

Databanken en Webtechnologie: Project 21-22: Verslag

Seppe Goossens

November 17, 2021

Contents

1	Database	3
1.1	ER Diagram	3
1.2	Interactie met de database	4
2	RESTful API	5
2.1	Initializing	5
2.2	Communication	6
2.3	Classes	7
2.4	Functions	8
2.5	Routing	8
2.6	Start App	10
3	Website	11
3.1	Log In	11
3.2	Homepage	12
3.3	Create Class	13
3.4	Register Presence	13
3.5	See Presence	13
3.6	Assign Course	16
3.7	Error Pages	17

1 Database

1.1 ER Diagram

Het ER diagram en dus de database zijn zo opgesteld dat elke gebruiker (administrator, teacher, student) een rol nummer krijgt. Deze rol nummers worden bijgehouden in een aparte tabel. De rol tabel wordt gelinkt aan een account tabel. Deze tabel bevat de email, naam, password en de rol van de gebruiker. De account tabel is dus voor elke gebruiker hetzelfde doordat de rol een aparte tabel heeft gekregen. Naast dit hebben we ook een active user tabel. Deze tabel houdt bij welke gebruiker er op het huidige moment is ingelogd. Bij het inloggen wordt er ook een unieke session key toegekend die zal gebruikt worden om de data van de huidige gebruiker op te vragen. De account tabel wordt gelinkt aan de cursus tabel. De cursus tabel bevat de naam van het vak en ook welke gebruikers dit vak beheren of volgen door de accountID in te stellen als de foreign key. Elk vak kan een les bevatten, hierdoor wordt de class tabel gelinkt aan de cursus tabel. De classID en courseID zijn ingesteld als primaire sleutels omdat elk vak meerdere unieke lessen kan bevatten. Bovendien bevat de class tabel ook nog de naam van de klas en de unieke code dat de studenten kunnen gebruiken om hun aanwezigheid kenbaar te maken. De aanwezigheid wordt apart bijgehouden via de accountID als foreign key zodat de leerkracht exact kan zien welke studenten wel en niet aanwezig waren tijdens de les. Via dit design is het mogelijk om elke gebruiker apart toe te kennen aan een specifiek vak en een bijhorende les. Naast dit is het ook mogelijk om voor elke les de aanwezige studenten te bekijken.

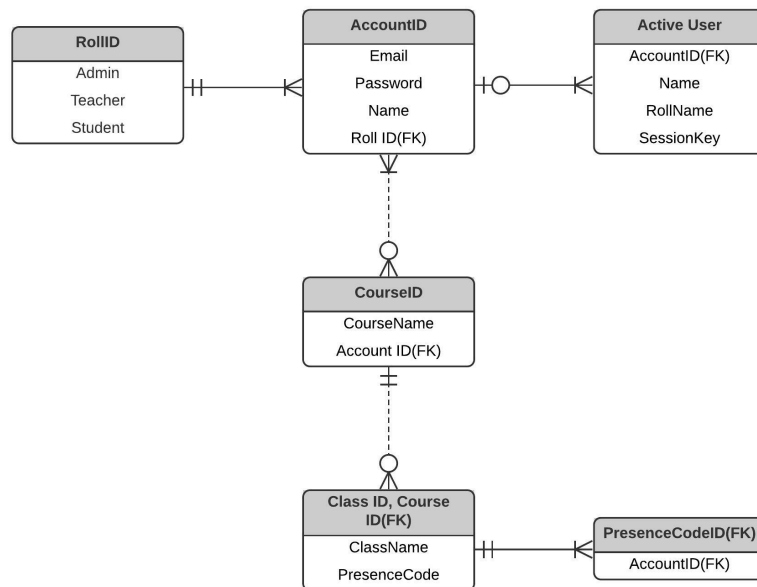


Figure 1: ER Diagram

1.2 Interactie met de database

Voor het maken van de database heb ik gebruik gemaakt van DB Browser for SQLite. Ik heb de database gemaakt volgens het ER diagram in figuur 1. Via queries in SQLite kan er data opgevraagd en toegevoegd worden aan de database. Deze queries zijn dus essentieel bij de interactie tussen de API en de database. Hieronder wordt er uitgelegd hoe de queries werken bij een aantal voorbeelden die gebruikt worden in het project.

