TICT-V1OODC1-15

OBJECT ORIENTED DESIGN & CONSTRUCTION

PROPEDEUSE HBO-ICT

TUTORIAL JAVA & POLYMER

GEEN TENTAMENSTOF!

NIET VERPLICHT!

Cursuseigenaar: Bart.vanEijkelenburg@hu.nl
Auteur: Bart van Eijkelenburg (SIE)

Versie: 1.0 (13-03-2018)

VERSIEBEHEER

#	Wijzigingen	Door:
1.0	Eerste versie ter ondersteuning van het Group Project (SIE)	B. van Eijkelenburg

INHOUDSOPGAVE

INLEIDING

De afgelopen weken heb je bij deze cursus kennisgemaakt met de programmeertaal Java. De andere cursus, TICT-V1AUI-15, heeft onder andere aandacht gegeven aan het bouwen van een GUI in Polymer. Deze tutorial wil je helpen om te begrijpen hoe je deze twee technieken met elkaar kunt combineren.

Een GUI die je ontwikkelt met behulp van Polymer, is over het algemeen bedoeld als webapplicatie. Daarmee bedoelen we een applicatie die je niet op je eigen computer installeert, maar opent via bijvoorbeeld een browser als Firefox, Chrome, Edge etc.

Een webapplicatie open je dus op je eigen computer, maar de applicatie staat eigenlijk op de computer waar de eigenaar van webapplicatie deze heeft ondergebracht (**gehost**). Bij een webapplicatie identificeren we daarom twee verschillende computers. De computer van de gebruiker noemen we **client**, de computer waar alle code van de webapplicatie staat noemen we **server**.

Hoe werkt dit nu eigenlijk? De browser van de client is een zeer krachtige applicatie die veel kan. De browser kan bijvoorbeeld HTML-code interpreteren, en op basis daarvan een zichtbare gebruikersinterface tonen. Ook weet een browser hoe hij CSS moet toepassen, en kan de browser zelfs JavaScript code uitvoeren. Daar maken we handig gebruik van bij een webapplicatie. De client vraagt namelijk altijd als eerste een HTML-pagina op van de server, en toont deze. Als de HTML-pagina CSS, plaatjes en/of JavaScript gebruikt, dan zullen deze pagina's ook geladen worden en (indien nodig) uitgevoerd worden. Je hebt dit de afgelopen weken bij de cursus TICT-V1AUI-15 toegepast.

Echter, aan alleen een client hebben we niet genoeg. Ten eerste omdat gebruikers meestal geen toegang moeten hebben tot bijvoorbeeld gevoelige gegevens. Er moet dus nog een afgeschermd gedeelte van de applicatie bestaan. Daarnaast hebben de verschillende gebruikers vaak met elkaars gegevens te maken. Ter illustratie kan je denken aan Osiris: als een docent een cijfer registreert, wil je dat cijfer later kunnen zien. Die informatie moet niet op de computer van de docent alleen blijven staan. De docent voert dit cijfer in op de client, en deze client moet dat cijfer doorsturen naar de server. De server fungeert als het ware als een centraal punt waar alle data wordt opgeslagen. Op de server moet programmacode draaien die dit cijfer ontvangt, in een database kan invoegen en kan beschermen tegen oneigenlijk gebruik van die data. De server heeft dus eigenlijk **twee** taken: ten **eerste** moet de server HTML, CSS, JavaScript, plaatjes etc. *serveren*. Ten **tweede** draait er ook gewoon programmacode op de server die berichten kan ontvangen en hierop kan reageren.

