# Tunisian Tapestry RESTful API

Sirine Soltani

January 23, 2024

# Contents

**Abstract**

Addressing the void in the representation of Tunisian events on prominent event platforms, we introduce "Tunisian Tapestry," a Tunisian events explorer API designed to showcase the richness and diversity of cultural events and celebrations in Tunisia. Tailored to diverse interests within the Tunisian entertainment sector, the API aims to boost tourism, fortify the Tunisian community, and support local art and artists. The API empowers users to explore, purchase tickets, and actively contribute by adding their own events, fostering a vibrant and engaged community.

# 1 Introduction

## 1.1 Background

In Tunisia, the digital landscape holds immense potential for showcasing our diverse cultural tapestry, yet, unfortunately, it remains largely untapped. Despite the abundance of cultural events, our unique mosaic of traditions and celebrations often stays hidden from the global and even local stage. This raises the need for Tunisian Tapestry to uplift our local community, support our artists, and enrich our entertainment scene.

## 1.2 Problem Statement

Presently, our culture is underappreciated, with artists lacking necessary support, community satisfaction falling short, and tourism and entertainment industries not reaching their full potential. Addressing these challenges, ”Tunisian Tapestry” emerges as our solution, aiming to support local events and highlight the brilliance of our cultural richness.

## 1.3 Motivation and Objectives

This endeavor is motivated by the profound impact it can have on community well-being, the fostering of artistic expression, the enhancement of tourism, and the overall economic prosperity of Tunisia. ”Tunisian Tapestry” aims to first cater to the needs of our local community, art, and entertainment industry, with a broader vision of eventually showcasing Tunisia’s cultural richness to a global audience.

## 1.4 Overview

The Event Explorer API serves as a platform, enabling users to seamlessly discover events and purchase tickets. Additionally, users can actively contribute by adding their own events, fostering a vibrant community across diverse categories.

# 2 Methodology, Technologies, and Framework

### 2.0.1 Methodology

This API follows RESTful principles, employing a stateless architecture for clear communication and scalability. It utilizes standard HTTP methods and resource-oriented design to ensure a straightforward and efficient approach to handling data. This RESTful foundation prioritizes simplicity, reliability, and easy integration across various applications and platforms.

### 2.0.2 Programming Language

The API was predominantly developed using Python, selected for its versatility, readability, and extensive support for web development.

### 2.0.3 Framework

Flask served as the foundational structure for the API, providing a lightweight yet powerful web framework. Flask's simplicity and flexibility were pivotal in facilitating efficient request handling, data management, and overall system performance.

### 2.0.4 Database Management

SQLite was employed as the Database Management System to establish a structured and scalable database.

### 2.0.5 Migration

Alembic was used to manage database migrations, ensuring a smooth and organized transition.

### 2.0.6 Testing

Endpoint testing was conducted using Postman for comprehensive assessment and validation.

# 3 Implementation

## 3.1 Code Structure and Organization

### 3.1.1 Files Structure

```
your_project/

 app/
    __init__.py
    commands.py
    forms.py
    models.py
    routes.py
    templates/

 cert.pem
 key.pem
 client_secrets.json
 config.py
 main.py
 requirements.txt
```

### 3.1.2 Main Files Overview

`main.py` The entry point for the Flask application. Initializes the app, creates database tables, and runs the app on port 5000 with SSL encryption. SSL encryption is facilitated through self-signed certificates.

`app/__init__.py` Initializes the Flask application with extensions like Flask-Migrate, Flask-JWT-Extended, and Flask-Login. Loads configurations from `config.py` and Google OAuth2 credentials from `client_secrets.json`. Registers blueprints, defines CLI commands, and sets up user loading functions.

`app/models.py` Defines data models (`User`, `Event`, `Participant`, and `Bookmark`) using Flask-SQLAlchemy. Establishes relationships between users, events, and their interactions.

`app/routes.py` Endpoint implementation of the application. It handles various functionalities related to user authentication, event management, registration, etc.

`app/forms.py` Implements Flask-WTF forms for user registration, event creation, and event bookmarking. Validates user-submitted data.

`config.py` Centralizes Flask application configurations. Includes settings for secret key, database URI, CSRF protection key, Google OAuth2 credentials, and JWT token expiration.