Data opvragen uit de database:

Data opvragen uit de database gebeurt via het **SELECT** statement. Om te kiezen uit welke tabel we de data gaan halen maken we gebruik van de **FROM** statement. Om naar een specifieke record in de database te zoeken kunnen we gebruik maken van het **WHERE** statement. Hier is een voorbeeld van het opvragen van de accountID voor een gegeven email:

```
SELECT AccountID FROM Account WHERE Email = 'email@email.com'
```

Deze query retourneert dan alle accountID's die voldoen aan de **WHERE** statement. In dit geval zal er maar 1 record geretourneerd worden omdat de accountID en email uniek zijn aan 1 gebruiker.

Data toevoegen aan de database:

Er moet ook data toegevoegd worden aan de database. Dit kan gebeuren via het **INSERT INTO** statement. Opnieuw moet er gespecificeerd worden in welke tabel en welke kolommen de data toegevoegd moet worden. Hieronder een voorbeeld waarbij de gebruiker wordt toegevoegd aan de aanwezigheids tabel:

```
INSERT INTO "main"."Presence" ("CodeID", "AccountID") VALUES ('AAAAA',2);
```

Waarbij **VALUES** staat voor de waarde van de variabelen **"CodeID"** en **"AccountID"** die moeten worden ingevuld.

Data verwijderen uit de database:

Er kan ook data verwijderd worden uit de database. Dit gebeurt via het **DELETE** statement. Bij een **DELETE** hoort ook altijd een **WHERE** statement om te specificeren welke rij er verwijderd moet worden uit de gekozen tabel. Hieronder een voorbeeld van het verwijderen van een gebruiker uit de active user tabel:

```
DELETE FROM ActiveUser WHERE Email = 'email@email.com'
```

2 RESTful API

De code voor de API kan terug gevonden worden in `main.py`. De code is opgedeeld in de volgende onderverdeling:

- Initializing
- Communication
- Classes
- Functions
- Routing
- Start app

Hieronder overlopen we elk van deze punten en wordt er uitleg gegeven over hoe de code is opgebouwd.

2.1 Initializing

In dit deel van de code worden de libraries ingeladen, de connectie gemaakt met de database en wordt de app en de API gemaakt. De gebruikte libraries zijn de volgende:

```
import sqlite3
import json
import flask
from flask_restful import Resource, Api
from flask_api import status
from flask import (Flask, render_template, Markup, request, make_response, jsonify)
import requests
import random
import string
```

De connectie met de database kan als volgt gemaakt worden:

```
conn = sqlite3.connect("\\Project2122.db")
```

Vervolgens definiëren de url van de API:

```
api_address = "http://127.0.0.1:8080"
```

En maken we de app en de API aan:

```
# Create our app
app = flask.Flask(__name__)
# Create an API for our app
api = Api(app)
```

2.2 Communication

In dit deel worden alle functies aangemaakt die de database moeten contacteren en hierbij data opvragen of toevoegen aan de database.

Opvragen van data:

Voorbeeld van zo een functie waarbij de accountID voor een gegeven username wordt opgevraagd:

```
# Get AccountID for given username
def get_accountID(username):
    url = api_address + "/show/accountID/" + username
    data = requests.get(url)
    return data.json()
```

Eerst wordt de URL opgeemaakt door het reeds gedefinieerde api adres samen te voegen met een tweede deel. Dit tweede deel van de URL, in dit voorbeeld `"/show/accountID/" + username`, bestaat uit een zelf gekozen stukje URL. Hierachter plaatsen we de gevraagde variabelen, in dit geval de username. Vervolgens wordt deze url meegegeven aan de `requests.get()` functie. Deze functie vraagt aan de API om de database te contacteren en te retourneren wat er op de meegegeven URL als resultaat getoond wordt. Vervolgens zetten we de data om in een json en retourneren we de data.

