Project Report

JCC Cinephile Schedule

Imen Kenza Chouaieb

Senior Student

January 2025

Declaration

I certify that I am the author of this report and that any assistance I have received in its preparation is fully acknowledged and disclosed in this report. I have also cited any source from which I used data, ideas, or words, either quoted or paraphrased. Further, this report meets all of the rules of quotation and referencing in use at Tunis Business School, as well as adheres to the fraud policies listed in the Tunis Business School honor code.

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification to this or any other university or academic institution.

Name	**Signature**
Imen Kenza Chouaieb	———————————-

Abstract

The JCC Carthage Film Festival is one of Tunisia's most anticipated cultural events, drawing significant interest from local and international film enthusiasts. However, one of the prominent challenges attendees face, particularly cinephiles, is the absence of a centralized resource that provides information on the schedules and agendas of various cinema theaters and related ceremonies.

This problem in accessible information can lead to a decreased attendance rate, as attendees often struggle to navigate the numerous offerings and timings associated with the festival. The JCC Cinephile Schedule is developed as a web service solution to address this issue. This aims to centralize all cinema theater programs into a single platform, allowing users to easily access and review schedules, dates, and times associated with film screenings and events. Ultimately, the JCC Cinephile Schedule represents a vital step towards enriching the cultural landscape of Tunisia by promoting accessibility and engagement in the realm of film.

Nomenclature

List of abbreviations

JCC: **J**ournées **C**inématographiques de **C**arthage

API: **A**pplication **P**rogramming **I**nterface

JWT: **J**SON **W**eb **T**okens

Contents

List of Figures

Chapter 1

Project Context

1.1 Introduction

In this project, I focused on conceptualizing and implementing an additional service to be integrated into the JCC Official platform. This new API is designed to enhance the experience of event attendees by providing access to essential information about film screenings and allowing them to schedule their favorite films in their digital calendars.

1.2 Background

The JCC Programs in the Tunis governorate offer a diverse array of cinema venues. In the latest edition, many attendees, especially cinema enthusiasts, expressed their disappointment at the lack of a single location to find all the details regarding film screenings. They noted the lengthy process of checking each venue's screening agenda individually, as well as the time-consuming searches on social media. This fragmented information landscape creates a significant barrier for JCC attendees, hindering their ability to easily discover and plan their film-watching experiences.

1.3 Motivation

My project initiative, the JCC Cinephile schedule, responds to this need and contributes to enriching Tunisia's cultural landscape. In today's fast-paced world, where rapid access to services and information is crucial, it aims to bridge the gap by providing cinema enthusiasts with a centralized and easily accessible platform.

1.4 Objectives

1. Centralizing all information related to film screenings in one tab.
2. Allowing attendees to organize their agendas for the films they plan to attend.

1.5 Project Scope

JCC Cinephile Schedule aims to build a RESTful API that shares all information about film screenings, films, and venues, and manages the schedules of the attendees that contain the screenings they are willing to attend. To achieve this, the project uses Visual Studio Code

with Python and Flask as the main backend framework, Marshmallow for data validation, and Flask-smorest as a REST API framework . In addition, Insomnia is used for API routes testing and GitHub and Git for the project's workflow assistance. In the upcoming version, the project uses SQL-Alchemy, and Migrate for database management,JWT-Manager for the creation, verification, and management of JWT and Flasgger for the API documentation through Swagger.

1.6 Project Versions

- **Version 01:** This initial version is static and meets the goal of information centralization. It gathers information solely from administrators.
- **Upcoming version:** This version introduces interaction with JCC attendees. It enables the collection of information from attendees by allowing them to schedule the film screenings they wish to attend.

Chapter 2

Technical Implementation of the first version

This section will briefly and consistently describe the architecture of the project and the different implementations and technologies.

2.1 Database Structure

The database schema was built in the app itself.

