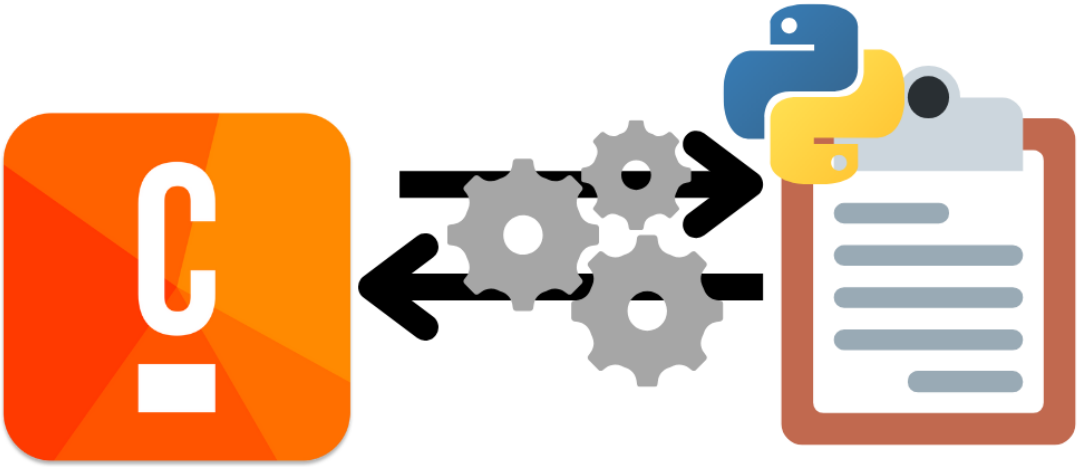
TBZ – PE22b – M254_LB3

Python Camunda

A python task manager that has integrated Camunda processes



Contents

Legend

The legend serves the purpose of clarifying the authorship, thereby making it easier for the reader to understand who is writing. This added clarity helps ensure that the reader can follow the narrative or information being presented more effectively.

Text	Author
This is my text, written by Rayan Lee Bopp	Rayan Lee Bopp
This is my text, written by Florian Merlin Fröbel	Florian Merlin Fröbel
This is my text, written by Nick Ramon Huber	Nick Ramon Huber
This is my text, written by Lars Schulthess	Lars Schulthess

Utilization of individualized text

This is general text; it will be written from the perspective of "We." All general text should use "We" to maintain a collective voice. If an individual wishes to provide a personal interjection, please use your assigned colour style. The name of this style is AUTHOR [YOUR INITIALS].

This is my text, written by Rayan Lee Bopp

This is my text, written by Florian Merlin Fröbel

This is my text, written by Nick Ramon Huber

This is my text, written by Lars Schulthess

Introduction

This document is the final assignment given to us by Mr. Kälin, serving as both a research and project report. Within this document, you will find a comprehensive overview of our research, detailed accounts of our achievements, the methods we employed to complete the project, and the key findings we discovered. Additionally, we will discuss the challenges we faced and how we overcame them, providing a thorough insight into the entire project process.

Scope statement

As this document serves dual purposes, functioning as both a research paper and a project report, you will find detailed information on the technical implementation of our project, as well as insights into our learning process to deepen our understanding of the theoretical concepts involved.

We have set our focus on two key aspects: first, the integration of Camunda forms to enhance the user experience and improve business process performance within transactions. This means we aim not only to improve functionality but also to optimize the customer experience.

We aimed to achieved this by using a preexisting project, as a blank, on which we then implemented the individual business processes. Mainly the account creation BPMN.

Planning

Agenda

Shown below this is the Agenda I've created for our team; it strictly adheres to the subjects Mr. Kälin.

Datum	Woche	Im Unterricht behandelte Inhalte	Gruppe Agenda	Termine
14.05.2024	1	Begriff Geschäftzprozesse, grafische Gestaltung GP	Dokumentation in GitHub	keine
21.05.2024	2	BPMN-Symbole kennenlernen, Prozesse lesen/Interpretieren	Dokumentation in GitHub, Vorbereitung auf LB1 und Vorbereitung auf LB2 (Erstes Meeting)	Meeting (Teams, Zimmer oder Discord)
28.05.2024	3	Bestehende Geschäftsprozesse erklären, Vorbereitung LB1	Dokumentation in GitHub, Vorbereitung auf LB1 und Vorbereitung auf LB2	Kick-Off Meeting
04.06.2024	4	LB1 - Start LB2	Start and LB2 laut meetings von Woche 2 und 3	Weekly Meeting (Freitag)
11.06.2024	5	Sporttag	Start Vorbereitungen für LB3 (Meeting) und Abschluss LB2	Weekly Meeting (Freitag)
18.06.2024	6	Start LB3	Start LB3 laut vorbereitungen von Meetings	Weekly Meeting (Freitag)
25.06.2024	7	Arbeit an LB3	Arbeit LB3 laut vorbereitungen von Meetings	Weekly Meeting (Freitag)
02.07.2024	8	Abschluss LB3	Arbeit LB3 laut vorbereitungen von Meetings (Dokumentations abschluss)	Weekly Meeting (Freitag)
09.07.2024	9	Präsentation LB3	Präsentations abgabe	Weekly Meeting (Freitag)

Meetings

As seen in the Agenda above we have a strict plan it depicts when each meeting should take place, for example take the Kick-Off Meeting which was strategically placed the week before so that we could immediately start into the project while also having enough preparation to easily tackle the LB2.