Toevoegen van data:

Voor het toevoegen wordt het zelfde principe gevolgd. Alleen moet er geen data geretourneerd worden. Hier een voorbeeld waarbij de huidige gebruiker toegevoegd wordt aan de database:

```
# Add Active user to the database -> seperate Active User Table
def add_session_db(accountID, username, name, roll, token):
    url = '{}/add/activeuser/{},{},{},{},{}'
    .format(api_address, accountID, username, name, roll, token)
    requests.get(url)
```

Voor meerdere variabelen wordt er gebruik gemaakt van de `.format()` functie om de hele URL om te zetten in 1 string. Idealiter zou de functie een bool moeten retourneren als de data succesvol toegevoegd is aan de database.

Checken van data:

Om te kijken of een bepaalde record al aanwezig is in de database zijn er ook check functies. Deze nemen een variabelen en kijken of die al dan niet aanwezig is in de database. Dit kan bijvoorbeeld handig zijn bij het aanmaken van een course en te kijken of deze al dan niet al bestaat en zo dus moet aangemaakt worden of niet. Deze functie retourneert dan een bool afhankelijk van het resultaat. Voorbeeld voor het controleren of een course al aanwezig is in de database of niet:

```
# Check if Course exists in the database
def check_course_indb(course_name):
```

```

url = api_address + "/show/course/" + course_name
data = requests.get(url)
data_json = data.json()
#Json is empty -> Doesn't exist
if data_json['Course Name'] == False:
    return False
else:
    return True

```

2.3 Classes

In de klassen sectie worden de klassen aangemaakt die queries bevatten. Er zijn 3 klassen gebruikt.

Show Class:

We beginnen met de show klasse. Deze klasse komt overeen met de **SELECT** query. Er is een query aangemaakt die aan de functionaliteit van de klas voldoet. Deze query en de benodigde variabelen worden dan meegegeven aan de `conn.execute()` functie. De functie retourneert de overeenstemmende data. Vervolgens itereren we over deze data en wordt deze omgevormd tot een lijst. Tot slot wordt er een json geretourneerd die wordt aangevuld met de verschillende onderdelen van de lijst. Hieronder een voorbeeld van een show klasse die de accountID voor een gegeven username retourneert.

```

# Show accountID for given username
class ShowAccountID(Resource):
    def get(self, username):
        q = "SELECT AccountID FROM Account WHERE Email = ?"
        data = conn.execute(q, (username, )).fetchall()
        accountID = []
        for item in data:
            for row in item:
                accountID.append(row)
        return {"Account ID": accountID[0]}

```

Sommige show klassen heb een iets andere layout afhankelijk van de functionaliteit van de klasse. Sommige klassen retourneren een false statement als de geretourneerde data van de query leeg is. Deze klassen worden dan bijvoorbeeld gebruikt bij een check functie.

Add Class:

De add klassen, in tegenstelling tot de show klassen, voegen data toe aan de database. Deze komen dus overeen met het **INSERT INTO** statement. Hierbij retourneren ze dus ook niets. Idealiter zou je hier een bool kunnen retourneren die zegt of de data al dan niet succesvol is toegevoegd aan de database, zoals bij de add functies in het vorige deel. De add klassen volgen hetzelfde principe alleen is er `conn.commit()` toegevoegd zodat de API weet dat hij effectief de data moet toevoegen aan de database. Hieronder een voorbeeld van het toevoegen van de actieve gebruiker aan de aanwezigheids tabel:

```
# Add active user to the presence table
class AddToPresence(Resource):
    def get(self, presence_code, account_ID):
        q = """INSERT INTO "main"."Presence" ("CodeID", "AccountID") VALUES (?, ?);"""
        conn.execute(q, (presence_code, account_ID))
        conn.commit()
```

