Full Stack Web Development

Tuur Vanhoutte

12 mei 2020

Inhoudsopgave

1 Client-server

1.1 Waarmee werkt een full-stack web developer?

- Hardware
- Databases
- API/back-end code: de logica
- Front-end code: visuele schil
- Project management
- . . .

1.2 We splitsen een applicatie in 3 delen:

1. Front-end
   - toont de data aan de gebruiker
   - HTML, CSS, Javascript
2. Back-end
   - haalt data uit de DB, verwerkt deze en geeft de nodige data terug aan de frontend
   - PHP, Python
3. Database
   - opslag van alle data
   - SQL-server, MySQL, MariaDB

1.3 Nood aan communicatie tussen 3 (verschillende) machines

- Front-end op de client
- Back-end op de webserver
- Data op de database(server)

1.4 Webserver

Webserver levert 'served' inhoud aan clients

1.4.1 Verschillende producten:

- Apache (since 1995)
- Ngingx (since 2004)
- IIS (since 1995)

1.5 TCP/IP

- Communicatie tussen de verschillende machines
- Fundament van client-server technologie
- Zorgt voor de verbinding
- Zegt niets over de aard en inhoud van de data: welke data/structuur/volgorde/...
- Zegt wel:
  - Afspraken over hoe de communicatie verloopt
  - HTTP(S), (S)FTP SMTP, POP3, IMAP, . . .

1.5.1 Stappenplan

1. Server zet IP en poort open
2. Server wacht op request
3. Client stuurt request naar IP:POORT
4. Server accepteert connectie
5. Data overdracht

1.5.2 TCP

= Transfer Control Protocol

- Poorten
- Bepaalt het communicatiekanaal voor een specifieke applicatie
- Server kan adhv poort beslissen voor welke applicatie er verbinding wordt gevraagd

1.5.3 IP

= Internet protocol, het adres van de server

1.6 HTTP

= **HyperText Transfer Protocol**

- Protocol voor webgebaseerde communicatie
- Computer + IP + Poort + HTTP-communicatie = WEBserver
- Tekstgebaseerd, dus vlot leesbaar
- Request-response: client stuurt verzoek, server antwoordt
- Stateless Protocol: requests onafhankelijk van elkaar: HTTP heeft geen geheugen

1.7 URL

= **Uniform Resource Locator**

Opbouw: protocol://server:poort/map/Bestand.txt?querystring

- Voorbeeld: http://www.howest.be:80/map/opleiding.php?querystring=mct
- Protocol: welk protocol gebruiken we (=http)
- Server: Naam of IP adres (naam omgezet via DNS)
- Poort: Welke TCP/IP poort willen we gebruiken (HTTP=80, HTTPS=443, ...)
- Map/Bestand.txt: websites bestaan uit mappen en bestanden
- Querystring: extra variabelen

1.8 HTTP Request

Webtechnologie is opgebouwd rond **Request - Response** model

Voorbeeld HTTP-request:

```
1 GET /wiki/Hoofdpagina HTTP/1.1
2 Host: nl.wikipedia.org
3 Connection: close
4 User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; nl; rv:1.8.0.3) Gecko/20060426
      Firefox 1.5.0.3
5 Accept: text/xml,text/html,text/plain,image/png,image/jpeg,image/gif
6 Accept-Charset: ISO-8859-1,utf-8
```

1.8.1 Request Headers

Bevat metadata over clients en servers (in beide response en request) zoals bvb:

1.8.2 Message Body

- Bij GET: altijd leeg
- Bij POST: POST variabelen bvb van formulier
- Client IP
- Cookies
- Browser info

1.8.3 Methods: welke vraag wordt er gesteld?

- GET
  - Vooral bedoeld om data op te halen
  - Variabelen worden meegestuurd via de querystring
- HEAD
- POST

  - Variabelen worden meegestuurd via de message body
  - Bewerkingen op data
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT
- PATCH

1.9 HTTP Response

HTTP-versie: 1.0/1.1/2.0

```
1 HTTP/1.1 200 OK
2 Date: Mon, 27 Jul 2009 12:28:53 GMT
3 Server: Apache/2.2.14 (Win32)
4 Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
5 Content-Length: 88
6 Content-Type: text/html
7 Connection: Closed
```

1.9.1 Status-codes: numerieke waarde, stelt het resultaat voor van de request

- 1xx: informele melding: gevolgd door een andere
- 2xx: request kan succesvol opgevolgd worden
- 3xx: request wordt omgeleid naar ergens anders (redirect)
- 4xx: onsuccesvol door client error: 404, 403, ...
- 5xx: onsuccesvol door serverfout, softwareproblemen, ...

1.9.2 Response

- Reason-Phrase: textuele melding bij status-code. Vb: OK, File Not Found
- Response-Headers: response metadata, vb: redirects, cookies, type van content
- Message-body: bevat de data van het resultaat, HTML, CSS, XML, ...

Surfen naar een pagina levert een hele lijst request/responses op.

1.9.3 Wat doet een webserver bij request om response te genereren?

Simple File Requests

- Opvragen inhoud van een fysieke file
- Statische informatie
- Geen verwerking nodig

1. Client stuurt request naar de server
2. Server leest request en zoekt de gevraagde file
3. Server maakt response, voegt headers toe
4. Server neemt de inhoud van de file en plakt die in de body
5. Server stuurt response terug naar client

Programming requests

- Data uit een database/databron
- Programmeertaal nodig
- Programmacode(script) op de server uitgevoerd

1. Client stuurt request naar de server, naar een file waar programmeercode instaat
2. Server leest request en zoekt het gevraagde script
3. Server kan code zelf niet uitvoeren, stuurt door naar applicatie die de code uitvoert
4. Applicatie voert code uit, genereert response + output & stuurt terug naar webserver
5. Server stuurt response terug naar client

2 Events

Events zijn gebeurtenissen:

- Gebruiker drukt op een knop
- Pagina is klaar met laden
- Formulier wordt verstuurd
- Scrollen door een pagina
- Verkleinen van de pagina
- ...