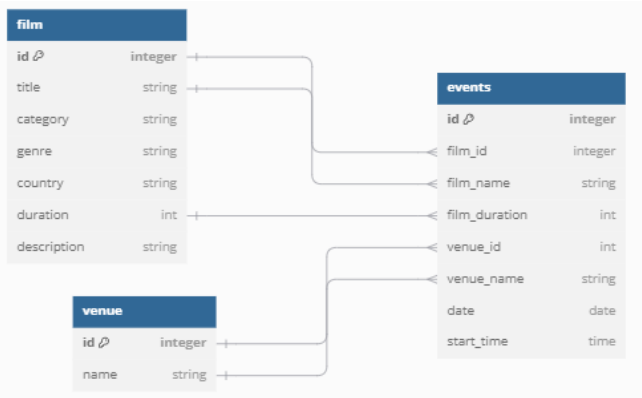

Figure 2.1: Schema Diagram of the Project's Relational Database

2.2 App Backend

2.2.1 Application requests

Film requests

- **Get all films:**

```
@blp.response(200, FilmSchema(many=True))
def get(self):
    return list(db.get("films", {}).values())
```

Figure 2.2: GET all JCC films

- **Post a film:**

```python
@blp.arguments(FilmSchema)
@blp.response(201, FilmSchema)
def post(self, new_film):
    try:
        film_id = max(film.id for film in db.get("films", {}).values()) + 1
    except ValueError:
        film_id = 1

    film = FilmModel(id=film_id, **new_film)
    if "films" not in db:
        db["films"] = {}
    db["films"][film_id] = film
    return film
```

Figure 2.3: POST a new film

- **Get a film by its id:**

```python
@blp.route("/film/<int:film_id>")
class FilmById(MethodView):
    @blp.response(200, FilmSchema)
    def get(self, film_id):
        try:
            return db["films"][film_id]
        except KeyError:
            abort(404, message="Film not found")
```

Figure 2.4: GET information about a specific film by ID

- **Update a film's information by its id:**

```python
@blp.arguments(FilmSchema)
@blp.response(200, FilmSchema)
def put(self, updated_film, film_id):
    try:
        film= db["films"][film_id]
        for key, value in updated_film.items():
            setattr(film,key,value)
        return film
    except KeyError:
       abort(404, message="Film not found")
```

Figure 2.5: PUT information of a film

- **Delete a film:**

```python
def delete(self, film_id):
    try:
        del db["films"][film_id]
        return {"message": "Film deleted."}
    except KeyError:
        abort(404, message="Film not found")
```

Figure 2.6: DELETE a film

- **Get list of films by their category in JCC:**

```python
@blp.route("/film/category/<string:category>")
class FilmByCategory(MethodView):
    @blp.response(200, FilmSchema(many=True))
    def get(self, category):
        films=db.get("films", {}).values()
        filtered_films=[film for film in films if film.category == category]
        return filtered_films
```

Figure 2.7: GET all films of a specific category

- **Get list of films by genre:**

```python
@blp.route("/film/genre/<string:genre>")
class FilmByGenre(MethodView):
    @blp.response(200, FilmSchema(many=True))
    def get(self, genre):
        films=db.get("films", {}).values()
        filtered_films=[film for film in films if film.genre == genre]
        return filtered_films
```

Figure 2.8: GET all films of a specific genre

Venue requests

- **Get all venues:**

```python
@blp.route("/venue")
class VenueList(MethodView):
    @blp.response(200, VenueSchema(many=True))
    def get(self):
        return list(db.get("venues", {}).values())
```

Figure 2.9: GET all JCC venues

- **Post new venue:**

```python
@blp.arguments(VenueSchema)
@blp.response(201, VenueSchema)
def post(self, new_venue):
    try:
        venue_id = max(venue.id for venue in db.get("venues", {}).values()) + 1
    except ValueError:
        venue_id = 1
    venue = VenueModel(id=venue_id, **new_venue)
    if "venues" not in db:
        db["venues"] = {}
    db["venues"][venue_id] = venue
    return venue
```

Figure 2.10: POST a new venue

- **Get venue by id:**

```python
@blp.route("/venue/<int:venue_id>")
class VenueById(MethodView):
    @blp.response(200, VenueSchema)
    def get(self, venue_id):
        try:
            return db["venues"][venue_id]
        except KeyError:
            abort(404, message="Venue not found")
```