De delete klasse werkt volgens hetzelfde principe. Onderdaan vinden we alle `api.add_resource()`. Hiermee roepen we de klasse op met zelfgemaakte URL. In deze URL kunnen we variabelen meegeven door gebruik te maken van `<type:var>`. Hieronder vind je de `api.add_resource()` van de bovenstaande klassen.

```
api.add_resource>ShowAccountID, "/show/accountID/<string:username>"
api.add_resource(AddToPresence, "/add/presence/<string:presence_code>,<string:account_ID>")
```

Merk op dat de naam van de klassen en de naam en type variabelen overeen moeten komen. Voor het gebruik van de klassen en de functies in het vorige deel, zie de API documentatie

2.4 Functions

De functie sectie bevat zelf gemaakt functies die gebruikt worden om bijvoorbeeld een class code of session key aan te maken als ook het converteren van een json naar een list die de juist indexen bevat.

2.5 Routing

De routing sectie bevat alle `@app.route()` die nodig zijn voor het laden en doorverwijzen van de html pagina's. Deze app routes vormen in feite de connectie tussen je website en de API. Er zal een website geretourneerd worden via een render template. In deze render template kan je data meegeven die je bijvoorbeeld uit je database hebt gehaald. Zie de code in de `"""ROUTING"""` sectie in `main.py`. De uitleg die hier wordt gegeven is ter verduidelijking.

Log in:

Voor het inloggen moet je de volgende URL invullen: "http://127.0.0.1:8080/login". Hierbij zal er een render template geretourneerd worden die de "login.html" pagina bevat. De `render_template("page.html")` verwijst je door naar de meegegeven pagina. Elke app route zal dus ook eindigen met het retourneren van een render template met de gewenste pagina. Voorbeeld van de login routing:

```
#Show log in page
@app.route("/login")
def login_page():
    return render_template("login.html")
```

Log in attempt:

Om effectief in te loggen word je doorverwezen naar de login attempt pagina. Hierbij wordt er gekeken naar wat de gebruiker ingevuld heeft als gebruikersnaam en wachtwoord en wordt er gecontroleerd of deze bestaan in de database

en of deze gebruikersnaam en wachtwoord wel overeenkomen. Als dit klopt dan wordt er een session key aangemaakt die uniek is aan de gebruiker. Deze session key wordt zowel bijgehouden in de database als in een cookie via de volgende code: `response.set_cookie("auth_token", token)`. Vervolgens word je doorverwezen naar de homepage via de render template. Als je gebruikersnaam en wachtwoord niet overeenkomen dan word je doorverwezen naar een error pagina.

Homepage:

Als je op de homepage bent aangekomen wordt er gecontroleerd of je nog steeds ingelogd bent door de cookie te raadplegen: `token = request.cookies.get("auth_token")`. Deze token wordt dan vergeleken met die in de database. Als deze hetzelfde zijn is de gebruiker nog steeds ingelogd. Zo niet wordt er een error pagina getoond. Dit gebeurt elke keer als je naar een nieuwe pagina gaat. Bij de homepage wordt er gewoon een welkom scherm getoond met de naam van de gebruiker, dit ook via de render template

Create Class:

Voor het maken van een klasse kan je doorverwezen worden naar de create class pagina. Deze pagina is enkel toegankelijk voor een admin of een teacher. Bij het aanmaken van de klas wordt er ook een presence code aangemaakt die de studenten kunnen invoeren voor het registreren van hun aanwezigheid. Er wordt ook gecontroleerd of de code al dan niet bestaat. Vervolgens wordt de naam van de course, class en de presence code toegevoegd aan de database en wordt er een render template geretourneerd die de presence code toont. Als de klas al bestaat zal dit ook worden getoond via een render template.