#### 3.1.3 Configuration and Security Files

**Authentication Configuration** `Google OAuth Credentials (client_secrets.json):` Stores credentials for Google OAuth2 authentication. Contains client details, project information, and redirect URIs.

**Dependency Management** `requirements.txt:` Lists dependencies for the application. Includes Flask, Flask-SQLAlchemy, Flask-WTF, and others.

**Security Certificates** `SSL Certificates (cert.pem and key.pem):` Provides SSL certificates for secure communication over HTTPS. Essential for encrypting data in transit.

## 3.2 Implementation Details

### 3.2.1 main.py

App initialization:

```
app = create_app()
```

Database Creation :

```
if __name__ == '__main__':
    with app.app_context():
        # Create database tables if they do not exist
        db.create_all()
        logging.info("Database tables created successfully.")
```

App running:

```
app.run(debug=True, port=5000, ssl_context=('cert.pem', 'key.pem'))
```

Runs the Flask application with debugging enabled on port 5000 and sets up SSL encryption using self-signed certificates ('cert.pem' and 'key.pem').

### 3.2.2 app/init.py

**Imported Libraries:**

- Flask: Main Flask class for creating the application instance.
- SQLAlchemy: ORM library for database interactions in Flask.
- LoginManager: Handles user authentication in Flask applications.
- OAuth: Manages OAuth authentication in Flask.
- CSRFProtect: Provides CSRF protection for forms in Flask.

- JSON: Deals with JSON encoding and decoding.
- create_sample_user: Custom CLI command for creating a sample user.
- db: SQLAlchemy database instance.
- JWTManager: Manages JSON Web Tokens in Flask.
- Migrate: Flask-Migrate extension for database migrations.
- Swagger: Flasgger extension for Swagger API documentation.

**Initialization:**

- `client_secrets`: Loads client secrets for OAuth authentication from a JSON file.
- `jwt`: Initializes the JWTManager for handling JSON Web Tokens.
- `login_manager`: Initializes the LoginManager for user authentication.
- `oauth`: Initializes the OAuth extension for OAuth authentication.
- `csrf`: Initializes the CSRFProtect extension for CSRF protection.
- `migrate`: Initializes Flask-Migrate for handling database migrations.
- `client_secrets`: Stores Google OAuth2 credential data loaded from the `client_secrets.json` file in JSON format.

```
jwt = JWTManager()
login_manager = LoginManager()
oauth = OAuth()
csrf = CSRFProtect()
migrate = Migrate()
```

```
% Load client secrets from JSON file
with open('client_secrets.json') as f:
    client_secrets = json.load(f)
```

**Flask Application Configuration:**

- Initializes the Flask application (`app`) and loads configurations from `config.py`.
- Sets the secret key for the application.
- Configures the SQLAlchemy database URI for SQLite.

```
def create_app():
    app = Flask(__name__)

    % Load configuration from config.py
    app.config.from_pyfile('config.py')

    % Set the secret key for the application
    app.config['SECRET_KEY'] = os.environ.get('FLASK_SECRET_KEY
', 'default_secret_key')

    % Set the SQLALCHEMY_DATABASE_URI configuration
    app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///project.
db'

```

**Extensions Initialization:**

- **JWT Initialization:** Initializes the JWT extension.
- **Flask-Migrate Initialization:** Configures Flask-Migrate for database migrations.
- **Swagger Initialization:** Sets up Swagger for enhanced API documentation.
- **CSRF Protection Configuration:** Dynamically enables or disables CSRF protection.
- **Database (db) Initialization:** Initializes the SQLAlchemy database extension.
- **LoginManager Initialization:** Sets up the LoginManager extension.
- **OAuth Initialization:** Configures the OAuth extension.
- **CSRF Protection Initialization:** Initializes the CSRFProtect extension.

```
jwt.init_app(app)
```

```
migrate.init_app(app, db)
```

```
Swagger(app, template_file='C:/Users/Thinkpad/3D Objects/
EventExplorer/swagger.yml')
```

```
app.config['WTF_CSRF_ENABLED'] = False
```

```
db.init_app(app)
```

```
login_manager.init_app(app)
```

```
oauth.init_app(app)
```

```
csrf.init_app(app)
```