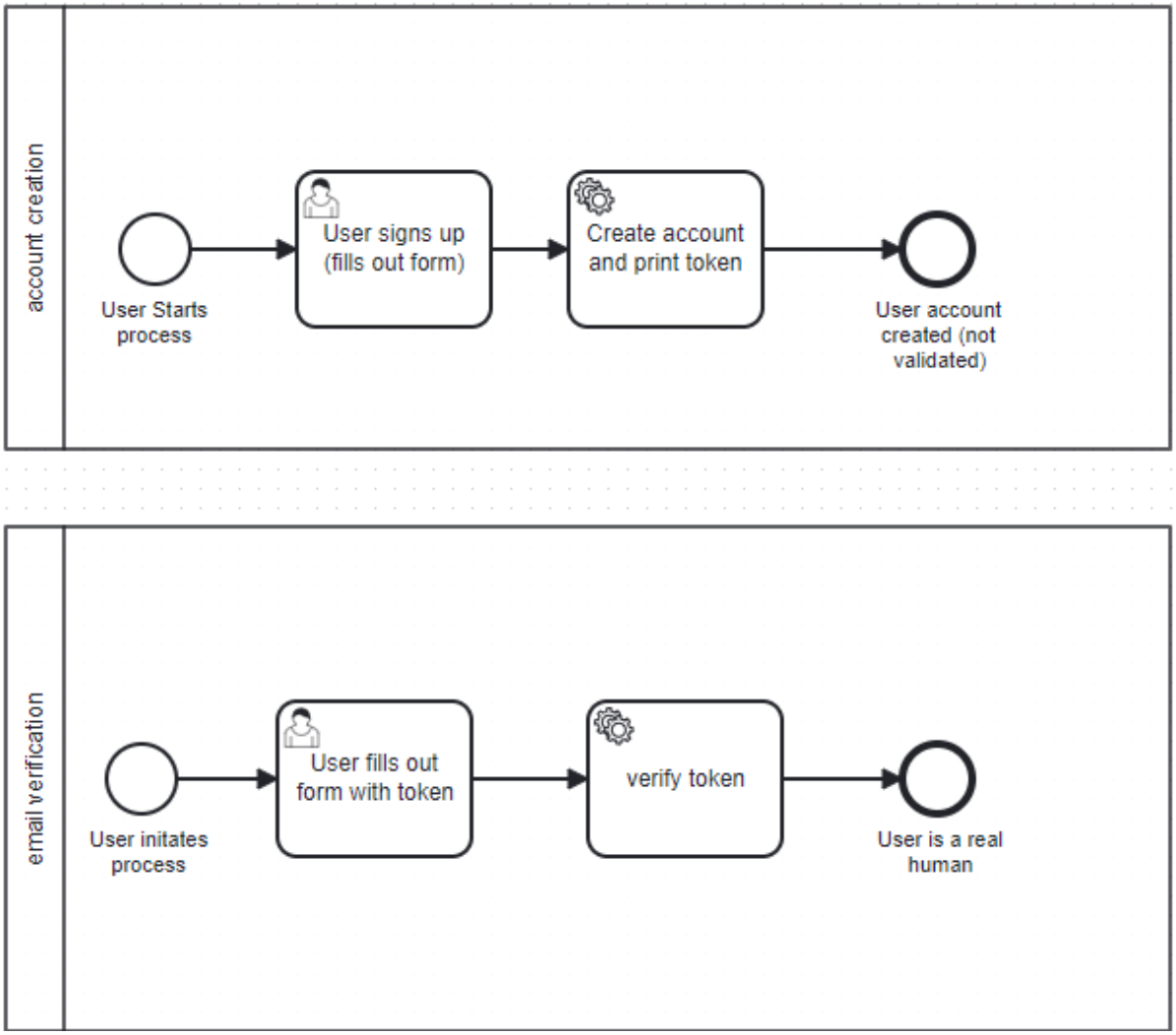
Weekly User Stories

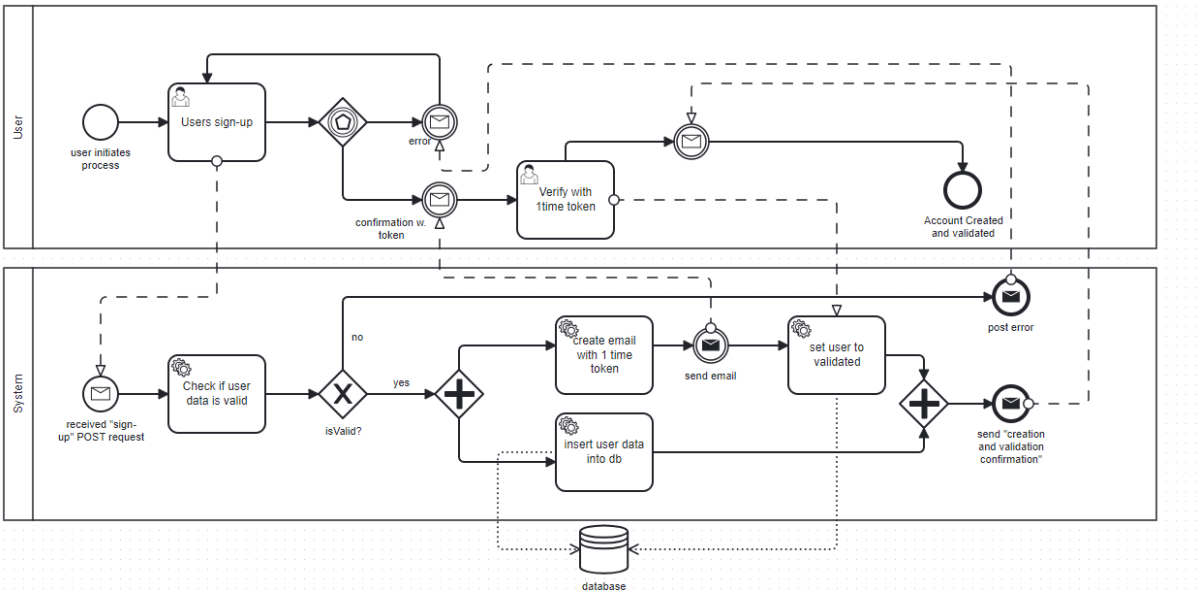
I have devised a strategy which should prevent any procrastionation from happening. You can see them on our GitHub. ---> Weekly User Story - Template

Graphical representation

For the BPMN we kept it simple, as we later found out that if you overcomplicate the BPMN diagrams as well as the HTML forms the Camunda Engine can't keep up as well as it used to.

So we have decided to use two BPMN diagrams, one should be capable of being as an extension of our web application and the other should be able to just imitate our process.





Realization

Realization – FMF

I had the task to plan with Rayan and to work on task creation/task deletion.

For this the initial steps where to create a python-file, linking this into JavaScript, and then finally making this js accessible from the html index file.

```
M254/
|
├── Files/
|   ├── LB3-Code/
|   |   ├── .vscode/
|   |   ├── database/
|   |   |   ├── LB3_DB_Layout.mwb
|   |   |   ├── LB3_DB_Layout.sql
|   |   ├── modules/
|   |   |   ├── __pycache__/
|   |   |   ├── auth.py
|   |   |   ├── task_creation.py
|   |   |   ├── task_deletion.py
|   |   |   ├── verify.py
|   |   ├── static/
|   |   |   ├── img/
|   |   |   ├── js/
|   |   |   |   ├── darkmode.js
|   |   |   |   ├── register.js
|   |   |   |   ├── task_creation.js
|   |   |   |   ├── task_deletion.js
|   |   |   |   ├── verify.js
|   |   ├── templates/
|   |   |   ├── index.html
|   |   |   ├── verify.html
|   |   ├── app.py
|   |   ├── README.md
|   |   ├── requirements.txt
|   |   ├── LB3-Doc/
|   |   ├── LB3-Files/
|   ├── README.md
|   ├── .gitignore
|   ├── LICENSE
```

…planed and implemented by Rayan.

…ed multiple layouts and solutions but …o two different Files (task_creation.py & …d a better overview.