Register Presence:

Het registreren van de aanwezigheid verloopt ongeveer hetzelfde als bij het aanmaken van de klasse. Er wordt gecontroleerd of de huidige gebruiker al dan niet is ingeschreven in de course die correspondeert aan de ingegeven presence code. Als dit klopt dan wordt de accountID en de presence code opgeslagen in de daar voor voorziene tabel in de database. Als er iets fout loopt, zal ik dit opnieuw weergegeven worden via een render template met een foutmelding.

See Presence:

Voor het bekijken van de aanwezigheden moet er een klas naam ingegeven worden. Er wordt eerst gecontroleerd of deze klas al dan niet aanwezig is in de database. Vervolgens worden de studenten die aanwezig waren voor die specifieke les uit de database gehaald als ook de studenten die niet aanwezig waren. Deze worden dan via de render template geretourneerd naar de webpagina. Op de webpagina verschijnt er dan een tabel met de aanwezige en niet aanwezige studenten. Opnieuw wordt er een foutmelding gegeven als er iets fout loopt.

Assign Course:

Deze pagina is enkel beschikbaar voor de admin. Er wordt gecontroleerd of de ingevulde username en course name effectief bestaan in de database. Als dit het geval is dan wordt de ingevulde username opgeslagen bij de meegegeven course. Ook hier wordt er een foutmelding gegeven als er iets fout loopt.

2.6 Start App

De app kan gestart worden via de volgende code:

```
if __name__ == '__main__':  
    app.run(port=8080)
```

3 Website

In de volgende secties gaan we de functionaliteit van de website bespreken als ook hoe de code erachter is opgebouwd.

3.1 Log In

De log in pagina bevat 2 velden. Een voor de username en een voor het paswoord. Deze moeten ingevuld worden vooraleer je de login knop kan indrukken.

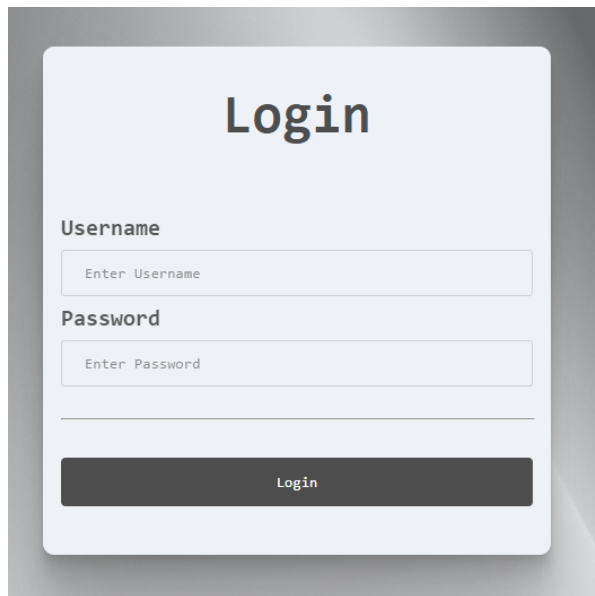
The image shows a login form on a light blue background. At the top, the word 'Login' is written in a large, bold, black font. Below it, there are two input fields. The first is labeled 'Username' in bold black text, and the input field has a placeholder text 'Enter Username'. The second is labeled 'Password' in bold black text, and the input field has a placeholder text 'Enter Password'. Below these fields is a horizontal line, and then a dark grey button with the text 'Login' in white.

Figure 2: Login

De HTML code voor het ingeven van de username en paswoord als ook voor het drukken op de knop ziet er als volgt uit:

```
<form method="post" action="/login_attempt">
  <div class="container">
    <label for="username"><b>Username</b></label>
    <input type="text" id = "username"
      placeholder="Enter Username" name="username" required>
    <label for="password"><b>Password</b></label>
    <input type="password" id = "password"
      placeholder="Enter Password" name="password" required>
    <hr>
    <button type="submit" value="Log In">Login</button>
  </div>
</form>
```