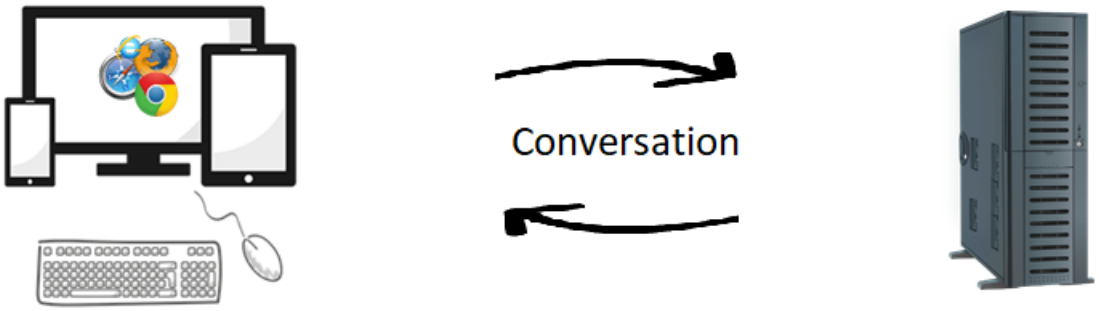
Op deze communicatie tussen client en server gaan we in blok 4 uitgebreid in, maar gedurende het Group Project zul je hier toch al gebruik van moeten maken. In feite is dit een behoorlijk complexe techniek, maar we hebben geprobeerd om de complexe details van deze techniek zoveel mogelijk voor je te verbergen. Deze tutorial laat je stap voor stap zien

hoe je een server-applicatie in Java kunt schrijven zonder alle bijzonderheden client-server communicatie te kennen. We gaan daarbij vooral in op ***hoe*** je dat doet. Een complete start-applicatie is bij het Group Project verstrekt, wij maken een minimale applicatie waarin alleen één volledig voorbeeld besproken zal worden.

CLIENT-SERVER COMMUNICATIE

Zoals in de inleiding al is uitgelegd, communiceren de client en server regelmatig met elkaar. Het zijn echter wel totaal verschillende soorten applicaties. De client gebruikt namelijk in veel gevallen (en zeker in ons geval met Polymer) JavaScript om data te versturen naar de server. JavaScript is de taal die elke moderne browser begrijpt.

Op de serverkant heb je echter veel meer keuze in welke taal je gebruikt. Wij gebruiken voor onze server Java als programmeertaal, maar dat zou ook Python, C#, Ruby, PHP, JavaScript etc. kunnen zijn. Iedere taal heeft z'n eigen sterke kanten, en er is dus niet een taal die het beste is. Maar omdat wij Java hebben geleerd in blok 3, gaan we daarmee verder.



Figuur 1: Communicatie tussen een client (links afgebeeld) en server.

Omdat er dus eigenlijk sprake is van twee verschillende applicaties die met elkaar communiceren, moet er een manier bedacht worden *waarop* ze met elkaar kunnen communiceren. We gaan tijdens deze cursus niet in op de details van het HTTP-protocol wat daarachter zit. Wat zich op de achtergrond afspeelt noemen we voor het gemak de ***conversation*** tussen client en server (zie ook Figuur 1). We komen daar later op terug.

Wel moeten we kijken naar de manier waarop we informatie vanuit de client (JavaScript) kunnen sturen naar de server (Java), en weer terug. Het moet een communicatiewijze zijn zijn die beide kanten begrijpen. We kiezen daarom (zoals dat gebruikelijk is) voor JSON.

JSON staat voor JavaScript Object Notation. JSON is een **notatiewijze**; een afspraak hoe we gegevens **noteren**. Doordat hier afspraken over gemaakt zijn, is er in vrijwel elke taal wel een library of functionaliteit beschikbaar waarmee JSON-tekst ingelezen of geproduceerd kan worden. JSON kent een beperkte hoeveelheid notatietypen:

- getallen `14.91` **(gehele èn komma-getallen)**
- strings "tekst" **(altijd met dubbele quotes)**

- booleans `true`, `false` (en `null`)
- array `[ "tekst", 14.92, null ]` **(vergelijkbaar met Python-list)**
- objecten `{ "bedrag": 1.1, "rood": false, "mylist": ["tekst", 4] }`

Vrijwel alle datastructuren in de meeste talen kunnen met deze notatietypen genoteerd worden. Een voorbeeld van een JSON bericht van de OpenWeatherMap API (zoals je moet toepassen bij TICT-V1AUI-15) is hieronder getoond:

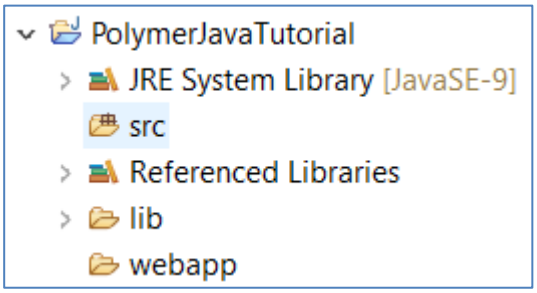
```
{
   "sys": {
      "weather": [
         {
            "main": "clouds",
            "description":"overcast clouds"
         }
      ],
      "main": {
         "temp": 289.5,
         "pressure":1013,
         "wind": {
            "speed": 7.31
         },
         "rain": {
            "3h":0
         }
      }
   }
}
```

Er is hier sprake van een object met 1 eigenschap/attribuut ('sys'), die op zijn beurt weer verwijst naar een object met 2 eigenschappen ('weather' en 'main'). Deze eigenschappen verwijzen ook weer naar objecten, die op hun beurt weer eigenschappen van strings en getallen hebben. Doordat dit bericht aan de JSON-eisen voldoet, kan dit bericht ingelezen worden in elke taal die JSON-ondersteuning heeft.

HET BASISPROJECT

Nu we gezien hebben welke vorm de berichten tussen client en server moeten hebben, kunnen we stap voor stap deze gedistribueerde applicatie bouwen. Hoewel er twee aparte applicatie-onderdelen zijn, kunnen we beiden wel gewoon in 1 Java-project bouwen. Dat is niet verplicht, maar voor ons wel het handigste omdat dit in de startapplicatie van het Group Project ook gedaan is.

1. Download van Sharepoint (map Java-sources) het basisproject dat we al voor je hebben klaargezet. Kies in Eclipse voor menu **File > Import**, en kies voor **General > Existing Projects into Workspace**. Kies vervolgens voor **Select archive file**, en selecteer het project dat je hebt gedownload. Klik **Finish**.



Als het goed is zie je nu een project in Eclipse verschijnen dat er qua structuur uitziet zoals hiernaast is weergegeven. De JRE System Library en **src** directory ken je van eerdere Java-projecten

Er is nu ook een **lib** directory bijgekomen. Daarin staat een library die we voor dit project nodig hebben, namelijk **jsonfileserver-1.0.jar**. Deze library is door de HU voor je klaargezet. Hierin zit onder andere een server (Jetty), en een Java library om met JSON te kunnen omgaan. Beiden zitten niet standaard in Java, dus zijn deze aan het **build path** van je project toegevoegd. Dat betekent dat de compiler ook in deze library zoekt naar klassen die je in je project gebruikt.

Tot slot is er de **webapp** directory. Hierin komt de Polymer GUI. Het project is nog geheel leeg, we gaan het project vanaf nu opbouwen. We beginnen ook met de Polymer GUI.

DE POLYMER GUI

Als het goed is heb je de Polymer CLI en bijbehorende tools al geïnstalleerd voor de cursus TICT-V1AUI-15. **Als dat inderdaad zo is, kan je de volgende vier stappen overslaan!** Voor de volledigheid zijn de minimaal benodigde installatieinstructies echter wel opgenomen**:**

2. Download en installeer Git.
3. Download en installeer de Node Package Manager.
4. Installeer Bower met het commando: **npm install -g bower**
5. Installeer de Polymer CLI: **`npm install -g polymer-cli`**

We gaan nu de Polymer applicatie door de Polymer CLI laten initialiseren.

6. Ga in een commandconsole naar de **webapp** directory van je project, of rechtermuisklik in Eclipse op de directory **webapp**, en kies voor **Show in Local Terminal > Terminal**. Dit laatste is alleen mogelijk als je, zoals eerder aangegeven, de Eclipse versie voor *JEE developers* hebt gedownload.
7. Initialiseer de Polymer app: **`polymer init`** Creëer een Polymer-2-applicatie, en kies als application/element naam voor **webapp** en **webapp-app**.