**Google OAuth2 Configuration:**

- Configures Google OAuth2 using credentials from `client_secrets.json`.
- Defines the Google OAuth2 remote app with necessary parameters.
- Sets the redirect URI for Google OAuth2.

```
google = oauth.remote_app(
    'google',
    consumer_key=client_secrets['web']['client_id'],
    consumer_secret=client_secrets['web']['client_secret'],
    request_token_params=None,
    base_url='https://www.googleapis.com/oauth2/v1/',
    request_token_url=None,
    access_token_method='POST',
    access_token_url='https://accounts.google.com/o/oauth2/
token',
    authorize_url='https://accounts.google.com/o/oauth2/auth',
)

% Set the redirect_uri using the setter method
google.redirect_uri = "http://127.0.0.1:5000/callback"
```

**Blueprints, User Loader, and CLI Command Registration:**

- Registers blueprint, `main_blueprint`, for core application routes.
- Implements a user loader function for Flask-Login based on the User model.
- Registers the custom CLI command `create_sample_user`.
- To prevent circular import errors, imports the `main_blueprint` from the `app.routes` module at the end of the file to avoid circular import issues.

```
from .routes import main_blueprint
app.register_blueprint(main_blueprint)

```

```
@login_manager.user_loader
def load_user(user_id):
    # Implement the actual user loader based on your User model
    from .models import User
    return User.query.get(int(user_id))
```

```
app.cli.add_command(create_sample_user)
```

### 3.2.3 app/models.py

Details on these models and their relationships will be covered in the data model structure section.

### 3.2.4 app/routes.py

Details on these models and their relationships will be covered in the features and functionalities section. Image

## 3.3 Key Features and Functionalities

### 3.3.1 Overview

The routes and functionality of the API are designed to provide various features related to user authentication, event management, ticket purchase, and bookmarking. Here is a breakdown of the key features and endpoints:

### 3.3.2 1- User Registration

**Endpoint:** /Register

**Request:** Post /Register

**Body:**

```
{
  "username": "Ahmed",
  "email": "ahmed@example.com",
  "password": "securepassword123"
}
```

**Description:** This endpoint handles user registration, ensuring completeness of input data, password hashing, and database storage. The response informs the user about the success of the registration process.

**Logic:**

- Listens for a POST request to the /register endpoint.
- Check if the incoming request data is in JSON format.
- Extracts the username, email, and password from the JSON data.
- Validates that all required fields (username, email, password) are present; otherwise, returns an error response.
- Hashes the provided password using the SHA-256 method.
- Creates a new user in the database with the provided username, email, and hashed password.
- Commits the changes to the database.

- Responds with a JSON message indicating successful account creation and prompts the user to log in.
- If the request format is invalid, returns a JSON error response.

```
@main_blueprint.route('/register', methods=['POST'])
@csrf.exempt
def register():
    if request.is_json:
        % If the request is in JSON format, extract data
        data = request.get_json()

        username = data.get('username')
        email = data.get('email')
        password = data.get('password')

        if not all([username, email, password]):
            return jsonify({'error': 'Incomplete JSON data'}), 400

        % Check if the username is already taken
        existing_username = User.query.filter_by(username=username).first()
        if existing_username:
            return jsonify({'error': 'Username already taken'}), 400

        % Check if the email is already used
        existing_email = User.query.filter_by(email=email).first()
        if existing_email:
            return jsonify({'error': 'Email already used'}), 400

        % Validate the input data as needed (e.g., check email format, password length, etc.)

        hashed_password = generate_password_hash(password, method='sha256')

        new_user = User(
            username=username,
            email=email,
            password_hash=hashed_password
        )

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

        % Assuming registration is successful
        response_data = {
            'message': 'Account created successfully! You can now log in.',
        }
        return jsonify(response_data), 201  % Use the appropriate status code

    return jsonify({'error': 'Invalid request format'}), 400  % JSON response for non-JSON requests
```

### 3.3.3 2- Google OAuth2 Login and Logout

**Endpoints:** /login, /callback, /logout

**Description:** Handles Google OAuth2 login, callback, and logout.

**2.1- Login**

**Request:** GET /login

**Logic:**

- Initiates the login process by generating the authorization URL using the OAuth2 flow.