Voor het verkrijgen van de data uit de velden in de back end kan je `request.form.get()` gebruiken. Voorbeeld voor het verkrijgen van de username en paswoord uit de velden:

```
username = request.form.get('username')
password = request.form.get('password')
```

Om te weten wanneer de knop wordt ingedrukt kan je ook van `request.form.get()` gebruik maken. Voorbeeld:

```
request.form.get('create_class_button') == 'pressed'
```

3.2 Homepage

Op de homepagina kan je een navigatiebalk terug vinden met alle mogelijke functionaliteiten. Dit laat toe om makkelijk naar de ander pagina's te navigeren. De HTML code voor het maken van een navigatie bar ziet er als volgt uit:

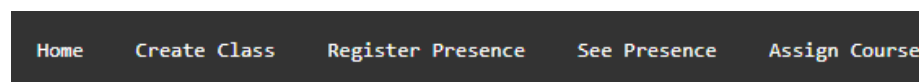


Figure 3: Navigatiebalk

```
<nav>
  <li><a href="{{url_for('homepage')}}">Home</a></li>
  <li><a href="{{url_for('create_class')}}">Create Class</a></li>
  <li><a href="{{url_for('register_presence')}}">Register Presence</a></li>
  <li><a href="{{url_for('see_presence')}}">See Presence</a></li>
  <li><a href="{{url_for('assign_course')}}">Assign Course</a></li>
</nav>
```

Elk van de onderdelen bevat een shortcut naar de nodige html pagina. Deze code komt ook voor in elk van de andere html pagina's zodat je altijd naar de gewenste pagina kan gaan. Naast de navigatiebalk is er ook een welkom scherm met de naam van de gebruiker:



Figure 4: Welcome Homepage

In de back end wordt de naam van de gebruiker opgevraagd d.m.v. de cookie die de session key retourneert waarna deze zal gebruikt worden om de naam van de gebruiker op te vragen in de database. De naam wordt dan via de render template geretourneerd.

```
token = request.cookies.get("auth_token")
active_user_data = get_active_user_data(token)
if token:
    name = active_user_data['Name']
    return render_template("homepage.html", contents = "Welcome " + name)
```

De contents wordt dan in een header in de HTML code als volgt ingevuld:

```
<h1>{{contents}}</h1>
```

3.3 Create Class

De create class tab laat de teacher toe om een klas aan te maken. Eens dat de klas succesvol is aangemaakt zal er een unieke presence code geretourneerd worden. De klas in kwestie behoort tot een cursus en dus deze zal ook ingevuld moeten worden. Hiervoor zijn dus 2 velden en een knop voor voorzien:

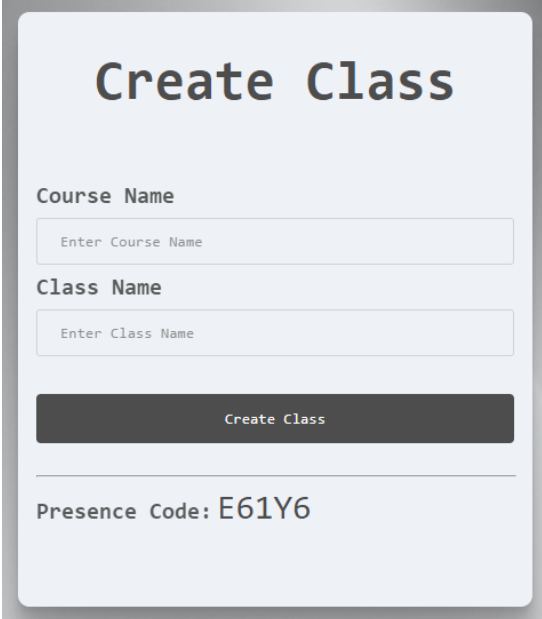
The image shows a web form titled "Create Class". It has a light blue background with rounded corners. At the top, the title "Create Class" is in a large, bold, dark font. Below the title, there are two input fields. The first is labeled "Course Name" and has a placeholder text "Enter Course Name". The second is labeled "Class Name" and has a placeholder text "Enter Class Name". Below these fields is a dark grey button with the text "Create Class" in white. At the bottom of the form, there is a line of text that reads "Presence Code: E61Y6".