2.1 Events opvangen en afhandelen

Addresseren van het object waarop het event plaatsvindt (kan ook document zijn)

```
1 const btn = document.querySelector("button");
2 // Toevoegen van een eventlistener
3 btn.addEventListener(type, listener[, options]);
```

- Type = veel mogelijkheden: `https://developer.mozilla.org/en-US/docs/Web/Events`
- Listener
  - Indien geen parameters, een named function
  - Indien wel parameters, een anonieme function
- Options: `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener#Parameters`

2.2 DOMContentLoaded

Nu we events kennen kunnen we het uitvoeren van javascript uitstellen tot de pagina helemaal geladen is

- Load-event: wordt uitgevoerd als de hele pagina geladen is (HTML, CSS, imgs, scripts)
- DOMContentLoaded-event: wordt uitgevoerd als de DOM geladen is (HTML)

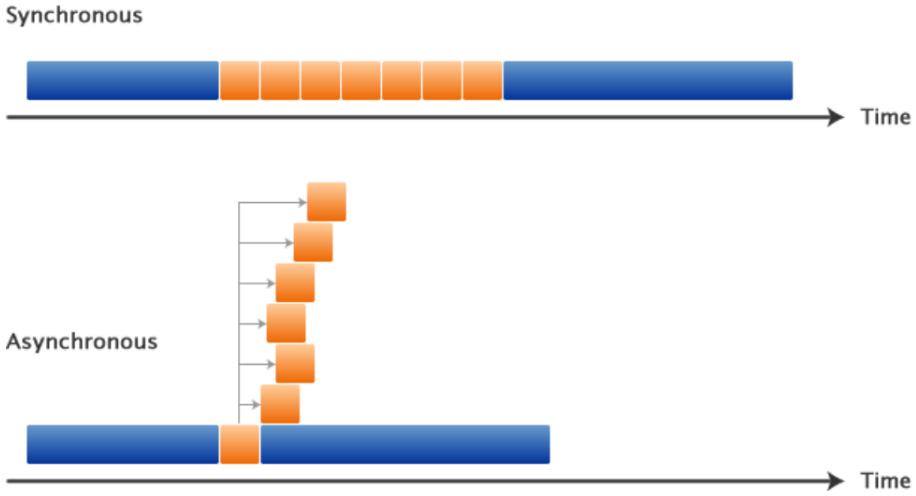
Hierdoor kunnen we de init uitstellen en de javascript opnieuw in de head plaatsen

2.3 Fetch

Fetch haalt asynchroon inhoud op van ergens anders

```
1  const url = "https://example.com/file.json";
2  const urldelay = "https://example.com/file.json?delay=3"
3  const urlerror = "https://example.com/onbestaandBestand.json"
4
5  const toonGebruikers = function(gebruikerArray){
6      console.log(gebruikerArray);
7      for(let user of gebruikerArray){
8          console.log(user.last_name);
9      }
10 }
11
12 const init = function(){
13     fetch(url).then(function(response) {
14         if(!response.ok){
15             console.log('response niet OK');
16             throw Error(`Er is een probleem bij het fetchen: ${response.status}`);
17         } else {
18             return response.json();
19         }
20     })
21     .then(function(json) {
22         //console.log(json);
23         arrUsers = json.data;
24
25         toonGebruikers(arrUsers);
26     })
27     .catch(function(error) {
28         console.error(`Fout bij het verwerken: ${error}`)
29     }
30 }
31
32 document.addEventListener("DOMContentLoaded" , init);
```

2.4 Synchronous vs Asynchronous



3 HTML Forms

3.1 Formulieren

Bestaat uit 2 delen:

1. Front-end: Website met formulierelementen
2. Back-end: Verwerking op de webserver (PHP, ASP, Python)

3.1.1 Front-end

Formulieren worden gemaakt in HTML en begrensd door <form>

```
<form method="post" action="bestemming">
    <!--
    inhoud van het formulier
    tags
    velden
    -->
</form>
```

3.1.2 Elementen

Zie `http://nativeformelements.com/` of `https://cbracco.github.io/html5-test-page/`

- Fieldset
- Legend
- Label
- Input
- Textarea

- Select
- Option
- Button

3.1.3 Tekst- & paswoordveld

```
1     <input type="text" name="veldnaam" value="inhoud veld" />
2     <input type="password" name="veldnaam" value="inhoud veld" />
```

Attributen:

- name
- id
- size: numeric value
- maxlength: numeric value
- value: text or numeric characters, assign an initial value to the textbox
- readonly: display only and cannot be edited
- autocomplete
- autofocus
- placeholder: text or numeric, brief info to assist the user
- required: value is vereist
- accesskey: keyboard characters
- tabindex: numeric, tab key order

3.1.4 Waarom een name en een ID?

- id = identificatie van een element, kan via Javascript benaderd worden
- name = naam van het element, de naam waarmee de waarde doorgestuurd wordt

Tip: gebruik altijd name en ID en geef ze dezelfde waarde.

```
1 <input type="text" name="naam" id="naam" placeholder="Geef hier uw naam in" />
```

3.1.5 Label

```
1 <label for="vrnm">Voornaam</label>
2 <input type="text" name = "vn" id="vrnm"/>
```

- Label element = span met speciale functie
- Verplicht te gebruiken!
- Naar accessibility en usability toe.

3.1.6 Textarea

Meerdere regels invoer

```
<textarea name="commentaar" rows="4" cols="50">Hier komt de tekst</textarea>
```

3.1.7 Select & Option (dropdownlist)

```
<select name="browser" size="4">
    <option value="ed">Edge</option>
    <option value="fifo">Firefox</option>
    <option value="chr">Chrome</option>
    <option value="op">Opera</option>
    <option value="an">Andere</option>
</select>
```

Multiple mogelijk $\Rightarrow$ checkbox!

3.1.8 Optgroup

```
<select>
    <optgroup label="Swedish cars">
        <option value="volv">Volvo</option>
        <option value ="saab">Saab</option>
    </optgroup>
    <optgroup label="German cars">
        <option value ="merc">Mercedes</option>
        <option value ="audi">Audi</option>
    </optgroup>
</select>
```

Opgepast: Label is een element & een attribuut!