Zoals je waarschijnlijk al weet, maakt Polymer nu een standaard project aan, met een aantal bestanden. We gaan hier verder niet in op de diverse bestanden. Hiervoor verwijzen we naar het lesmateriaal van TICT-V1AUI-15.

We hebben niet alles nodig van de gegenereerde applicatie. We gaan de applicatie daarom iets aanpassen. Dat kan in Eclipse, maar je kunt de webapp directory ook openen met Visual Studio Code of Atom, een IDE die je mogelijk voor TICT-V1AUI-15 ook al gebruikt.

8. Open **webapp-app.html** in de directory webapp/src/webapp-app, en verwijder de **inhoud** van de template-tag (laat de tag zelf staan).
9. Wijzig **prop1** in **name**, en pas de value aan naar een lege string.

Als het goed is ziet de file er nu zo uit:

```
<link rel="import" href="../../bower_components/polymer/polymer-element.html">

<dom-module id="webapp-app">
  <template>
  </template>

  <script>
    /**
     * @customElement
     * @polymer
     */
    class WebappApp extends Polymer.Element {
      static get is() { return 'webapp-app'; }
      static get properties() {
        return {
          name: {
            type: String,
            value: ''
          }
        };
      }
    }

    window.customElements.define(WebappApp.is, WebappApp);
  </script>
</dom-module>
```

Deze dom-module gaan we voorzien van:

- Een button, in Polymer een **paper-button** element.
- Een invoerveld, in Polymer een **paper-input** element.
- Een element om AJAX calls uit te voeren, in Polymer een **iron-ajax** element.
- Een paragraph, een gewoon HTML-element.

De bedoeling is dat op deze pagina een naam ingevoerd kan worden, welke naar de server gestuurd zal worden. De server stuurt op zijn beurt dan een gepersonaliseerd bericht terug. Deze reactie van de server moet dan in de paragraaf komen te staan.

Als het goed is heb je alle Polymer-elementen al voorbij zien komen bij TICT-V1AUI-15. We voegen deze elementen ook aan ons project toe. We doen dit met behulp van bower.

10. Open een console in de webapp directory (zie stap 6).
11. Installeer paper-button: **`bower install --save PolymerElements/paper-button`**
12. Installeer paper-input: **`bower install --save PolymerElements/paper-input`**
13. Installeer iron-ajax: **`bower install --save PolymerElements/iron-ajax`**

De benodigde onderdelen zijn nu aan onze Polymer-applicatie toegevoegd, en we kunnen ze nu in onze dom-module importeren.

14. Voeg de volgende tags (bovenin) toe aan de **webapp-app.html** file:

```
<link rel="import" href="../../bower_components/paper-button/paper-button.html">
<link rel="import" href="../../bower_components/paper-input/paper-input.html">
<link rel="import" href="../../bower_components/iron-ajax/iron-ajax.html">
```

We kunnen nu de template-tag uitbreiden met de geimporteerde elementen. We zullen dit eerst doen, en zullen daarna ingaan op de ingevoegde code.

15. Voeg in de template-tag de volgende code toe:

```
<paper-input label="Uw naam:" value="{{name}}"></paper-input>
<paper-button on-click="clickHandler">GO</paper-button><br/>
<p>Response: <span id="response_text"></span></p>
```

De toegevoegde code behoeft misschien enige uitleg. De eerste regel levert een invoerveld op, met een automatisch gegenereerde label met de tekst ‘Uw naam:’. De value van dit invoerveld wordt gekoppeld aan de property ‘naam’ die we bij stap 9 hebben gemaakt.

De tweede regel zorgt voor een button met daarop de tekst ‘GO’. Als je erop klikt moet de JavaScript functie `clickHandler` aangeroepen worden. Die bestaat ook nog niet, maar zullen we in stap 17 toevoegen.

De derde regel is een paragraaf-element. Dit is een gewoon HTML-element, met daarin een eenvoudig span-element waarin het antwoord van de server getoond moet worden.