- If redirection is allowed (ALLOW$_REDIRECTIONisTrue$), $theuserisredirectedtotheauthorizationURL.Ifre$

```
@main_blueprint.route("/login")
def login():

    if ALLOW_REDIRECTION:
        authorization_url, state = flow.authorization_url()

        session["state"] = state
        return redirect(authorization_url)
    else:
        authorization_url, _ = flow.authorization_url()
        print(f"Authorization URL: {authorization_url}")
        return jsonify(message='Redirection disabled for testing', authorization_url=authorization_url)
```

**2.2- Callback**

**Logic:**

- Handles the callback after the user grants permission.
- Fetches the token and verifies the state to ensure security.
- Retrieves user information from the received ID token.
- Stores user details in the session.

- Returns a JSON response indicating a successful callback.

```
@main_blueprint.route("/callback")
def callback():
    if ALLOW_REDIRECTION:
        flow.fetch_token(authorization_response=request.url)

        if not session["state"] == request.args["state"]:
            abort(500)  % State does not match!

        credentials = flow.credentials
        id_info = id_token.verify_oauth2_token(
            id_token=credentials._id_token,
            request=google.auth.transport.requests.Request(
session=request),
            audience=flow.client_config['client_id']
        )

        session["google_id"] = id_info.get("sub")
        session["name"] = id_info.get("name")
        return jsonify(message='Callback successful')  % Adjust
 as needed
    else:
        return jsonify(message='Callback processing disabled
for testing')
```

**2.3- Logout**

**Request:** GET /logout

**Logic:**

- Clears the user's session, effectively logging them out.
- Returns a JSON response indicating a successful logout.

```
@main_blueprint.route("/logout")
def logout():
    session.clear()
    return jsonify(message='Logged out successfully')
```

### 3.3.4 3- JWT Login

**Endpoint:** /login/JWT

**Request:** POST /login/JWT

**Body:**

```
{
  "username": "example_user",
  "password": "password123"
}
```

**Description:** Allows users to log in using a JSON Web Token (JWT).

**Logic:**

- Receives a POST request with JSON data containing the user's credentials (username and password).
- Retrieves the user from the database based on the provided username.
- Checks if the user exists and if the provided password matches the stored hashed password.
- If authentication is successful, generate an access token using Flask-JWT-Extended.
- Returns a JSON response indicating a successful login along with the generated access token.
- If authentication fails, returns a JSON response indicating invalid credentials.

```python
@main_blueprint.route("/login/JWT", methods=["POST"])
def login_normal():
    data = request.get_json()
    username = data.get("username")
    password = data.get("password")


    user = User.query.filter_by(username=username).first()

    if user and user.check_password(password):

        % Generate an access token using Flask-JWT-Extended
        access_token = create_access_token(identity=user.
username)

        return jsonify(message='Login successful', access_token
=access_token), 200
    else:
        return jsonify(message='Invalid credentials'), 401
```

### 3.3.5 5- Searching an Event

**Endpoint:** /search$_{e}vent$

**Request:** GET /search$_{e}vent?title = example_{e}vent$

**Description:** This endpoint allows users to search for an event by its title.

**Logic:**

- Extracts the title from the request URL.
- Searches for the event in the database using the provided title.
- Constructs a response with event details if the event is found.
- Handles various error scenarios and provides appropriate error responses.

```
@main_blueprint.route('/search_event', methods=['GET'])
def search_event():
    title = request.args.get('title')

    if not title:
        return jsonify({'error': 'Please enter a title to
search.'}), 400

    event = Event.query.filter_by(title=title).first()

    if not event:
        return jsonify({'error': 'Event not found!'}), 404

    event_details = {
        'title': event.title,
        'description': event.description,
        'date': event.date.isoformat(),
        'location': event.location,
        'category': event.category,
        'image': event.image,
        "tickets_available": event.tickets_available,
        "ticket_price": event.ticket_price
    }

    return jsonify(event_details)
```

### 3.3.6 4- Listing Events

**Endpoint:** /events

**Request:** GET /events

**Description:** This endpoint retrieves a list of all events available in the database.

**Logic:**

- It does not require authentication.
- Queries the database to retrieve details of all events.
- Constructs a JSON response containing details of each event such as title, description, date, location, category, image, tickets available, and ticket price.

```python
@main_blueprint.route('/search_event', methods=['GET'])
def search_event():
    title = request.args.get('title')

    if not title:
        return jsonify({'error': 'Please enter a title to 
search.'}), 400

    event = Event.query.filter_by(title=title).first()

    if not event:
        return jsonify({'error': 'Event not found!'}), 404

    event_details = {
        'title': event.title,
        'description': event.description,
        'date': event.date.isoformat(),
        'location': event.location,
        'category': event.category,
        'image': event.image,
        "tickets_available": event.tickets_available,
        "ticket_price": event.ticket_price
    }

    return jsonify(event_details)
```