Figure 5: Create Class Form

De HTML code is gelijkaardig aan die bij de login pagina. De geretourneerde presence code komt tevoorschijn onder de knop. Opnieuw is de code van de back end gelijkaardig aan die van de login en van de homepagina. De gemaakte code zal meegegeven worden aan de render template en deze zal dus ingevuld worden in de daarvoor voorziene `{{contents}}` header.

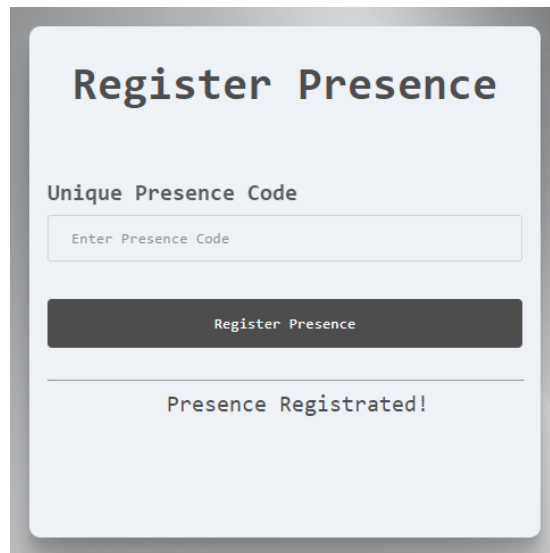
3.4 Register Presence

De register presence tab laat de student toe om zich kenbaar te maken bij een specifieke les. De student moet de verkregen code bij het aanmaken van de les invullen in de daarvoor voorziene veld. Als de registratie succesvol is verlopen dan wordt er een bericht geretourneerd.

Opnieuw is de back end en front end code gelijkaardig aan de code in de vorige delen.

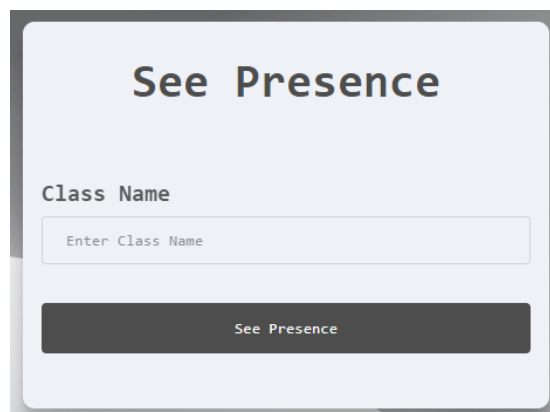
3.5 See Presence

Om te kijken welke studenten aanwezig waren en welke niet voor een specifieke les is er een see presence tab voorzien. Deze bevat 2 delen. Het eerste deel is bedoeld om de naam van de klas waar je de aanwezigheden wilt van controleren in te vullen. Opnieuw is de code hiervoor gelijkaardig aan de vorige delen.



The image shows a web form titled "Register Presence". It has a light blue background with rounded corners. At the top, the title "Register Presence" is in a large, bold, dark font. Below the title, the label "Unique Presence Code" is in a smaller, bold, dark font. Underneath this label is a text input field with the placeholder text "Enter Presence Code". Below the input field is a dark grey button with the text "Register Presence" in white. At the bottom of the form, there is a horizontal line, and below it, the text "Presence Registrated!" is displayed in a dark font.

Figure 6: Register Presence Form



The image shows a web form titled "See Presence". It has a light blue background with rounded corners. At the top, the title "See Presence" is in a large, bold, dark font. Below the title, the label "Class Name" is in a smaller, bold, dark font. Underneath this label is a text input field with the placeholder text "Enter Class Name". Below the input field is a dark grey button with the text "See Presence" in white.