16. Voeg het **iron-ajax element** aan de template-tag toe:

```
<iron-ajax
  id="example_ajax_call"
  method="POST"
  url="/example"
  handle-as="json"
  on-response="handleAjaxResponse"
></iron-ajax>
```

Dit element is in staat om voor ons een bericht naar de server te sturen. De verschillende attributen worden hieronder toegelicht:

- **id**: een unieke naam waarmee je dit element in je code kunt aanspreken.
- **method**: de verzendwijze van de gegevens. Kies altijd voor POST.
- **url**: hiermee geef je aan welke code op de server uitgevoerd moet worden.
- **handle-as**: het antwoord wordt behandeld als JSON-tekst.
- **on-response**: de JavaScript functie die uitgevoerd moet worden als er antwoord is.

Misschien heb je al opgemerkt dat ook de JavaScript functie `handleAjaxResponse` er nog niet is. We deze in stap 18 toevoegen. Eerst voegen we de ontbrekende `clickHandler` toe.

17. Voeg de functie `clickHandler` toe aan de klasse `WebappApp` (in **webapp-app.html**):

```
clickHandler() {
  this.$.example_ajax_call.contentType="application/json";
  this.$.example_ajax_call.params = { "user": this.name };
  this.$.example_ajax_call.body = { "user": this.name };
  this.$.example_ajax_call.generateRequest();
}
```

De eerste regel geeft aan dat de body van het bericht in JSON verzonden moet worden. Op regel twee en drie wordt de naam in het invoerveld (**this.name**) op twee manieren aan het iron-ajax element toegevoegd. Namelijk als **parameter**, en als **body**. Dubbelop, en dus onzinnig. Maar we doen dat, zodat je in deze tutorial beide manieren ziet. Uitleg:

In het iron-ajax element is ingesteld dat we een bericht naar de url **/example** willen sturen. Stel dat in het invoerveld de naam 'Jan' staat. Toevoegen van de naam als parameter zorgt er dan voor dat de url wordt uitgebreid naar **/example?user=Jan**.

De tweede manier is het instellen van de body. Dat wil zeggen je als het ware een **bijlage** bij het bericht naar de server voegt. Doordat het contentType (van die bijlage) op 'application/json' is ingesteld, zal Polymer de body omzetten naar een JSON-bijlage:

```
{ name: "Jan" }
```

Tot slot geeft de methode `clickHandler` aan het iron-ajax element de opdracht op het bericht te verzenden (`generateRequest()`).

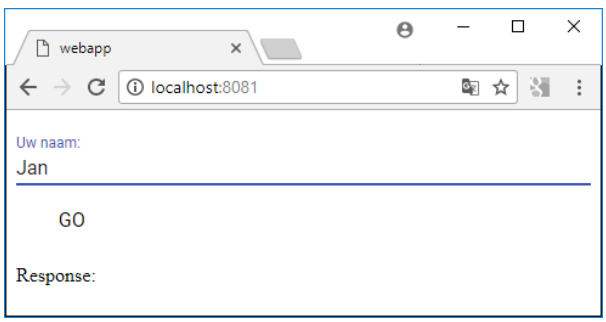
We zijn er nu bijna. Alle code om een bericht naar de server te versturen is geschreven. We willen alleen de reactie van de server ook nog op de pagina tonen. Dat doen we in de functie `handleAjaxResponse` (in stap 16 is deze functie aan het iron-ajax element gekoppeld).

18. Schrijf functie `handleAjaxResponse` in klasse `WebappApp` (in **webapp-app.html**):

```
handleAjaxResponse(result) {
  this.$.response_text.textContent = result.detail.response.message;
}
```

Variabele `result` bevat informatie over het verloop van het versturen van het bericht. Als het goed gegaan is, heeft deze variabele een attribuut `detail`, met weer een attribuut `response` met daarin de reactie van de server die in de paragraaf wordt geplaatst. We moeten dus in de Java applicatie programmeren dat het antwoord attribuut 'message' heeft.