…g as uniform as possible, i also created two …sk_deletion.py) and embedded them into the …

…e done, here are the most important functions …

task_creation.py

- **Purpose**: Establishes a connection to the MySQL database.
- **Functionality**:
    - Tries to connect to the database using the provided host, database name, user, password, and port.
    - If the connection is successful, it returns the connection object.
    - If the connection fails, it catches the error, prints an error message, and returns None.

create_task()

- **Purpose**: Handles the creation of a new task by inserting the task details into the database.
- **Endpoint**: `/create_task`
- **Method**: POST
- **Functionality**:
    - Retrieves task details (name, description, start_date, assign_user, created_by, priority_id) from the request form.
    - Validates that all required fields are provided.
    - Establishes a database connection using `create_connection()`.
    - If the connection fails, returns a 500 error with a message.
    - Prepares and executes an SQL query to insert the new task into the database.
    - Commits the transaction and returns a success message in JSON format.
    - Catches and handles any database errors, returning appropriate error messages.
    - Ensures the database connection is closed after the operation.

task_deletion.py

create_connection()

- **Purpose**: Identical to the `create_connection` function in `task_creation.py`.
- **Functionality**:
    - Tries to connect to the database using the provided host, database name, user, password, and port.
    - If the connection is successful, it returns the connection object.
    - If the connection fails, it catches the error, prints an error message, and returns None.

delete_task()