3.1.9 Radiobuttons

- name-attribuut groepeert radiobuttons
- if-for zorgt voor een koppeling tussen het label en de input.

```

<input type="radio" name="browser"id="browser1"value="FF"/>
<label for="browser1">Firefox</label><br/>

<input type="radio" name="browser"id="browser2"value="IE" checked/>
<label for="browser2">Edge</label>
```

3.1.10 Aankruisvakje

- name-attribuut groepeert radiobuttons
- if-for zorgt voor een koppeling tussen het label en de input.

```
<input type="checkbox" name="fruit[]"id="appels" value="appel" />
<label for="appels"> appels </label><br/>

<input type="checkbox" name="fruit[]"id="banaan" value="appel" />
<label for="banaan"> banaan </label>
```

3.1.11 Fieldset/legend

Kan gebruikt worden om een aantal controls van een formulier te groeperen door er een kader om te plaatsen.

```
<fieldset>
    <legend> Health information:</legend>
    Height: <input type="text" ... />
    Weight: <input type="text" ... />
</fieldset>
```

3.1.12 Verzenden en reset

- value-attribuut = tekst op de knop

```
<input type="submit" value="Verzend bericht" name="verzend" />
<input type="reset" value="Wis bericht" name="wis" />
```

3.1.13 Andere velden

```
<input type="email" name="..." />
<input type="time" name="..." />
<input type="url" name="..." />
<input type="number" name="..." />
<input type="search" name="..." />
<input type="date" name="..." />
<input type="range" name="..." />
```

3.2 Formulieren verzenden

3.2.1 Form method=GET/POST

GET	POST
parameters in de url	parameters in de body
gebruikt om documenten op te vragen	gebruikt om data te wijzigen
maximum lengte	geen maximum lengte
OK om te cachen	niet OK om te cachen
niet gebruiken voor wijzigen op de server	OK om te gebruiken voor wijzigingen op de server

3.3 Valideren van een form

Met ’required’ en ’pattern’

- HTML5 validate
- Javascript validatie

Dit kunnen we koppelen aan CSS selectoren :valid en :invalid

4 Flask routing

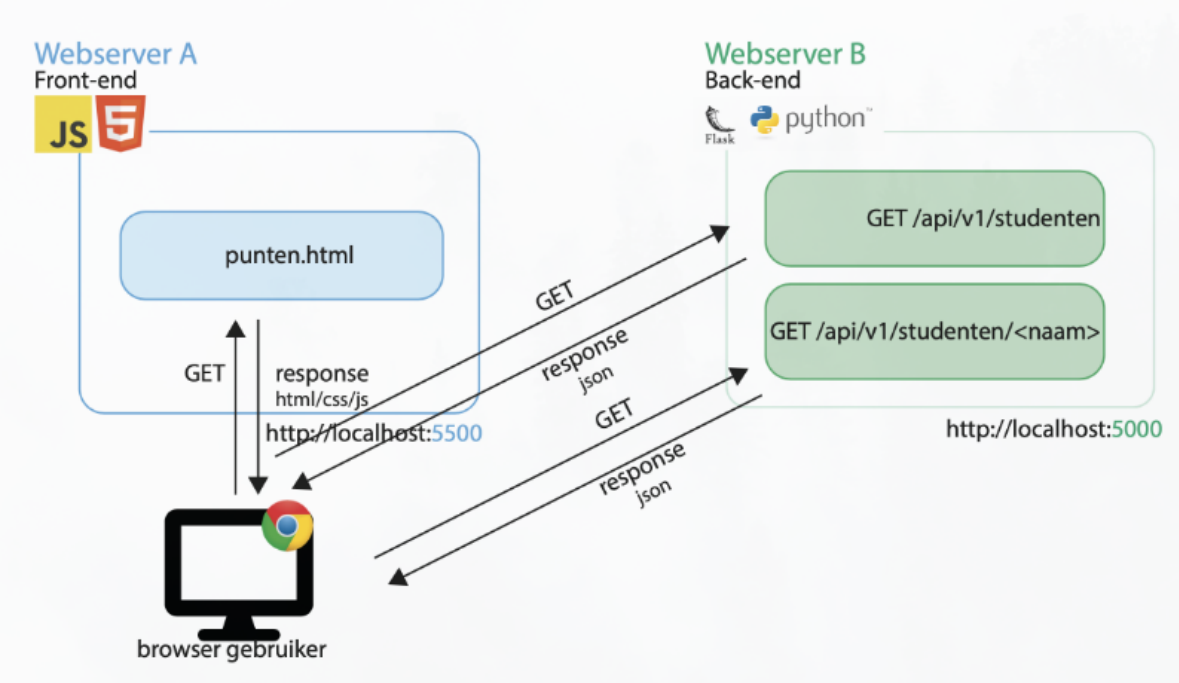
4.1 Backend

Flask is een micro webframework in Python

- Micro: bevat enkel de basisbenodigdheden, maar is uitbereidbaar
- webframework: web server gateway interface

Wij zullen Flask enkel gebruiken als backend server!

Flask is in staat om webrequests op te vangen en een response terug te sturen. Wij zullen dit gebruiken om via webrequests json data op te vragen.



Figuur 1: Werking Flask

In de rest van de module zullen we dus gebruik maken van een. . .

- Frontend server via de Live Server ⇒ werkt op 127.0.0.1:**5500**
- Backend server via Flask (Werkzeug) ⇒ werkt op 127.0.0.1:**5000**

Opmerkingen:

- Goed opletten wat op welke poort draait, het nummer kan ook anders zijn!
- Voor project1 zal Live Server vervangen worden door Apache

4.1.1 Opzetten van een Flask project in Visual Studio Code

1. Opzetten van een Virtual Environment (venv): een kopie van je python interpreter waarop je verschilllende packages installeert om de hoofdinterpreter niet te vervuilen
   - Open een terminal in Visual Studio Code
   - Ga naar de root van de FSWD-map
2. Typ dit in een terminal:

```
# windows
py -m venv venv_fswd
# mac/linux
python3 -m venv venv_fswd

```

3. Visual Studio Code zal detecteren dat je een virtual environment aanmaakt en zal je vragen dit toe te voegen aan je workspace. Doe dit!
4. Visual Studio Code zal vragen om **pylinter** toe te voegen aan je interpreter. Doe dit!
5. Indien er een update van **pip** is, installeer die update!
   - **pip** is de standaard package installer van python.