Hiernaast zie je nog een screenshot van de GUI. **Let op:** we gaan straks 1 server gebruiken voor Polymer GUI en Java applicatie. Maar je kunt je GUI nu alvast testen door in de webapp directory het commando **polymer serve** uit te voeren.



Voor de volledigheid volgt hier nog de complete code van **webapp-app.html**:

```
<link rel="import" href="../../bower_components/polymer/polymer-element.html">
<link rel="import" href="../../bower_components/paper-button/paper-button.html">
<link rel="import" href="../../bower_components/paper-input/paper-input.html">
<link rel="import" href="../../bower_components/iron-ajax/iron-ajax.html">

<dom-module id="webapp-app">
  <template>
    <paper-input label="Uw naam:" value="{{name}}"></paper-input>
    <paper-button on-click="clickHandler">GO</paper-button><br/>
    <p>Response: <span id="response_text"></span></p>

    <iron-ajax
       id="example_ajax_call"
       method="POST"
       url="/example"
       handle-as="json"
       on-response="handleAjaxResponse"
    ></iron-ajax>

  </template>

  <script>
    /**
     * @customElement
     * @polymer
     */
    class WebappApp extends Polymer.Element {
      static get is() { return 'webapp-app'; }
      static get properties() {
        return {
          name: {
            type: String,
            value: ''
          }
        };
      }

      clickHandler() {
        this.$.example_ajax_call.contentType="application/json";
        this.$.example_ajax_call.params = { "user": this.name };
        this.$.example_ajax_call.body = { "user": this.name };
        this.$.example_ajax_call.generateRequest();
      }

      handleAjaxResponse(result) {
        this.$.response_text.textContent = result.detail.response.message;
      }
    }

    window.customElements.define(WebappApp.is, WebappApp);
  </script>
</dom-module>
```

DE JAVA APPLICATIE

Nu de GUI volledig staat, kunnen we de code schrijven die op de server de berichten van de GUI kan ontvangen en beantwoorden. Zoals eerder aangegeven hebben we zoveel mogelijk van de technieken die op de achtergrond spelen, te abstraheren. We maken dan ook, net als bij iedere andere GUI, gewoon een normale klasse aan met een main() methode.

19. Maak de volgende klasse aan in de **src** directory van je project (package **controller**):

```
package controller;

import java.io.File;
import server.JSONFileServer;

public class Application {
  public static void main(String[] args) {
    JSONFileServer server = new JSONFileServer(new File("webapp"), 8888);
    server.start();
  }
}
```

Listing 1: De klasse Application (naam mag je ook zelf bedenken), met de main() methode

De eerste regel code in de methode main() maakt een server-object aan. Achter de schermen is de Jetty server ingebakken in de gegeven library. Maar daarover hoef je jezelf dit blok nog niet druk te maken. Deze server moet voor ons:

- HTML, CSS, JavaScript etc. kunnen aanleveren aan de client. Deze resources zijn allemaal onderdeel van de Polymer GUI. Daarom is de **eerste parameter een File-object met de directory** waar die **resources** te vinden zijn.
- Berichten van de GUI kunnen ontvangen en beantwoorden.

De eerste taak kan onze server nu al uitvoeren. Voor de tweede moeten we nog code gaan programmeren. Overigens luistert de server op poort 8888, maar je mag ook een andere poort kiezen. Zolang er maar geen ander proces al van deze poort gebruik maakt.

CONTROLLERS

Je kunt je voorstellen dat er bij een grote applicatie veel verschillende soorten berichten naar de server gestuurd zullen worden. Bijvoorbeeld voor het inloggen, uitloggen, registreren, ophalen van accountgegevens en nog veel meer. We groeperen de berichten die bij elkaar horen daarom in **controllers**. Een controller is simpelweg een klasse met een methode die berichten kan ontvangen.