- **Purpose**: Handles the deletion of an existing task by removing the task from the database.
- **Endpoint**: `/delete_task`
- **Method**: POST
- **Functionality**:
    - Retrieves the task ID from the request form.
    - Validates that the task ID is provided.
    - Establishes a database connection using `create_connection()`.
    - If the connection fails, returns a 500 error with a message.
    - Prepares and executes an SQL query to delete the task with the given ID from the database.
    - Commits the transaction.
    - Checks if any rows were affected (i.e., if the task was found and deleted).
    - Returns a success message if the task was deleted, or a 404 error if no task was found with the given ID.
    - Catches and handles any database errors, returning appropriate error messages.
    - Ensures the database connection is closed after the operation.

task_creation.js

Event Listener for Task Creation Form Submission

- **Purpose**: Handles the form submission for creating a new task.
- **Functionality**:
    - Waits for the DOM to be fully loaded.
    - Adds a `submit` event listener to the task creation form (`taskForm`).
    - Prevents the default form submission behavior to handle it via JavaScript.
    - Retrieves task details from the form fields (taskName, taskDescription, startDate, assignUser, createdBy, priorityId).
    - Constructs the form data in a URL-encoded format.
    - Sends an AJAX POST request with the task details to the `/create_task` endpoint.
    - Handles the server response by displaying an alert with the success or error message.
    - Logs any errors to the console.

task_deletion.js

Event Listener for Task Deletion Form Submission

- **Purpose**: Handles the form submission for deleting an existing task.

- **Functionality**:
  - Waits for the DOM to be fully loaded.
  - Adds a `submit` event listener to the task deletion form (`deleteTaskForm`).
  - Prevents the default form submission behavior to handle it via JavaScript.
  - Retrieves the task ID from the form field (taskId).
  - Constructs the form data in a URL-encoded format.
  - Sends an AJAX POST request with the task ID to the `/delete_task` endpoint.
  - Handles the server response by displaying an alert with the success or error message.
  - Logs any errors to the console.

HTML (Modals and Forms)

Task Creation Modal

- **Purpose**: Provides a user interface for creating a new task.
- **Functionality**:
  - Contains a form (`taskForm`) with fields for task name, description, start date, assigned user, and created by.
  - Includes a button to submit the form and a button to close the modal.

Task Deletion Modal

- **Purpose**: Provides a user interface for deleting an existing task.
- **Functionality**:
  - Contains a form (`deleteTaskForm`) with a field for the task ID to be deleted.
  - Includes a button to submit the form and a button to close the modal.

From the first test on it didn't work. I've tried for Hours to fix the Syntax and Structure. I've tried to merge the separate files into one, embedding the Js's into the verify.html, rewriting the connection part in the Scripts, changing classes etc...

I've sat together with Rayan and tried to look for Solutions on the Internet, I've added Error-handling to the Scripts in the hopes of getting more information on why it doesn't work.

I've checked the database countless times, reconnected and made new queries.

I've looked at the Console from VisualStudioCode, and from the Browser but the only error i could find was "Error". Completely unspecified and without any further information there wasn't much that i could do in the remaining Time.

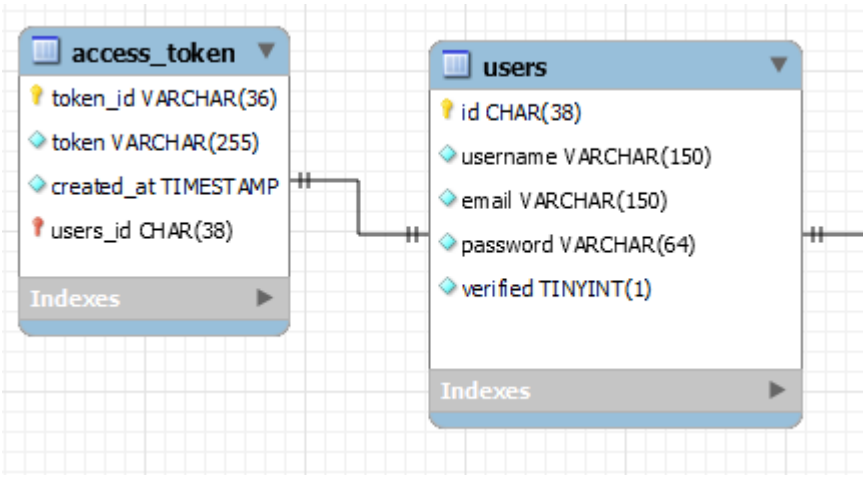
Realization – RLB

When we embarked on our project, we didn't fully grasp the potential implications. We began by developing a blank web application, essentially laying the groundwork for a platform we could later use to implement our processes.

Initially, I had the ambitious idea of creating an inventory management web application. However, realizing the complexity of this task, we opted instead for a more manageable project: developing a task manager.

My responsibilities included designing the basic layout of the web application and implementing the registration and verification functions. These tasks felt straightforward to me, as I had extensive experience with Flask and the Python-MySQL library.

To start, I needed to set up a MySQL database, for which I used a MariaDB Docker container. I then created the database using the following schema:



Note, that this only depicts the relevant tables, for my task.

So I began coding, the first thing I did was creating a simple layout with bootstrap and added some functionality to the interface, as in a dark mode toggler and a clear cookies function. After that I added the registration function and verification function.