Figure 2.11: GET information about a specific venue

- **Update a venue's information by its id:**

```python
@blp.arguments(VenueSchema)
@blp.response(200, VenueSchema)
def put(self, updated_venue, venue_id):
    try:
        venue= db["venues"][venue_id]
        for key, value in updated_venue.items():
            setattr(venue,key,value)
        return venue
    except KeyError:
        abort(404, message="Venue not found")
```

Figure 2.12: PUT information of a venue

- **Delete a venue:**

```python
def delete(self, venue_id):
    try:
        del db["venues"][venue_id]
        return {"message": "Venue deleted."}
    except KeyError:
        abort(404, message="Venue not found")
```

Figure 2.13: DELETE a venue

Event requests

- **Get all events:**

```python
@blp.route("/event")
class EventList(MethodView):
    @blp.response(200, EventSchema(many=True))
    def get(self):
        return list(db.get("events", {}).values())
```

Figure 2.14: GET all JCC films' screenings events

- **Post new event:**

```python
@blp.arguments(EventSchema)
@blp.response(201, EventSchema)
def post(self, new_event):
    try:
        event_id = max(event.id for event in db.get("events", {}).values()) + 1
    except ValueError:
        event_id = 1
    film_name = db.get("films", {}).get(new_event["film_id"]).title
    film_duration = db.get("films", {}).get(new_event["film_id"]).duration
    venue_name = db.get("venues", {}).get(new_event["venue_id"]).name
    event = EventModel(id=event_id,
                       film_id=new_event["film_id"],
                       film_name=film_name,
                       film_duration=film_duration,
                       venue_id=new_event["venue_id"],
                       venue_name=venue_name,
                       date=new_event["date"],
                       start_time=new_event["start_time"])
    if "events" not in db:
        db["events"] = {}
    db["events"][event_id] = event
    return event
```

Figure 2.15: POST a new event

- **Get an event by its id:**

```python
@blp.route("/event/<int:event_id>")
class EventById(MethodView):
    @blp.response(200, EventSchema)
    def get(self, event_id):
        try:
            return db["events"][event_id]
        except KeyError:
            abort(404, message="Event not found")
```

Figure 2.16: GET information about a specific event by ID

- **Update an event by its id:**

```python
@blp.arguments(EventUpdateSchema)
@blp.response(200, EventSchema)
def put(self, updated_event, event_id):
    try:
        event = db["events"][event_id]
        for key, value in updated_event.items():
            setattr(event, key, value)
        return event
    except KeyError:
        abort(404, message="Event not found")
```

Figure 2.17: PUT information of an event

- **Delete an event:**

```python
def delete(self, event_id):
    try:
        del db["events"][event_id]
        return {"message": "Event deleted."}
    except KeyError:
        abort(404, message="Event not found")
```

Figure 2.18: DELETE an event

- **Get all the events of a specific date:**

```python
@blp.route("/event/date/<string:date>")
class EventByDate(MethodView):
    @blp.response(200, EventSchema(many=True))
    def get(self, date):
        events = db.get("events", {}).values()
        parsed_date = datetime.strptime(date, "%Y-%m-%d").date()
        filtered_events = [event for event in events if event.date == parsed_date]
        return filtered_events
```

Figure 2.19: GET all the JCC events of a specific date

- **Get all the events organized for a specific film:**

```python
@blp.route("/event/film/<int:film_id>")
class EventByFilm(MethodView):
    @blp.response(200, EventSchema(many=True))
    def get(self, film_id):
        events=db.get("events", {}).values()
        filtered_events=[event for event in events if event.film_id == film_id]
        return filtered_events
```

Figure 2.20: GET all the JCC events of a specific film

2.2.2 Python Flask Application

The application was developed and deployed using the Flask web framework, a lightweight and modular tool for Python.

```python
if __name__ == '__main__':
    app = create_app()
    app.run(debug=True)
```

Figure 2.21: Application creation and run

2.2.3 Insomnia Testing

The Insomnia tool is an open-source desktop application designed for interacting with and testing APIs. It offers an intuitive interface, supports various protocols (HTTP, REST, GraphQL, etc.), and provides features like authentication helpers, code generation, and environment variables. In this section, the emphasis is placed on testing the HTTP requests made to the API.