20. Maak de volgende controller aan in dezelfde package:

```java
package controller;

import server.Conversation;
import server.Handler;

public class InputController implements Handler {
  public void handle(Conversation conversation) {
    System.out.println(conversation.getRequestedURI());
  }
}
```

Listing 2: De controller InputController implementeert interface Handler, wat resulteert in het uitschrijven van methode handle(..)

De interface `Handler` schrijft voor dat de methode handle(Conversation) uitgeschreven moet worden. Je kunt deze interface eventueel bekijken door (in Eclipse) op de tekst 'Handler' te klikken en te kiezen voor F3. De methode handle(Conversation) kan vervolgens een bericht van de client afhandelen.

Det methode handle(..) krijgt een Conversation-object als parameter mee. Hierin staat alle benodigde informatie over een binnengekomen bericht. Verder bevat dit object ook een methode om een antwoord (JSON) terug te sturen naar de client. De huidige code zorgt er overigens alleen nog maar voor dat de URI waarvoor het bericht binnenkwam, in de console geprint zal worden!

We missen echter nog één link in onze applicatie. We moet onze controller namelijk nog **registreren** bij de server, **èn** we moeten aangeven op welke URI deze controller moet reageren. De server zal dan bij elk binnenkomende bericht bekijken welke URI werd aangevraagd, en de informatie doorsturen naar de juiste controller.

21. Registreer in de klasse Application, methode main(), de controller bij de server:

```java
InputController controller = new InputController();
server.registerHandler("/example", controller);
```

De controller zal berichten afhandelen voor URI's **die beginnen** met /example. We testen dit:

22. Voer de klasse Application uit. Als alles goed gaat, zie je de onderstaande uitvoer:

```
2018-03-14 00:41:45.180:INFO::main: Logging initialized @1070ms
2018-03-14 00:41:45.487:INFO:oejs.Server:main: jetty-9.3.z-SNAPSHOT
2018-03-14 00:41:46.256:INFO:oejs.ServerConnector:main: Started
            ServerConnector@5cf86874{HTTP/1.1,[http/1.1]}{0.0.0.0:8888}
2018-03-14 00:41:46.263:INFO:oejs.Server:main: Started @2186ms
Server started on http://localhost:8888/
```

Je kunt nu de applicatie (**in Google Chrome**) openen op het adres dat in de console wordt getoond. Probeer dit, voer een naam in en klik op GO. Als het goed is zal nu in de console de aangesproken url worden geprint:

```
/example
```

Lukt deze stap niet, controleer dan eerst in Eclipse of er foutmeldingen in de Eclipse-console zijn geprint. Is dat niet het geval, dan kan je in Chrome de developer tools (F12) openen. Controleer in de developer-tools-console of er JavaScript fouten zijn opgetreden, en of je deze kunt oplossen.

Ook als alles goed gegaan is, is het handig om de developer tools te openen. Je zult namelijk ontdekken dat ook dan een foutmelding in de console is geprint. Namelijk:

```
Uncaught TypeError: Cannot read property 'message' of null
  at HTMLElement.handleAjaxResponse (webapp-app.html:46)
  at HTMLElement.handler (template-stamp.html:92)
  at HTMLElement.fire (legacy-element-mixin.html:392)
  at HTMLElement._handleResponse (iron-ajax.html:558)
```

Dat komt doordat onze Java-applicatie nog geen antwoord teruggeeft, terwijl de Polymer GUI dat wel verwacht. We zullen de controller-code zodanig aanpassen dat de server de tekst ‘The server says: Hi Jan!’ terugstuurt als ‘Jan’ is ingevoerd in het invoerveld. We sturen een JSON bericht als antwoord. Daarvoor hebben we nog wel de naam nodig uit het gestuurde bericht. Omdat de GUI dit als parameter èn als JSON bericht heeft opgestuurd, kunnen we dat op twee manieren doen.