4.1.2 Flask in python

```
from flask import Flask

app = Flask(__name__) # __name__ = naam van huidige module

if __name__ == "__main__": # zal uitgevoerd worden indien dit script rechtstreeks uitgevoerd wordt (dus niet via een import vanuit een andere file)
    app.run(host='127.0.0.1', port='5000', debug=True) # debug=True zorgt ervoor dat de server opnieuw wordt gestart bij wijziging van je code.

# @ = function decorator: zal de onderstaande functie koppelen aan een pad en een method. Je kan dit vergelijken met een form method en action, standaard method is hier GET
@app.route('/')
def start():
    return "Hallokes"

# nieuwe route toevoegen
@app.route('/page2')
def naam_van_een_nieuwe_functie():
    return "Dit is een tweede pagina"

# deel van de URL variabel maken
@app.route('/page/<nr>')
def pagina(nr):
    return f"Dit is pagina {nr}"

# we kunnen een URL gebruiken om te GET'en, maar ook om te POST'en
```

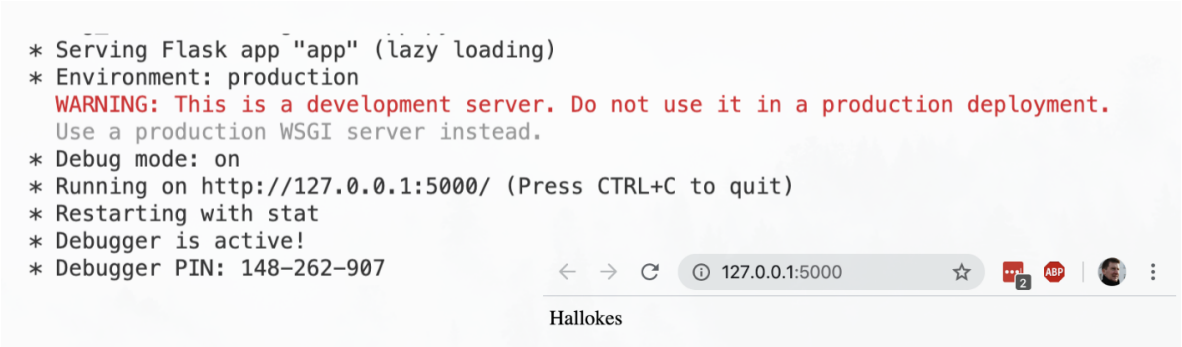
```python
@app.route("/getorpost", methods=['GET', 'POST'])
def getorpost():
    if request.method == 'GET':
        return "GET"
    else:
        return "POST"

# returnwaardes
@app.rotue('/')
def index():
    # return waarde, statuscode
    return jsonify({'name': Dieter, 'age': 39, 'city': 'Kortrijk'}), 200

# dataverwerking
@app.route('/showformdata', methods=['POST'])
def pagina1_fetch_json():
    naam = request.form['naam']
    voornaam = request.form['voornaam']

    persoon = {'naam': naam, 'voornaam': voornaam}

    return jsonify(persoon=persoon), 200
```



Figuur 2: Running Flask

4.2 Frontend

4.2.1 Datahandler

Om het opvragen van de data makkelijker te maken vanuit de frontend schreven we voor jullie een datahandler:

```javascript
const handleData = function(url, callbackFunctionName, callbackErrorFunctionName =
    null, method = 'GET', body = null) {
    fetch(url, {
      method: method,
      body: body,
      headers: { 'content-type': 'application/json' }
    })
      .then(function(response) {
        if (!response.ok) {
          console.warn(`>> Probleem bij de fetch(). Statuscode: ${response.status}`);
          if (callbackErrorFunctionName) {
            console.warn(`>> Callback errorfunctie ${callbackErrorFunctionName.name}(
    response) wordt opgeroepen`);
```

```
12              callbackErrorFunctionName(response);
13            } else {
14              console.warn('>> Er is geen callback errorfunctie meegegeven als parameter
    ');
15            }
16          } else {
17            console.info('>> Er is een response teruggekomen van de server');
18            return response.json();
19          }
20        })
21        .then(function(jsonObject) {
22          if (jsonObject) {
23            console.info('>> JSONobject is aangemaakt');
24            console.info(`>> Callbackfunctie ${callbackFunctionName.name}(response)
    wordt opgeroepen`);
25            callbackFunctionName(jsonObject);
26          }
27        });
28      /*.catch(function(error) {
29          console.warn(`>>fout bij verwerken json: ${error}`);
30          if (callbackErrorFunctionName) {
31            callbackErrorFunctionName(undefined);
32          }
33        })*/
34    };
```

De functie werkt met een **callbackfunctie** = een functie die zal uitgevoerd worden nadat de handleData functie uitgevoerd is.

```
const handleData = function(url, callbackFunctionName,
callbackErrorFunctionName = null, method = 'GET', body = null) {
```

Figuur 3: De callbackfunctie is de 2^{de} parameter van de handelData functie

5 Flask CRUD

5.1 CRUD

- **C**reate (of insert): toevoegen van nieuwe gegevens
- **R**ead (of select): opvragen van gegevens
- **U**pdate: wijzigen van gegevens
- **D**elete: verwijderen van gegevens

In Flask gebruiken we **Database.py** om de CRUD-functionaliteit te gebruiken in onze backend.

In **DataRepository.py** schrijven we alle CRUD-acties:

5.2 CRUD-acties zonder parameters

```
1 @staticmethod
2 def read_klanten():
3     sql = "SELECT * FROM tblKlant"
4     return Database.get_rows(sql)
```

5.3 CRUD-acties met parameters

```
@staticmethod
def read_klant(id):
    sql = "SELECT * FROM tblKlant WHERE KlantID = %s"
    params = [id]
    return Database.get_one_row(sql, params)
```

5.4 Andere CRUD-acties

Andere CRUD-acties doen we met de methode Database.execute_sql(sql, params):

```
    @staticmethod
    def delete_klant(id):
        sql = "DELETE FROM tblKlant WHERE KlantID = %s"
        params = [id]
        return Database.execute_sql(sql, params)
```

6 Dynamic eventlisteners

Soms wil je iets tonen aan de user (zwart, rood, ...), maar iets anders bijhouden 'achter de schermen' (#000, #F00, ...). Hiervoor gebruiken we een data-attribute (`https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes`).

