v1wac.firstapp**

Dit commando laat Maven (mvn) een project genereren (archetype:generate). Gekozen is voor de collectie van projecttemplates met Id **org.apache.maven.archetypes**. Binnen die groep is gekozen voor de templates **maven-archetype-quickstart**. Overige parameters zijn eigenschappen van je project:

DgroupId groep van (eigen) projecten  
DartifactId naam van je project  
Dpackage volledige package voor jouw broncode

Hierna is een nieuw project aangemaakt in de directory **firstapp** (gelijk aan het DartifactId). Open deze directory en bekijk de aangemaakte projectstructuur.

Het project kan nu door de meeste IDE’s geopend of geïmporteerd worden. Het is IDE-afhankelijk wat daarvoor de exacte werkwijze is. Daarvoor zijn voldoende beschrijvingen en handleidingen te vinden. Zie voor **IntelliJ** bijvoorbeeld de [documentatie](https://www.jetbrains.com/help/idea/2016.1/importing-project-from-maven-model.html).

## Deployment op Tomcat

* Een beschrijving om een applicatie via IntelliJ op Tomcat te deployen, zie [hier](https://www.jetbrains.com/help/idea/2017.1/defining-application-servers-in-intellij-idea.html)!

Deployment op Tomcat (zonder IDE) kan op twee manieren; handmatig kopiëren van de WAR of via de Tomcat-manager. De eerste is het eenvoudigst, de tweede bespaart je steeds een copy-commando. Beide manieren worden beschreven. Kies zelf wat je het handigste vindt. **Voordat je begint, zorg dat Tomcat gestart is (in deze voorbeelden op poort 8080)!**

1. Handmatig kopiëren van de WAR:
   1. Voer (in de projectdirectory) het commando **mvn package** uit. Bij succesvolle BUILD zal er een .war bestand in de target directory gemaakt worden.
   2. Wijzig de naam van de .war naar een logische naam, bijv. firstapp.war.
   3. Kopieer het .war bestand naar de %TOMCAT\_DIR%/webapps directory. Dit kan je eventueel met een copy commando doen (Windows: copy, Linux: cp).
2. Via de Tomcat-manager:
   1. Voeg deze rollen & gebruikers in %TOMCAT\_DIR%/conf/tomcat-users.xml om de Tomcat manager te mogen openen:

<role rolename=*"manager-gui"*/>

<role rolename=*"manager-script"*/>

<user username=*"admin"* password=*"secret"*

roles=*"manager-gui,manager-script"* />

* 1. Open de Tomcat-manager interface: <http://localhost:8080/manager/>
  2. Deploy en open de applicatie op <http://localhost:8080/firstapp>

De eerste werkwijze is het meest eenvoudig te automatiseren. Je kunt eventueel een script schrijven dat deze commando’s telkens voor je uitvoert. Alle besturingssystemen hebben de mogelijkheid om via een commandline commando’s uit te voeren, en deze commando’s kan je bijvoorbeeld in Windows in een tekstbestand met de extensie .bat plaatsen. Dit bestand is uitvoerbaar. In Linux en MAC OS zijn vergelijkbare scripts te maken. Je hoeft dan alleen het script telkens uit te voeren, om de applicatie te deployen.

1. De servlet gebruikt soms Path-objecten. Een java.nio.Path is een nieuwe versie van java.io.File. [↑](#footnote-ref-1)
2. Het reële path wordt op deze manier ontdaan van o.a. [symbolic links](https://en.wikipedia.org/wiki/Symbolic_link)! [↑](#footnote-ref-2)