#### 3.3.7 6- Ticket Purchase

**Endpoints:** `/purchase_ticket/<int:event_id>`

**Request:** `GET /purchase_ticket/<event_id>`

**Description:** This endpoint allows a logged-in user to purchase tickets for a specific event.

**Logic:**

- It requires a valid JWT token for authentication.
- Verifies the user's existence based on the JWT token.

- Retrieves the event using the provided event_id.
- Parses and validates the number of tickets to purchase.
- Checks ticket availability and handles various error scenarios.
- Updates the event's available tickets and the participant's ticket purchase record.
- Provides appropriate responses based on the outcome of the ticket purchase process.

```python
@main_blueprint.route('/purchase_ticket/<int:event_id>', methods=['POST'])
@jwt_required()
def purchase_ticket(event_id):
    username = get_jwt_identity()

    # Check if user exists
    user = User.query.filter_by(username=username).first()
    if not user:
        return jsonify(message='User not found'), 404

    event = Event.query.get_or_404(event_id)

    # Parse the number of tickets to purchase from the request
    try:
        num_tickets = int(request.json.get('num_tickets', 1))
        if num_tickets <= 0:
            raise ValueError("Number of tickets must be greater than zero")
    except ValueError:
        return jsonify(message='Invalid input for the number of tickets. Please provide a positive integer.'), 400

    # Check if tickets are available
    if event.tickets_available < num_tickets:
        return jsonify(message=f'Not enough tickets available for this event. Available: {event.tickets_available}'), 400

    participant = Participant.query.filter_by(user_id=user.id, event_id=event_id).first()

    # Deduct the number of tickets from available tickets
    event.tickets_available -= num_tickets

    # Update the participant entry or create a new one
    if participant:
        participant.num_tickets_purchased += num_tickets
    else:
        participant = Participant(user_id=user.id, event_id=event_id, num_tickets_purchased=num_tickets)
```

```python
            db.session.add(participant)

        db.session.commit()

        return jsonify(message=f'{num_tickets} ticket(s) purchased successfully!'), 201
```

### 3.3.8 7- Bookmarking

**Endpoints:** `/bookmark/<int:event_id>`

**Request:** `GET /bookmark/<event_id>`

**Description:** This endpoint allows a logged-in user to bookmark a specific event.

**Logic:**

- It requires a valid JWT token for authentication.
- Retrieves the current user based on the JWT token.
- Check if the user exists and return a 404 Not Found response if not.
- Attempts to retrieve the specified event using the provided event_id.
- Returns a 404 Not Found response if the event with the given ID is not found.
- Check if the event is already bookmarked by the user; if yes, return a 200 OK response.
- If the event is not bookmarked, create a new bookmark entry for the user and the event.
- Commits the changes to the database and returns a 201 Created response upon successful bookmarking.

```python
@main_blueprint.route('/bookmark/<int:event_id>', methods=['POST'])
@jwt_required()
def bookmark_event(event_id):
    # Get the current user
    username = get_jwt_identity()
    user = User.query.filter_by(username=username).first()

    if not user:
        return jsonify(message='User not found.'), 404

    try:
        event = Event.query.get_or_404(event_id)
    except:
        return jsonify(message=f'Event with ID {event_id} not found.'), 404

    # Check if the event is already bookmarked
    if Bookmark.query.filter_by(user_id=user.id, event_id=event_id).first():
        return jsonify(message='Event is already bookmarked.'), 200
    else:
        # Bookmark the event
        bookmark = Bookmark(user_id=user.id, event_id=event_id)
        db.session.add(bookmark)
        db.session.commit()
        return jsonify(message='Event bookmarked successfully!'), 201
```