Eerste variant, maakt gebruik van de parameter:

```
public void handle(Conversation conversation) {
  String name = conversation.getParameter("user");

  JsonObjectBuilder objBuilder = Json.createObjectBuilder();
  objBuilder.add("message", "The server says: Hi " + name + "!");
  conversation.sendJSONMessage(objBuilder.build().toString());
}
```

Met de eerste regel vraag je eenvoudig de benodigde parameter van het Conversation-object op. De rest van de code is nodig om het JSON antwoord op te stellen. In Java kost dat wat meer code. We gaan er in de Polymer GUI vanuit dat we een JSON-object terugsturen met daarin minimaal de eigenschap ‘message’ (zie stap 18). Daarom vragen we aan de JSON-library om een object te creëren, en daar de eigenschap met waarde aan toe te voegen (coderegel 2 en 3 van bovenstaande methode). Tot slot zetten we het complete Java-object om naar een string en sturen we die terug (coderegel 4).

Tweede variant, maakt gebruik van het door de GUI meegestuurde JSON bericht:

```
public void handle(Conversation conversation) {
  JsonObject jsonIn = (JsonObject) conversation.getRequestBodyAsJSON();
  String name = jsonIn.getString("user");

  JsonObjectBuilder objBuilder = Json.createObjectBuilder();
  objBuilder.add("message", "The server says: Hi " + name + "!");
  conversation.sendJSONMessage(objBuilder.build().toString());
}
```

Het verschil is klein en zit 'm op de eerste twee regels: in plaats van de parameter op te vragen, vragen we nu de meegestuurde JSON-bijlage op als Java-object. Het Conversation-object zal de conversie van JSON-tekst naar Java-object voor je proberen uit te voeren. Dat lukt overigens alleen als het meegestuurde bericht correcte JSON tekst is.

We weten dat het JSON bericht in eerste instantie een JSON-object bevat (zie stap 17, regel 3 van de `clickhandler`), daarom kunnen we de casting op regel 1 van deze methode toepassen. Omdat we weten dat de eigenschap 'user' in het JSON bericht staat (en dat dit een string is), kunnen we die ophalen met `lJsonObjIn.getString("user")`.

23. Kies welke variant je het meest aanspreekt, en pas deze toe in jouw applicatie. **Let op:** als je aanpassingen maakt aan de Java-code, dan moet je de applicatie herstarten. Voor de GUI code is dat overigens niet noodzakelijk!

Opmerking 1: *Met het volgen van deze tutorial heb je de eerste stappen gezet naar het doorgronden van de startapplicatie die bij het Group Project gegeven is. Die applicatie werkt vergelijkbaar, maar is veel groter. Het helpt om de stappen die we in deze tutorial doorlopen hebben, goed te doorgronden. Dat kan enerzijds door de Polymer-workshops te volgen, en anderzijds door de theorie van TICT-V1OODC-15 op tijd in de vingers te krijgen.*

Opmerking 2: *Het maken van een JSON-structuur in Java kan erg onoverzichtelijk worden. Daarom zorgt de JSON-library voor handige return-waarden, waardoor je methoden aanroepen achter elkaar kunt plaatsen. De JSON-tekst van OpenWeatherMap (pagina 6) kan in Java bijvoorbeeld als volgt worden opgebouwd:*

```
public void handle(Conversation conversation) {
  JsonObject sys_obj = Json.createObjectBuilder()
      .add("sys", Json.createObjectBuilder()
          .add("weather", Json.createArrayBuilder()
              .add(Json.createObjectBuilder()
                  .add("main", "clouds")
                  .add("description", "overcast clouds")))
      .add("main", Json.createObjectBuilder()
          .add("temp", 289.5)
          .add("pressure", 1013)
          .add("wind", Json.createObjectBuilder().add("speed", 7.31))
          .add("rain", Json.createObjectBuilder().add("3h", 0)))
  ).build();

  conversation.sendJSONMessage(sys_obj.toString());
}
```

*Zoals je kunt zien eindigen de meeste regels **niet** met een puntkomma. Eigenlijk is dit dus een gigantisch lang statement, maar door handig in te springen kan je het toch overzichtelijk programmeren. Dit is mogelijk gemaakt doordat de add(..) methoden allemaal het object returnen waar ze zelf op aangeroepen zijn. Zo kan je steeds weer doorlussen.*