I will now add some context as to how the application works.

```python
def create_connection():
    try:
        connection = mysql.connector.connect(
            host='localhost',
            database='testdatabase',
            user='testaccount',
            password='password',      You, 4 hours ago • Unc
            port=3306
        )
        if connection.is_connected():
            return connection
    except Error as e:
        print(f"Error while connecting to MySQL: {e}")
        return None
```

This code snippets is zuständing for the astiblishing the connection with the database we are using, this is done by using these credentials.

```python
@verify.route('/verify', methods=['POST'])
def verify_account():
    """
    Verify user's account with personal access token.
    """
    token = request.form.get('token')

    # Create a database connection
    connection = create_connection()
    if connection:
        cursor = connection.cursor()
        try:
            # Check if the token exists in the database
            cursor.execute("SELECT * FROM access_token WHERE token = %s", (token,))
            token_data = cursor.fetchone()
            if token_data:
                user_id = token_data[3]  # Get the user ID associated with the token
                # Update the user's account status to verified
                cursor.execute("UPDATE users SET verified = 1 WHERE id = %s", (user_id,))
                connection.commit()
                return jsonify({'message': 'Account verified successfully!', 'status': 'text-success'}), 200
            else:
                return jsonify({'message': 'Invalid token!', 'status': 'text-danger'}), 400
        except Error as e:
            return jsonify({'message': f"Error: {str(e)}", 'status': 'text-danger'}), 500
        finally:
            cursor.close()
            connection.close()
    else:
        return jsonify({'message': 'Database connection failed!', 'status': 'text-danger'}), 500
```

This function adds the user to the database by using the data which the user has entered through the Web-Interface. The script knows which data this is because the HTML uses javascript to communicate with the backend.

```javascript
$(document).ready(function() {
    $('#register-form').on('submit', function(e) {
        e.preventDefault();  // Prevent the default form submission

        // Get form data
        let username = $('#register-username').val();
        let email = $('#register-email').val();
        let password = $('#register-password').val();
        let confirmPassword = $('#register-confirm-password').val();

        // Send AJAX POST request to register endpoint
        $.ajax({
            url: '/api/register',
            method: 'POST',
            contentType: 'application/x-www-form-urlencoded',
            data: {
                username: username,
                email: email,
                password: password,
                confirm_password: confirmPassword
            },
            success: function(response) {
                // Update the message element with success response
                $('#message').text(response.message).removeClass().addClass(response.status);
                console.log("Registration response:", response);
            },
            error: function(response) {
                // Update the message element with error response
                $('#message').text(response.responseJSON.message).removeClass().addClass('text-danger');
                console.log("Registration error:", response);
            }
        });
        $.ajax({
            url: '/start-account-creation',  // Adjusted URL
            method: 'POST',
            contentType: 'application/x-www-form-urlencoded',
            data: {
                username: username,
                email: email,
                password: password,
                confirm_password: confirmPassword
            },
            success: function(response) {
                $('#message').text(response.message).removeClass().addClass(response.status);
                console.log("Registration response:", response);
            },
            error: function(response) {
                $('#message').text(response.responseJSON.message).removeClass().addClass('text-danger');
                console.log("Registration error:", response);
            }
            });      You, 6 hours ago • Uncommitted changes
    });
```

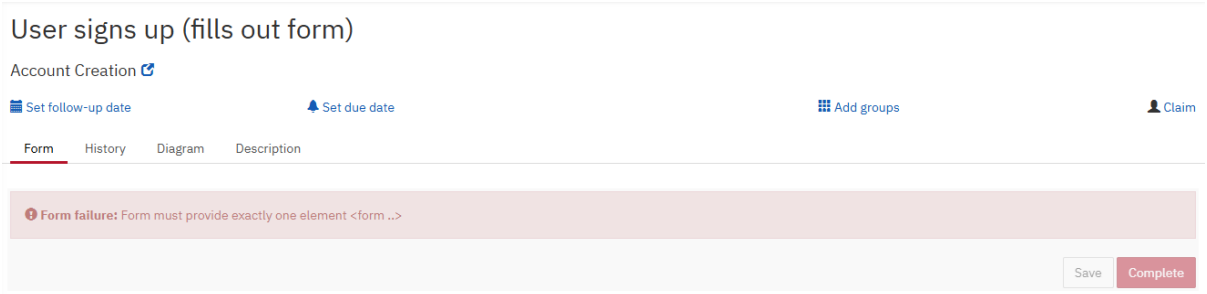
This for example is the js script that handles communication with the backend and also the backend script which communicates with the camunda rest API

This python script actually handles the communication between web-app and rest API

```python
@app.route('/start-account-creation', methods=['POST'])
def start_account_creation():      You, 5 hours ago • Uncommitted changes
    name = request.form['name']
    email = request.form['email']
    password = request.form['password']

    start_instance = pycamunda.processinst.StartInstance(
        url='http://localhost:8080/engine-rest',
        key='accountcreation',
        variables={
            'name': {'value': name, 'type': 'String'},
            'email': {'value': email, 'type': 'String'},
            'password': {'value': password, 'type': 'String'}
        }
    )
    try:
        response = start_instance()
        return jsonify({'message': 'Process started successfully!', 'data': response}), 200
    except pycamunda.PyCamundaException as e:
        return jsonify({'message': str(e)}), 500
```

Now using the Web-application is good but unfortunately Camunda 7 has limitations as it only allows one form per html document. Now that is something unfortunate, I can't do anything about it since Camunda 7 doesn't receive any more updates.