6.1 Data-attributes

Data-attributes zien er in de HTML-code zo uit:

```
<ul>
    <li class="example" data-code="#000">black</li>
    <li class="example" data-code="#F00">red</li>
    <li class="example" data-code="#0F0">green</li>
    <li class="example" data-code="#00F">blue</li>
    <li class="example" data-code="#FF0">yellow</li>
    <li class="example" data-code="#0FF">cyan</li>
    <li class="example" data-code="#F0F">magenta</li>
</ul>
```

We hebben 2 mogelijkheden om deze data-attribute aan te spreken:

```
// met getAttribute():
this.getAttribute("data-naam");
// met het dataset variabele:
this.dataset.naam; // Let op: niet data-naam!
```

6.2 Eventlisteners

Voorbeeld uit de treinoefening:

```
function listenToSelectStation(){
    // selecteer alle options
    let options = document.querySelectorAll(".js-station");

    // loop over alle options
    for (let option of options) {
        // voeg voor elke option een eventlistener toe
        option.addEventListener("click", function() {
            // haal het destinationID uit de dataset:
            currentDestinationID = this.dataset.destinationId;
            // roep de getter op met de huidige destinationID
            getTrainsByDestination(currentDestinationID);
        });
    }
}
```

7 Javascript - Support

7.1 URL Queryparams

Manier om de querystring van een URL uit te lezen in JavaScript.

- Vroeger kon dit enkel door gebruik te maken van reguliere expressies en string splitting functies: ingewikkeld, veel werk
- Sinds Jan 2016:URLSearchParams = API: makkelijk,weinig werk
- Ondersteuning? `CanIUse.com`

```
// url = https://www.example.com/index.html?waarde=hallo
console.log(window.location.search) // returnt "?waarde=hallo"

// Met URLSearchParams:
const querystring = new URLSearchParams(window.location.search);
console.log(querystring.get("waarde")); // print "hallo"

//querystring.has(): returnt een boolean
if (querystring.has("waarde2")) { ... }
```

7.2 1 app.js voor meerdere html bestanden

7.2.1 Voor grote projecten:

- Gemeenschappelijke code in 1 bestand
- Aparte code in aparte bestanden
- Nadeel: code staat verspreid en is niet makkelijk te onderhouden

7.2.2 Voor kleine projecten:

- Met een if-statement in init() die checkt welke functie op welke pagina moet worden uitgevoerd.
- Alle code in 1 bestand
- Gemeenschappelijke code bovenaan

```
const functie_die_ik_enkel_op_pagina1_nodig_heb = function() {
    functie_die_ik_op_elke_pagina_nodig_heb("ik word uitgevoerd op pagina 1");
};

const functie_die_ik_enkel_op_pagina2_nodig_heb = function() {
    functie_die_ik_op_elke_pagina_nodig_heb("ik word uitgevoerd op pagina 2");
};

const init = function() {
    if (document.querySelector(".js-pagina1")) {
        functie_die_ik_enkel_op_pagina1_nodig_heb();
    } else if (document.querySelector(".js-pagina2")) {
        functie_die_ik_enkel_op_pagina2_nodig_heb();
    }
};
```

7.3 this vs event.target

- Het gebruik van het keyword **this** bij dynamische eventhandlers:
  - this = het element waarop het event getriggerd werd
- Het gebruik van het **event** bij dynamische eventhandlers:
  - evt = alle properties en method die verband houden met het event

```
const functie_die_ik_op_elke_pagina_nodig_heb = function(message) {
    document.querySelector(".js-result").innerHTML += `${message}<br/>`;
    console.log(message);
};

const doeIets = function(element) {
    functie_die_ik_op_elke_pagina_nodig_heb(element);
    element.innerHTML = "Hier heb ik reeds op geklikt.";
};

const doeIetsAnders = function(evt) {
    functie_die_ik_op_elke_pagina_nodig_heb(evt.target);  //het element waar op
    geklikt wordt
    functie_die_ik_op_elke_pagina_nodig_heb(evt.screenX); //toont waar de x-positie
    waar je geklikt hebt
};


const init = function() {
    let items = document.querySelectorAll(".c-container__item");

    for (const item of items) {
        item.addEventListener("click", function(e) { //belangrijk dat je 'e' meegeeft.
            //gebruik van this
            doeIets(this); //this moet niet meegegeven worden
            //gebruik van het event
            doeIetsAnders(e);
        });
```

```
27      }
28  };
```

8 Realtime communicatie

8.1 Websockets

- Websocket is een netwerkprotocol dat full-duplexcommunicatie biedt over een TCP verbinding
- Een verbinding over ws:// of wss://
- Om een websocket te openen wordt een speciale HTTP-request gestuurd met een 'connection: upgrade'-aanvraag
- Hierop stuurt de server een response met code '101 Switching Protocols'
- Daarna kan er gecommuniceerd worden zonder http-request of http-responses, maar wel via zogenaamde messages

8.1.1 Zelf programmeren

- Kan perfect zelf, maar we kiezen voor een library
- Socket.io: een javascript library en de bijhorende Python extension Flask-socketio
- Een gedeelte is al geschreven en we kunnen dit sneller toepassen

8.2 Socket.io

8.2.1 front-end

Laad de code van Socket.io in de header in, voor je je eigen app.js inlaadt:

```
1  <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/2.3.0/socket.io.js"
       integrity="sha256-bQmrZe4yPnQrLTY+1gYylfNMBuGfnT/HKsCGX+9Xuqo=" crossorigin="
       anonymous"></script>
```

Om nu een websocket connectie naar de backend/server te openen typ je:

```
1  const lanIP = `${window.location.hostname}:5000`;
2  const socketio = io(lanIP);
```

Eerst halen we de huidige hostname op en plakken daar :5000 aan, zo komen we aan het adres van de backend. Daarna openen we de connectie naar de backend met io(adres_van_backend)