### 3.3.9 8- Viewing Bookmarked List

**Endpoint:** `/bookmarked_events`

**Request:** `GET /bookmarked_events`

**Description:** This endpoint retrieves a list of events that a user has bookmarked.

**Logic:**

- Requires a valid JWT token for authentication.
- Extracts the username from the JWT token.
- Retrieves the user based on the extracted username.
- Returns a 404 Not Found response if the user is not found.
- Queries the database to retrieve events bookmarked by the user.
- Prepares a JSON response containing details of the bookmarked events.
- The response includes event details such as title, description, date, location, category, and image.

```python
@main_blueprint.route('/bookmarked_events')
@jwt_required()
def bookmarked_events():
    # Get the identity (username) from the JWT token
    username = get_jwt_identity()

    # Assuming you have a User model with a username field
    user = User.query.filter_by(username=username).first()

    if not user:
        return jsonify(message='You need to connect first'), 404

    # Access the bookmarks for the current user
    bookmarked_events = Event.query.join(Bookmark).filter_by(user_id=user.id).all()

    # Prepare the response
    events_data = [{
        'title': event.title,
        'description': event.description,
        'date': event.date.isoformat(),
        'location': event.location,
        'category': event.category,
        'image': event.image,
    } for event in bookmarked_events]

    return jsonify(events=events_data)
```

### 3.3.10 9- Event Creation

**Endpoint:** `/create_event`

**Request:** `POST /create_event`

**Body:**

```
{
  "title": "New Event",
  "description": "Exciting event description",
  "date": "2024-02-01",
  "location": "Venue XYZ",
  "category": "Entertainment",
  "image": "https://example.com/event_image.jpg",
  "tickets_available": 100,
  "ticket_price": 20.00
}
```

**Description:** Allows users to add their events.

**Logic:**

- Requires a valid JWT token for authentication.

- Exempts the endpoint from CSRF protection.
- Validates the form data using the EventForm class, which includes title, description, date, location, category, image, tickets_available, and ticket_price fields.
- Retrieves the username (identity) from the JWT token.
- Retrieves the user object from the database based on the username.
- If the user is not found, returns a 404 error indicating the need to log in.
- If form validation is successful, create a new Event object with the form data and the user's ID.
- Adds the event to the database and commits the changes.
- Responds with a success message if the event creation is successful; otherwise, returns errors if form validation fails or an exception occurs during event creation.

```python
@main_blueprint.route('/create_event', methods=['POST'])
@jwt_required()
@csrf.exempt
def create_event():
    form = EventForm()

    # Get the identity (username) from the JWT token
    username = get_jwt_identity()

    # Assuming you have a User model with a username field
    user = User.query.filter_by(username=username).first()

    if not user:
        return jsonify(message='You need to login first'), 404

    if form.validate_on_submit():
        try:
            event = Event(
                title=form.title.data,
                description=form.description.data,
                date=form.date.data,
                location=form.location.data,
                category=form.category.data,
                image=form.image.data,
                tickets_available=form.tickets_available.data,
                ticket_price=form.ticket_price.data,
                user_id=user.id
            )

            db.session.add(event)
            db.session.commit()

            return jsonify(message='Event created successfully!'), 201
        except Exception as e:
            return jsonify(message=f'Error creating event: {str(e)}'), 500

    return jsonify(errors=form.errors), 400

```

## 3.4 Data Model Structure

### 3.4.1 1. Model Architecture Overview

The application employs an Object-Relational Mapping (ORM) architecture using Flask-SQLAlchemy. This allows us to define data models as Python classes, which are then translated into database tables. This approach simplifies database interactions by providing a high-level, Pythonic representation of the data.

### 3.4.2 2. Data Models

**User Model (User):** Represents user information.

**Attributes:** id (Primary Key), username, email, password_hash, events, bookmarks.

**Methods:**

- `get_all_users()`: Returns a JSON representation of all users.
- `check_password(password)`: Verifies user password.

**Event Model (Event):** Represents event details.

**Attributes:** id (Primary Key), title, description, date, location, category, image, price, user_id (Foreign Key), participants.