Unfortunately I have way too little time left to accurately depict this as I'd originally intended.

However, this is unfortunately one of the things that one needs to account for. But making up for this I'll make something in Camunda 8 and upload it to my GitHub.

Realization – LS

Ziel

Mein Ziel war es, aus bereitgestellten Registrationsinformationen eine funktionierende Login-Funktion zu erstellen. Ich arbeite bei diesem Projekt das erste Mal mit Python, was für mich eine große Herausforderung, aber auch eine Chance darstellt, um Python zu lernen. Es war auch ein sehr professionell organisiertes Projekt, und daher war ein weiteres Ziel, so viele Learnings wie möglich daraus zu ziehen.

Python Learning

Da ich mit Python, wie gesagt, noch nicht vertraut war, wollte ich zuerst ein wenig in das Thema reinkommen. Dafür habe ich meine ersten kleinen Python-Codes geschrieben und ausprobiert. Dies war mein erstes Script, welches prüft, ob eine Zahl gerade oder ungerade ist:

```python
def ist_gerade(n):
    return n % 2 == 0

zahl = int(input("Bitte geben Sie eine Zahl ein: "))
if ist_gerade(zahl):
    print(f"{zahl} ist gerade.")
else:
    print(f"{zahl} ist ungerade.")


```

Solche kleinen Codes habe ich ein paar geschrieben, einfach um mich im Thema Python wohler zu fühlen. Danach fühlte ich mich bereit, an die Login-Funktion zu treten.

Login-Funktion

Bei der Login-Funktion gab es mehrere Schwierigkeiten. Einerseits war es schwer, sich im Code zurechtzufinden, vor allem mit meinen wenigen Python-Kenntnissen. Zum anderen war mein Wissen trotzdem noch relativ gering und ich musste mir vieles zusammengoogeln.

```python
@auth.route('/login', methods=['POST'])
def login():
    """
    Login a user and return a personal access token.
    """
    username = request.form.get('username')
    password = request.form.get('password')

    # Hash the password
    hashed_password = hashlib.sha256(password.encode()).hexdigest()

    # Create a database connection
    connection = create_connection()
    if connection:
        cursor = connection.cursor()
        try:
            # Verify the user credentials
            cursor.execute("SELECT id FROM users WHERE username=%s AND password=%s", (username, hashed_password))
            user = cursor.fetchone()

            if user:
                user_id = user[0]
                access_token = generate_token()
                created_at = datetime.datetime.now()

                # Insert the personal access token into the database
                cursor.execute("INSERT INTO access_token (token_id, token, created_at, users_id) VALUES (%s, %s, %s, %s)",
                               (str(uuid.uuid4()), access_token, created_at, user_id))

                connection.commit()
                return jsonify({'message': 'Login successful!', 'token': access_token, 'status': 'text-success'}), 200
            else:
                return jsonify({'message': 'Invalid username or password!', 'status': 'text-danger'}), 401
        except Error as e:
            return jsonify({'message': f"Error: {str(e)}", 'status': 'text-danger'}), 500
        finally:
            cursor.close()
            connection.close()
    else:
        return jsonify({'message': 'Database connection failed!', 'status': 'text-danger'}), 500

```

Hier ist die Schritt-für-Schritt-Erklärung des Login-Codes:

1. Route definieren:

@auth.route('/login', methods=['POST'])

- Das ist ein Dekorator in Flask, der eine neue Route (URL) definiert. Diese Route `/login` akzeptiert nur POST-Anfragen. POST-Anfragen werden normalerweise für das Senden von Daten, wie Benutzeranmeldedaten, verwendet.

2. Funktion definieren:

def login():

- Hier definieren wir eine Funktion namens `login`, die ausgeführt wird, wenn jemand die `/login`-Route aufruft.

3. Eingabedaten abrufen:

username_or_email = request.form.get('username_or_email')

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

- Diese Zeilen holen sich die Eingabe des Benutzers aus den Formularfeldern `username_or_email` und `password`. Die Formularfelder werden in einer POST-Anfrage gesendet.

4.Passwort hashen:

hashed_password = hashlib.sha256(password.encode()).hexdigest()

- Diese Zeile hasht das eingegebene Passwort mit dem SHA-256-Algorithmus. Hashing ist eine Technik, um Passwörter sicher zu speichern. Anstatt das Passwort im Klartext zu speichern, speichern wir den Hashwert, der schwer zu erraten ist.

5. Datenbankverbindung herstellen:

connection = create_connection()

- Diese Zeile ruft die Funktion `create_connection()` auf, die eine Verbindung zur Datenbank herstellt. Wenn die Verbindung erfolgreich ist, wird sie in der Variable `connection` gespeichert.