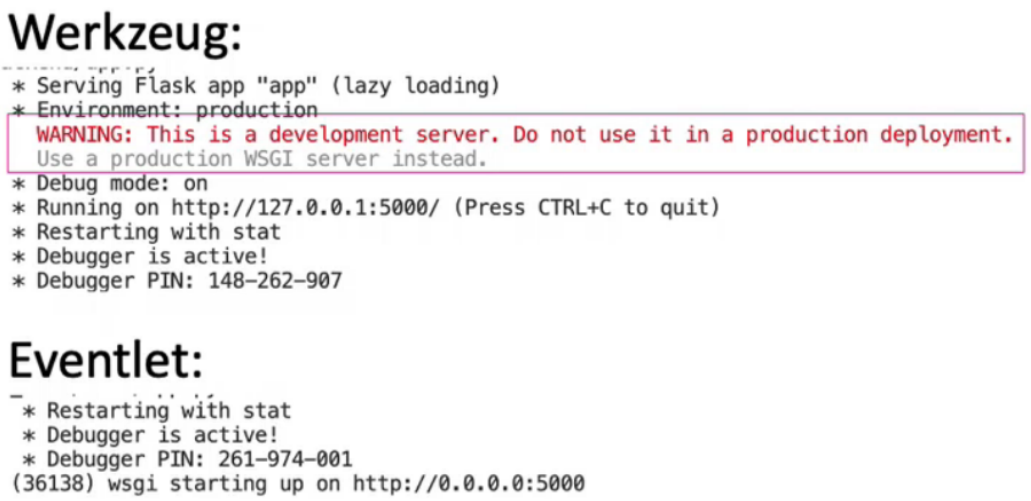
8.2.2 back-end

```
1  pip install flask-socketio
2  # nieuwe development server, want de flask server ondersteunt geen sockets
3  pip install eventlet
```

Nu kunnen we in onze app.py de code toevoegen om een SocketIO server op te zetten:

```
1 from flask_socketio import SocketIO, send, emit
2
3 socketio = SocketIO(app, cors_allowed_origins="*")
4
5 if __name__ == '__main__':
6     # socketio heeft een wrapper waarin we de app moeten meegeven.
7     # zo worden de socketio eigenschappen toegevoegd aan de app.
8     # door 0.0.0.0 te gebruiken kan de server ook van buitenaf worden aangesproken.
9     socketio.run(app, host="0.0.0.0", port=5000, debug=True)
```

Indien je je pythoncode nu runt, zal Python de nieuwe server (eventlet) opstarten ipv Werkzeug (de webserver van Flask). Je kan dit herkennen aan volgende code:



Figuur 4: Werkzeug vs Eventlet

8.2.3 Werking

Hoe kan je nu 'messages' sturen van de front-end naar de backend?

Je gebruikt **socketio.send** of **socketio.emit** om een boodschap te versturen, waar socketio de variabele is die onze verbinding voorstelt.

Met socketio.send:

front-end ⇒ back-end (in JavaScript):

```
1 socketio.send("dit is een boodschap");
```

back-end ⇒ front-end (in Python):

```
1 socketio.send("dit is een boodschap")
```

Met socketio.emit:

front-end ⇒ back-end (in JavaScript):

```
1 socketio.emit("naam", "dit is een boodschap");
```

back-end ⇒ front-end (in Python):

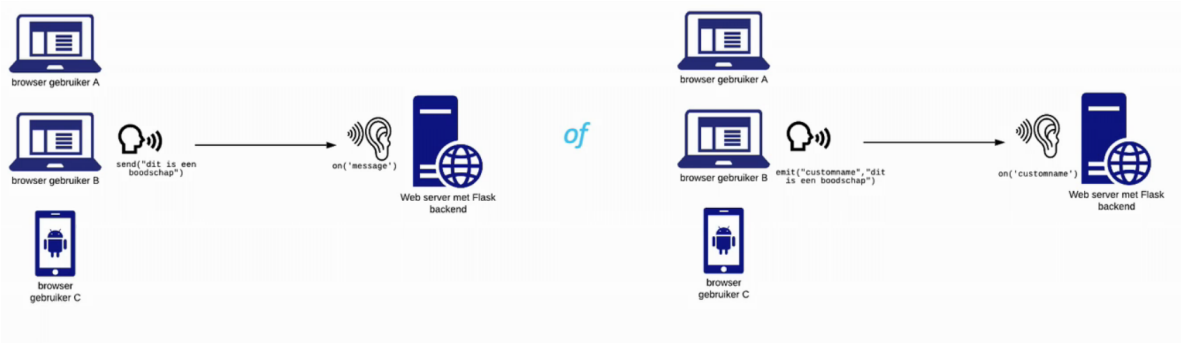
```
1 socketio.emit("naam", "dit is een boodschap")
```

Bij een send kunnen we enkel een string doorsturen.

→ eenvoudig, standaard

Bij een emit kunnen we ook een object doorsturen (payload), samen met een custom eventnaam

→ Meer variaties, ook objecten



Figuur 5: send vs emit

Het verschil kan je goed zien bij het opvangen.

Versturen we in de frontend (JS) de volgende code:

```
1 // voor send:
2 socketio.send("dit is een boodschap");
3
4 // voor emit:
5 socketio.emit("customname", "dit is een boodschap");
```

dan moeten we die in Python ontvangen met:

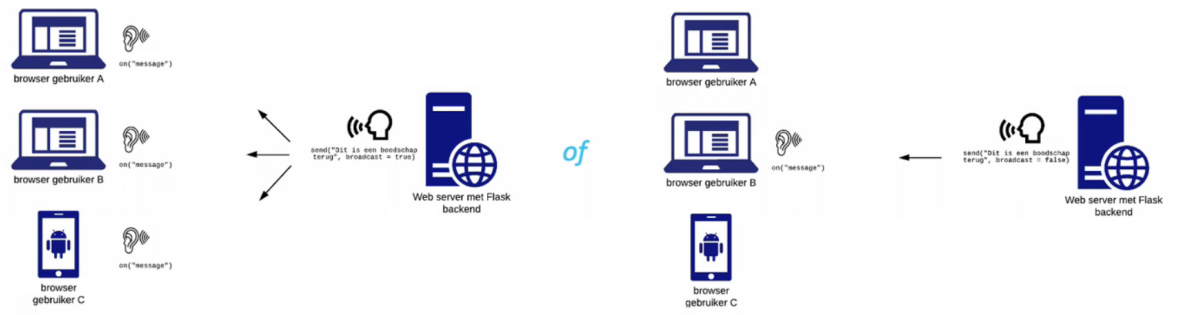
```
1 # 'message' vangt alles op dat via send binnenkomt
2 @socket.on('message')
3 def naam_functie(msg):
4     print(f"{msg}")
5
6 # 'customname' vangt enkel op indien de naam 'customname' is:
7 @socket.on('customname')
8 def naam_custom_functie(msg):
9     print(f"${msg}")
```