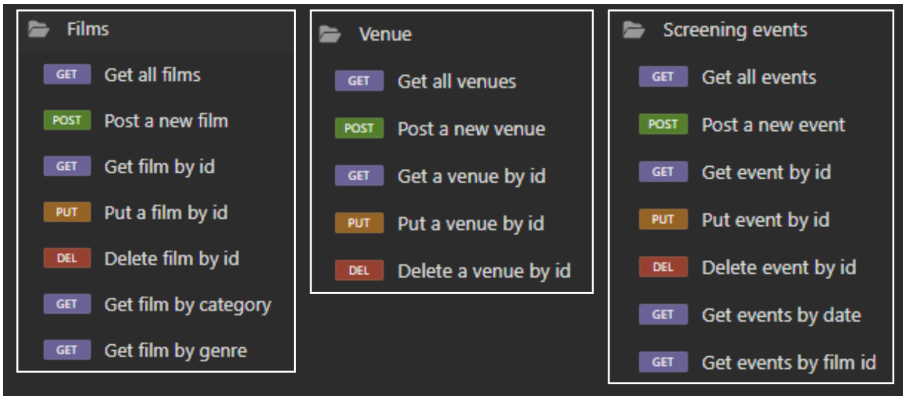

Figure 2.22: All tested routes in Insomnia

2.2.4 Swagger API Documentation

Swagger streamlines the process of maintaining comprehensive documenation by enabling its automatic generation directly from the API definitions, ensuring consistency and currency.

Figure 2.23: Swagger UI

2.2.5 Git and Github for Version Control

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.GitHub is a proprietary developer platform that allows developers to create, store, manage, and share their code. The objective of using Git and GitHub for the project was to implement version control, track changes, and maintain a comprehensive history of the codebase.

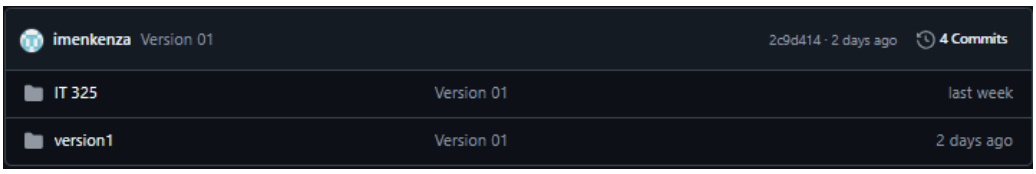

Figure 2.24: Github repository

Chapter 3

Upcoming version

This section will concisely outline the development of the initial version aimed at enhancing the API's dynamism. As this version is still undergoing testing and addressing remaining issues, it is regarded as an upcoming release.

3.1 Database Management

3.1.1 Database Structure

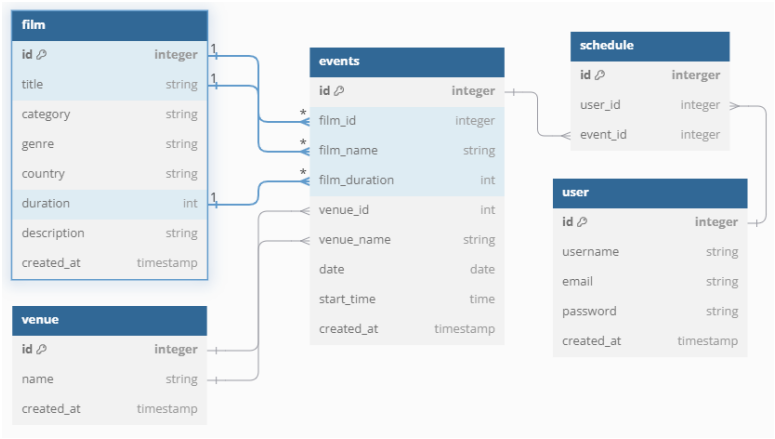

Figure 3.1: Full Schema Diagram of the Relational Database

3.1.2 Database Migration

Flask-Migrate is an extension that handles SQLAlchemy database migrations for Flask applications using Alembic.
MySQL is an open source relational database management system (RDBMS) that's used to store and manage data. Its reliability, performance, scalability, and ease of use make MySQL a popular choice for developers.

```
SQLALCHEMY_DATABASE_URI=mysql+pymysql://root:1234@localhost/jcc_db
```