6. Überprüfen der Datenbankverbindung:

if connection:

- Diese Zeile überprüft, ob die Verbindung zur Datenbank erfolgreich war. Wenn `connection` nicht `None` ist, fahren wir fort.

7. Cursor erstellen:

cursor = connection.cursor(dictionary=True)

- Diese Zeile erstellt einen Cursor, der zum Ausführen von SQL-Befehlen in der Datenbank verwendet wird. Der Parameter `dictionary=True` bedeutet, dass die Ergebnisse als Wörterbuch (Dictionary) zurückgegeben werden, was den Zugriff auf die Daten erleichtert.

8. Fehlerbehandlung beginnen:

try:

- Diese Zeile beginnt einen `try` -Block, der Fehlerbehandlung ermöglicht. Wenn ein Fehler auftritt, können wir ihn im `except` -Block behandeln.

9. SQL-Abfrage ausführen:

*cursor.execute("SELECT * FROM users WHERE (username = %s OR email = %s) AND password = %s", (username_or_email, username_or_email, hashed_password))*

- Diese Zeile führt eine SQL-Abfrage aus, um zu überprüfen, ob ein Benutzer mit dem eingegebenen Benutzernamen (oder E-Mail) und Passwort in der Datenbank existiert.

10. Ergebnis der Abfrage abrufen:

user = cursor.fetchone()

- Diese Zeile holt das erste Ergebnis der Abfrage. Wenn kein Benutzer gefunden wird, ist `user` `None`.

11. Überprüfen, ob ein Benutzer gefunden wurde:

if user:

return jsonify({'message': 'Login successful!', 'status': 'text-success'}), 200

else:

return jsonify({'message': 'Invalid username/email or password!', 'status': 'text-danger'}), 401

- Diese Zeilen überprüfen, ob `user` nicht `None` ist, was bedeutet, dass ein Benutzer gefunden wurde. Wenn ein Benutzer gefunden wurde, senden wir eine Erfolgsmeldung zurück und setzen den HTTP-Statuscode auf 200 (OK). Wenn kein Benutzer gefunden wurde, senden wir eine Fehlermeldung zurück und setzen den HTTP-Statuscode auf 401 (Unauthorized).

12. Fehlerbehandlung:

except Error as e:

return jsonify({'message': f"Error: {str(e)}", 'status': 'text-danger'}), 500

- Wenn ein Fehler auftritt, fangen wir ihn hier ab. `e` enthält die Fehlerdetails. Wir senden eine Fehlermeldung zurück und setzen den HTTP-Statuscode auf 500 (Internal Server Error).

13. Ressourcen freigeben:

finally:

cursor.close()

connection.close()

- Dieser Block wird immer ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht. Er schließt den Cursor und die Datenbankverbindung.

14. Datenbankverbindung fehlgeschlagen:

else:

return jsonify({'message': 'Database connection failed!', 'status': 'text-danger'}), 500

- Wenn die Datenbankverbindung fehlgeschlagen ist (d.h., `connection` ist `None`), führen wir diesen Block aus. Wir senden eine Fehlermeldung zurück und setzen den HTTP-Statuscode auf 500 (Internal Server Error).

Der Code ist nicht ganz kompatibel mit der Umgebung, jedoch hatte ich keine Zeit mehr, ihn anzupassen. Der Code wäre jedoch an sich funktional und läuft. Ich bin sehr stolz auf den Code und wie schnell ich Python gelernt habe.

Projekt Learnings

Ich habe in diesem Projekt unabhängig vom Fachwissen sehr viel über Projektmanagement gelernt. Die angesetzten Weeklys waren fixe Termine in der Freizeit, welche das Commitment unter Beweis stellten. Auch die ganze Rollenteilung und Arbeitsaufteilung war sehr professionell gestaltet und ich denke, davon kann ich viel lernen. Es gab auch eine Verfügbarkeitsliste, in der jeder eintrug, wann und über welchen Kanal man verfügbar ist, was eine große Hilfe war.

Fazit

Ich habe in diesem Projekt extrem viel gelernt. Von Projektmanagement über Python zu den dazugehörigen Geschäftsprozessen. Auch die Arbeit im Team lernte ich durch die Weeklys nochmals besser kennen und ich sehe jetzt, wie wichtig der regelmäßige Austausch ist. Das Projekt war für mich ein voller Erfolg und ich bin stolz auf das, was ich in dieser kurzen Zeit erreicht habe.