We kunnen nu boodschappen sturen van de client naar de server (client-server is altijd 1-op-1 connectie)

Indien we nu boodschappen sturen van de server naar de client kunnen we kiezen:

- We sturen een boodschap naar de ene client die net een boodschap stuurde
- We sturen een boodschap naar alle clients die momenteel verbonden zijn (=broadcast). Dit kan zowel met send als met emit.

Server → client kan zowel 1-op-1 als 1-op-veel connectie zijn.



Figuur 6: Broadcast aan of uit

Voorbeeld: we versturen nu eens een boodschap van server naar client(s) bij het ontvangen van een bericht:

```
@socket.on('message')
def naam_functie(msg):
    print(f"{msg}")
    socketio.send("Berichtje terug", broadcast=True)
    # je kan natuurlijk ook socketio.emit() gebruiken ipv send
    # om een extra payload door te sturen
```

Alle verbonden clients zullen nu "Berichtje terug" ontvangen, ze zullen dit kunnen opvangen met een socket.on("message")

9 API authenticatie & authorisatie met JSON web tokens

Hoe kunnen we communicatie tussen Frontend en API beveiligen?

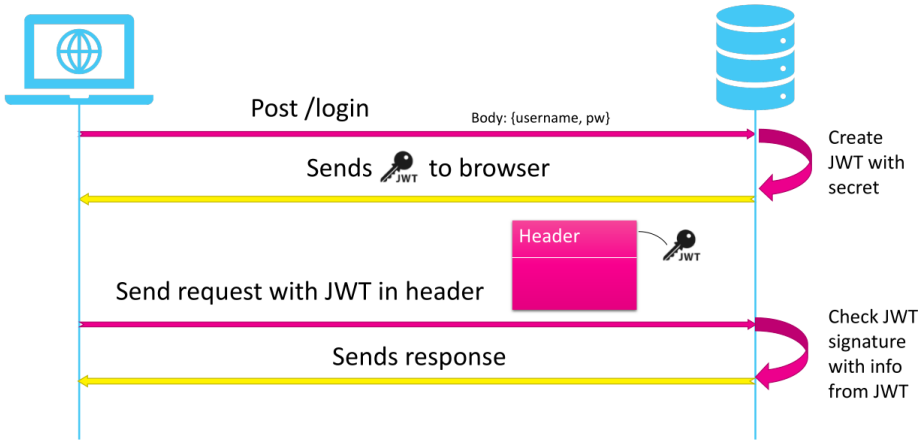
Authenticatie ⇒ Wie ben je?

Autorisatie ⇒ Waar heb je toegang tot?

9.1 Verschillende authenticatietypes

- Anonymous: iedereen krijgt toegang
- API keys: bij elk request een API key meegeven via Authorization Header
  - Bv: “api-key”: “5742d889-4de3-4cf7-803c-ea3bff667eb9”

- Basic Authentication: Bij elk request een Base64 encoded string meegeven van username en password via de Authorization Header
- Token Authentication : De client, na al dan niet verificatie van paswoord/username, krijgt een toegangsticket (een token), dat vervalt na x tijd. Bij elke volgende request eeft de client dit token mee via de Authorization Header
  - Veel gebruikt!



Figuur 7: Token authenticatie van dichter bij bekeken

9.2 Wat zijn JSON web tokens (JWT)?

- JSON Web Token (Industry standard RFC 7519)
  - Een open standaard (RFC7519) om op een compacte en veilige manier gegevens tussen twee partijen uit te wisselen.
  - Een JSON object
  - De informatie kan geverifieerd en vertrouwd worden door de digitale handtekening.
- Een toegangsticket dat credentials en claims bevat.
- Wordt door client verstuurd bij elke request
- Bestaat uit drie delen: een **header**, een **payload** en een **signature**

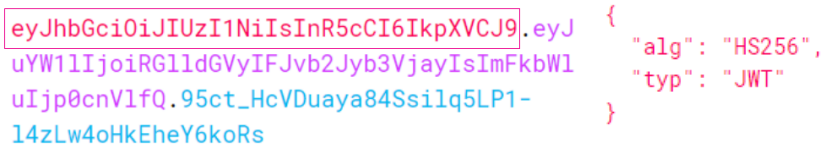
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
uYW1lIjoiRGlldGVyIFJvb2Jyb3VjayIsImFkbWl
uIjp0cnVlfQ.95ct_HcVDuaya84Ssilq5LP1-
l4zLw4oHkEheY6koRs
```

Figuur 8: JWT string: header, payload en signature

Met behulp van `http://www.jwt.io/`: illustreert het coderen, decoderen en verifiëren van een JWT ticket. De gehashte header en gehashte payload worden samengevoegd met een punt tussen, die string wordt dan op zijn beurt samengevoegd met een gekozen secret. De resulterende string wordt gehashed, met als resultaat de JWT signature.

9.2.1 JWT - Header

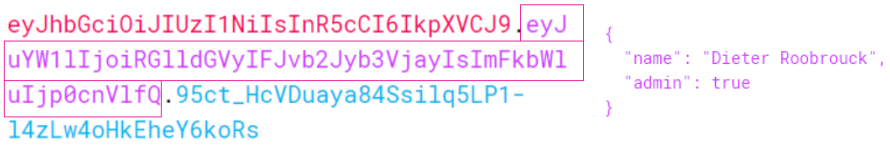
De Header bevat het gebruikte encryptie algoritme en type JWT.



Figuur 9: JWT Header

9.2.2 JWT - Payload

De payload bevat alle belangrijke gegevens over de gebruiker, inclusief autorisatie rechten zoals claims en permissies.

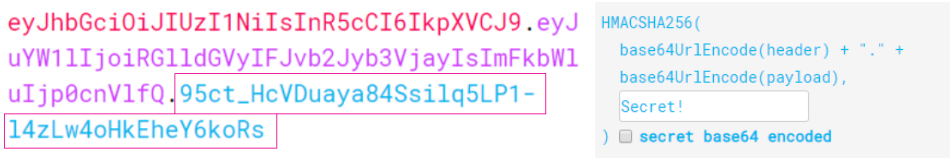


Figuur 10: JWT Payload

Deze zijn voor iedereen inzichtelijk dus let erop dat je hier geen gevoelige informatie inzet! Ook is een JWT token erop gemaakt om kort en compact te zijn dus denk aan de lengte van de namen van de claims die je aan het token toevoegt.

9.2.3 JWT - Signature

De handtekening wordt gebruikt om de echtheid van het token te garanderen. Ze bestaat uit een hash van header en payload aangevuld met een secret.



Figuur 11: JWT Signature

De secret is enkel gekend door de server en wordt nooit geshared!

9.3 API authentication using JSON Web Tokens

9.3.1 Stap 1: Backend (setup)