**No specified methods.**

**Participant Model (Participant):** Represents the relationship between users and events.

**Attributes:** user_id (Foreign Key), event_id (Foreign Key), num_tickets_purchased.

**No specified methods.**

**Bookmark Model (Bookmark):** Represents the bookmarking of events by users.

**Attributes:** user_id (Foreign Key), event_id (Foreign Key).

**No specified methods.**

### 3.4.3 3. Model Relationships

**User Model:**

**One-to-Many relationship with the Event model:**

```
events = db.relationship('Event', backref='organizer', lazy=True)
```

**One-to-Many relationship with the Bookmark model:**

```
bookmarks = db.relationship('Bookmark', backref='user', lazy=True)
```

**Event Model:**

**Many-to-One relationship with the User model:**

```
user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
```

**One-to-Many relationship with the Participant model:**

```
participants = db.relationship('Participant', backref='event_relationship', lazy=True)
```

**One-to-Many relationship with the Bookmark model:**

```
bookmarks = db.relationship('Bookmark', backref='event', lazy=True)
```

**Participant Model:**

**Many-to-One relationship with the User model:**

```
user_id = db.Column(db.Integer, db.ForeignKey('user.id'), primary_key=True)
```

**Many-to-One relationship with the Event model:**

```
event_id = db.Column(db.Integer, db.ForeignKey('event.id'), primary_key=True)
```

**Many-to-One relationship with the Event model using backref:**

```
event = db.relationship('Event', backref='participants_relationship', lazy=True)
```

**Bookmark Model:**

**Many-to-One relationship with the User model:**

```
user_id = db.Column(db.Integer, db.ForeignKey('user.id'), primary_key=True)
```

**Many-to-One relationship with the Event model:**

```
event_id = db.Column(db.Integer, db.ForeignKey('event.id'), primary_key=True)
```

**Many-to-One relationship with the Event model using backref:**

```
event = db.relationship('Event', backref='bookmarks', lazy=True)
```

## 3.5 Challenges

### 3.5.1 Dependency Version Conflicts

**Challenge:** Dealing with version conflicts, specifically with Flask-OAuthlib and Werkzeug. These conflicts led to issues in functionality, such as errors related to OAuth2 login and URL handling.

**Solution:** Searched for and selected compatible versions of dependencies rather than opting for the latest releases.

### 3.5.2 Database Migrations and Schema Changes

**Challenge:** Evolving database requirements and schema changes during development, leading to potential data migration challenges.

**Solution:** Implemented a database migration tool with Flask-Migrate to handle schema changes efficiently. Regularly performed database migrations and maintained a versioned migration history to manage changes seamlessly.

### 3.5.3 Testing OAuth2 Without Frontend

**Challenge:** Testing Google OAuth2 login functionality without a frontend posed challenges, as manual testing required intricate setup and verification steps.

**Solution:** Disabled automatic redirection, manually simulated OAuth2 interactions using Postman, crafting requests for authorization codes and access tokens.

### 3.5.4 Error Handling and Logging

**Challenge:** A wide variety of errors encountered during testing, coupled with a tedious troubleshooting phase.

**Solution:** Addressed the challenge by significantly improving error handling mechanisms. Implemented custom error messages and expanded logging capabilities to cover a wide range of errors encountered during testing. This enhancement provided detailed and clear information about errors, facilitating a more efficient diagnosis.

## 4 Conclusion and Perspectives

In conclusion, the development of our API has been a significant milestone, with a primary focus on backend functionalities aligned closely with the project's specific goals. By prioritizing key functionalities essential to the project's context, we have established a solid foundation that caters to immediate requirements. It is important to acknowledge that the current state of the API represents just the beginning of its potential. The API has numerous opportunities for expansion, evolution, and value enhancement.

Looking forward, the evolution of our API will be guided by a strategic vision aimed at maximizing its impact and ensuring sustained success. The following perspectives outline our future aspirations and focus areas:

**Data Gathering:**

- Diversify Sources: Expand data sources to enrich the dataset and provide more comprehensive insights.
- Real-time Updates: Aim for real-time data updates to deliver the latest information to users efficiently.

**Database Scaling:**

- Elastic Infrastructure: Implement scalable infrastructure to handle varying data loads dynamically.
- Cloud-Based Solution: Explore cloud options for elastic scalability, optimizing performance, and resource efficiency.

**Security Enhancements:**

- Proactive Measures: Invest in proactive threat intelligence to address security risks before they impact the API.
- Multi-Factor Authentication: Strengthen security with multi-factor authentication for user access control.

**Intuitive Front-end Development:**

- User-Centric Design: Prioritize user feedback for iterative design updates, ensuring an intuitive and enjoyable user experience.
- Responsive Design: Ensure the front-end adapts seamlessly to various devices and screen sizes, providing a consistent user experience.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.