Technical details

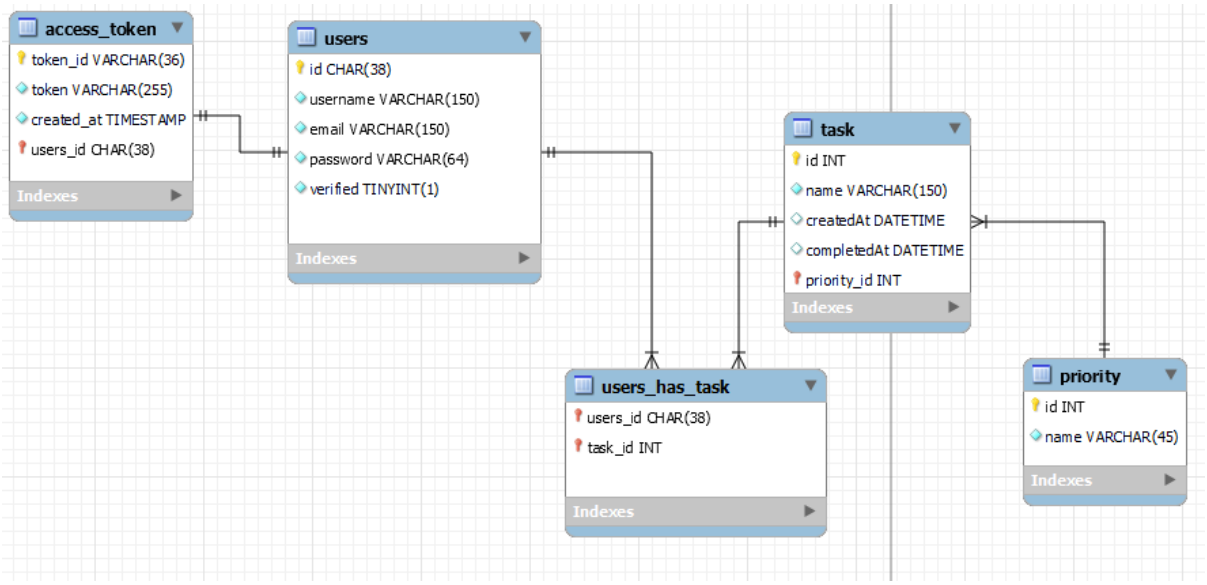
Libraries

Our python script was made with mainly the following libraries: Flask, Pycamunda and MySQL-connector python

- With the MySQL connector python library we could connect and execute remote commands in our MariaDB Database
- With the Flask library we could conjure up a web interface.
- PyCamunda library is an alternative library to Pycamunda external task client which essentially functions the same but has better syntax and looks overall alot cleaner.

Database

For the storing of our user data we used a docker container which hosts a MariaDB database, This database has the following layout.



All entries made into this database are automated.

Python Script

The cool thing about pycamunda is that there is no need to use threading since we pycamunda utilizes this natively, which is very cool. This way we dont have to define multiple threads.

```python
@app.route('/start-account-creation', methods=['POST'])
def start_account_creation():      You, 6 hours ago • Uncommitted changes
    name = request.form['name']
    email = request.form['email']
    password = request.form['password']

    start_instance = pycamunda.processinst.StartInstance(
        url='http://localhost:8080/engine-rest',
        key='accountcreation',
        variables={
            'name': {'value': name, 'type': 'String'},
            'email': {'value': email, 'type': 'String'},
            'password': {'value': password, 'type': 'String'}
        }
    )
    try:
        response = start_instance()
        return jsonify({'message': 'Process started successfully!', 'data': response}), 200
    except pycamunda.PyCamundaException as e:
        return jsonify({'message': str(e)}), 500

@app.route('/start-email-verification', methods=['POST'])
def start_email_verification():
    email_token = request.form['email_token']
    email = request.form['email']

    start_instance = pycamunda.processinst.StartInstance(
        url='http://localhost:8080/engine-rest',
        key='email_verify',
        variables={
            'email_token': {'value': email_token, 'type': 'String'},
            'email': {'value': email, 'type': 'String'}
        }
    )
    try:
        response = start_instance()
        return jsonify({'message': 'Process started successfully!', 'data': response}), 200
    except pycamunda.PyCamundaException as e:
        return jsonify({'message': str(e)}), 500
```

This python code snippet allows us to interact with the REST API of Camunda

Reflection

Teamwork

The teamwork, was alright in the beginning, then picked up in pace around the week 5 and 6 mark as we were in high gear but sadly some of the others dropped their pace and slowed down leading to an embarrassing result. However we still could deliver a project that completes the requirements.

Hardships

> ***In my eyes the main breaking point was the work moral of some of us. I will not mention who but i will mention it. Also there were some hardships when I was coding the Script I***

> ***couldn't get the PyCamunda script to work for the life of me until i stumbled accross a blog post of some random indian guy who explained it pretty well. Also the Input of Mr. Kälin helped alot. Unfortunatelly due to the limitation of the Camunda Engine I can't deliver a completed project however I've learned alot during these few weeks.***

Time management

Even though we had many safe guards in place to ensure that we wouldn't get into a time crunch we got into one. As of now as im writing this it is 1 hour before due date. Which speaks volumes, that even if you have multiple safeguards against something if these safeguards or measures are ignored, they don't wont use anything.

Valuable things I take with me

> ***The most valuable thing I take with me is that I now know how to interact with the camunda REST API and im confident to say if I have another shot at this project I'll absolutely ace it. Something that sticks to me especially is that Camunda doesn't care if you are using an external task it will just wait until you execute the external task.***