```
1 pip install flask-jwt-extended
```

```
from flask_jwt_extended import (
    JWTManager, jwt_requird, create_access_token, get_jwt_identity
)
app.config['JWT_SECRET_KEY'] = "Secret!"
jwt = JWTManager(app)
```

9.3.2 Stap 2: Frontend

We willen eerst de gebruiker identificeren als geldige gebruiker door middel van een POST-request naar de Login API-endpoint met in de payload een geldig e-mailadres en wachtwoord.

```
document.querySelector(".js-button").addEventListener("click", () => {
    const body = JSON.stringify({
        username: document.querySelector("#txtUsername").value,
        password: document.querySelector("#txtPassword").value
    });
    handleData('http://${lanIP}/api/v1/login', callbackShowToken,
    callbackShowErrorNoLogin, "POST", body);
})
```

9.3.3 Stap 3: Backend

1. We checken of de ontvangen gegevens kloppen
2. We maken een nieuwe JWT Token met behulp van **create_access_token**
3. We returnen het JWT Token terug naar de frontend

```
@app.route(endpoint + '/login', methods=['POST'])
def login():
    gegevens = DataRepository.json_or_formdata(request)

    username = gegevens['username']
    password = gegevens['password']

    #checken of de gegevens kloppen
    if username == "test" and password == "test":
        return jsonify(message="Missing username parameters"), 400
    else:
        return jsonify(message="Missing password parameters"), 400

    # normaal checken we in de usertabel of deze user bestaat
    # en of het wachtwoord bij die user klopt, maar dit laten we even achterwege

    #nieuwe JWT token maken
    expires = datetime.timedelta(seconds=10)
    access_token = create_access_token(identity=username, expires_delta=expires)

    #return JWT token
    return jsonify(access_token=access_token), 200
```

9.3.4 Stap 4: Frontend

Bij een volgende request richting de API moet de verkregen token meegegeven worden zodat de API de gebruiker kan verifiëren. Dit kan gedaan worden door de **Authorization header** in te vullen met de token die het login endpoint had teruggeven:

```
1 document.querySelector(".js-button-protected").addEventListener("click", () => {
2     document.querySelector(".js-result").innerHTML = "";
3     handleData('http://${lanIP}/api/v1/protected', callbackShowUser, callbackShowError
      , "GET", null, token); //token = Authorization header
4 });
```

* Daarvoor gaan we onze dataHandler moeten uitbreiden. Zie demo.

9.3.5 Stap 5: Backend

1. We beveiligen onze endpoint door een decorator toe te voegen: `@jwt_required`
2. We kunnen ook de identiteit opvragen van het token door middel van `get_jwt_identity`
3. We retourneren een response terug

```
1 @app.route(endpoint + "/protected", methods=["GET"])
2 @jwt_required
3 def protected():
4     current_user = get_jwt_identity()
5     return jsonify(logged_in_as=current_user), 200
```

Volledige uitleg: zie demo

10 Threading

Normaal wordt code van boven naar onder uitgevoerd. De volgende lijn van je programeercode kan slechts worden uitgevoerd als de bovenstaande lijn code is afgerond.

Met threading kunnen we ervoor zorgen dat code naast elkaar (asynchroon) kan worden uitgevoerd.

10.1 threading.Thread(target=callbackfunction)

```
1     def zeg_hello():
2     #zeg 6 maal hello maar wacht 2 seconden tussen elke boodschap
3     for i in range(6):
4         print(f"Hello student {i}")
5         time.sleep(2)
6
7 mijn_ander_proces = threading.Thread(target=zeg_hello)
8 mijn_ander_proces.start()
9
10 print('veeg het bord af')
11 print('start de pc op')
12 print('start de beamer op')
13 print('ga naar leho')
```

10.1.1 Output

```
Hello student 0
veeg het bord af
start de pc op
start de beamer op
ga naar leho
Hello student 1
Hello student 2
Hello student 3
Hello student 4
Hello student 5
```

10.2 threading.Timer(seconds, callbackfunction)

threading.Timer() wacht eerst 3 seconden, en roept dan de callbackfunctie op:

```
import time
import threading
def zeg_hello():
    #zeg 6 maal hello maar wacht 2 seconden tussen elke boodschap
    for i in range(6):
        print(f"Hello student {i}")
        time.sleep(2)

mijn_ander_proces = threading.Timer(3,zeg_hello)
mijn_ander_proces.start()
print('veeg het bord af')
time.sleep(1)
print('start de pc op')
time.sleep(1)
print('start de beamer op')
time.sleep(1)
print('ga naar leho')
time.sleep(1)
```

10.2.1 Output

```
veeg het bord af
start de pc op
start de beamer op
Hello student 0
ga naar leho
Hello student 1
Hello student 2
Hello student 3
Hello student 4
Hello student 5
```