Figure 3.2: Migration to MySQL

3.2 Insomnia Testing

By upgrading the database, 2 new folders have been created in Insomnia which are: User and Schedule

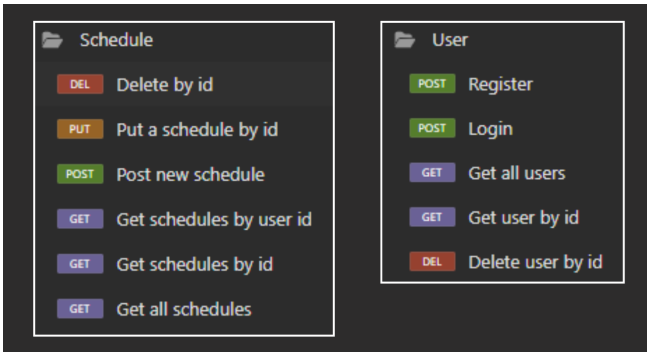

Figure 3.3: The added Insomnia routes for Testing

3.3 JWT for user authentication

JSON Web Tokens are an open, industry standard RFC 7519 method for representing claims securely between two parties. python-jwt is a JSON Web Token (JWT) implementation in Python developed by Gehirn Inc.

```
new_user = User(username=username, email=email, password=hashed_password)

db.session.add(new_user)
db.session.commit()

token = create_access_token(identity=new_user.id)
send_confirmation_email(email, token)
return {'message': "User Created Successfully. Please check your email for confirmation token", 'token':token}, 201
```

Figure 3.4: Tokenization for user registration

Chapter 4

Future Enhancements

4.1 Short-term Enhancements

The following enhancements aim to further develop and improve the web service:

- **Current Code Enhancements:** Necessary modifications will be made to ensure the application runs correctly.
- **Errors Handling:** Implementing more robust error handling mechanisms will make the API more resilient and user-friendly, particularly in the scheduling and login processes.
- **Additional endpoints:** New endpoints will be added to extend functionality, especially for managing events and films.
- **Google Sign-in Integration:** Users will be able to access and interact with the application through their Google accounts.
- **Integration of External APIs:** These APIs will be:
  - **Google Calendar:** Attendees will have their scheduled films automatically added to their Google Calendar, with options to delete and update them as needed.
  - **Google Maps:** The admin will be able to create a custom map displaying all the venues participating in the JCC.

4.2 Long-term Enhancements

The following enhancements aim to expand the service and align it with the changes introduced in different editions of the JCC:

- **Attendee Status Authentication:** Filter attendees into four categories based on their status: Cinema Professional, Cinema Student, University Student, and Other.
- **Tickets Booking System:** Upon completion of attendee status authentication, each category will have specific payment requirements. For example, cinema students can attend up to three films per day for free; additional films will cost 3 TND per film.
- **Database Update and Contenarization:** Add NoSQL data (images, trailers) and update the entire database for the 36th edition of JCC. Additionally, containerize it using Docker due to its volume.
- **Frontend Development:** Develop a new frontend that adheres to the theme and brand guidelines of the 36th edition of JCC.

Chapter 5

Conclusion

This report offers a detailed analysis of the development, testing, and anticipated improvements of the JCC Cinephile Schedule API. The initial version focused on enhancing information sharing with JCC attendees, setting a solid foundation for future enhancements aimed at expanding and refining the service.

Looking ahead, several improvements are planned to further develop the service, ensuring it continues to meet the evolving needs of its users. These enhancements will not only improve the overall user experience but also align with the dynamic nature of the JCC event.

Implementing this API has been a significant milestone in contributing to the cultural landscape of Tunisia. It has been a rewarding experience to play a role in enriching the cultural environment, with aspirations to extend this work to other prestigious events such as the Carthage theater and music festivals.

References

1. Flask
2. Insomnia Testing
3. Swagger API Documentation
4. Git
5. Project's Github Repository
6. Flask-Migrate
7. MySQL
8. JWT
9. Python jwt package