Figure 7: See Presence Form

Het tweede deel dient ervoor om effectief te kijken welke studenten er wel en niet aanwezig waren tijdens de opgegeven les. Bij de aanwezige student wordt de naam van de les, de naam van de student en hun email getoond. Bij de niet aanwezige studenten wordt er enkel de naam en de email getoond. In de back

Presence Table		
Present Students:		
Class Name	Name	Email
Basiswiskunde Les 1	Seppe Goossens	seppe.goossens@vub.be
Not Present Students:		
Name	Email	
Student Een	student1@vub.be	

Figure 8: See Presence Table

end wordt de benodigde data uit de database opgevraagd en meegegeven via een render template:

```
return render_template("see_presence.html",
    present_headings=present_headings,
    present_data=present_data,
    non_present_headings=non_present_headings,
    non_present_data=non_present_data)
```

De meegegeven data zijn 2D lijsten waar we dus over kunnen itereren. In de HTML code kunnen we itereren over de meegegeven data d.m.v. curly brackets te gebruiken. Hierin kunnen we dan python code invullen. Dit ziet er als volgt uit:

```
<div class="container">
  <table>
    <tr>
      {% for header in present_headings %}
      <th>{{ header }}</th>
      {% endfor %}
    </tr>
    {% for row in present_data %}
    <tr>
```

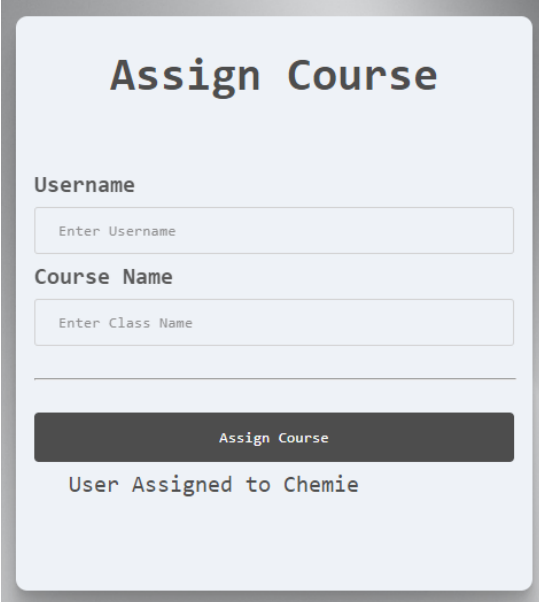
```

        {% for cell in row %}
        <td>{{ cell }}</td>
        {% endfor %}
    </tr>
    {% endfor %}
</table>
</div>

```

3.6 Assign Course

Deze tab is enkel toegankelijk voor de admin. Hiermee kan de admin studenten en leerkrachten toekennen aan een ingegeven cursus. De gebruikersnaam en de naam van de cursus moeten worden ingevuld. Opnieuw is de code gelijkaardig aan de vorige delen.



Assign Course

Username

Enter Username

Course Name

Enter Class Name

Assign Course

User Assigned to Chemie

Figure 9: Assign Course Form

3.7 Error Pages

Er zijn ook 2 error pagina's voorzien. De `wrong_roll_error_page.html` wordt geretourneerd als de huidige gebruiker geen toegang heeft tot de pagina. Dit kan bijvoorbeeld zijn als de student de "assign course" pagina wilt bereiken terwijl de student geen toegang heeft tot deze pagina. De error pagina ziet er als volgt uit:



Not Authorized to Assign a Course

Figure 10: Wrong Roll Error Page

Anderzijds is er ook de `login_error_page.html` pagina waarbij de gebruiker niet meer ingelogd is en dus geen toegang meer heeft tot de website. De gebruiker krijgt dan terug de mogelijkheid om in te loggen door in de navigatiebalk op "log in" te drukken. Deze pagina wordt ook getoond als de gebruiker de foute log in gegevens heeft ingevuld.