**Step 1: Define Requirements**

* **Features**:
  + Users can select a day, genre, and location.
  + API returns a list of nearby events.
  + An interactive map displays event locations.
  + Optional: Include filtering by event type (e.g., movie, panel, workshop).
* **Technology Stack**:
  + Backend: Flask/Django (Python) or Node.js.
  + Database: PostgreSQL with PostGIS (for geospatial queries) or MongoDB.
  + Mapping API: Leaflet.js, Google Maps API, or OpenStreetMap.
  + Deployment: Heroku, AWS, or Vercel.

**Step 2: Design Database**

* **Tables/Collections**:
  1. **Events Table**:
     + id: Unique event identifier.
     + name: Event name.
     + description: Event details.
     + date: Date and time of the event.
     + genre: Genre or type of event (e.g., drama, workshop).
     + location: Geolocation coordinates (latitude, longitude).
     + venue: Address or venue name.
  2. **Venues Table** (optional):
     + id: Venue identifier.
     + name: Venue name.
     + address: Full address.
     + coordinates: Latitude and longitude.

**Step 3: Implement API Endpoints**

* **API Structure**:
  1. GET /events: Returns a list of events based on filters (date, genre, location).
     + **Parameters**:
       - date (optional): Filter events by day.
       - genre (optional): Filter events by genre.
       - location (required): Latitude, longitude of the user’s location.
       - radius (optional): Distance in kilometers/miles.
     + **Response**: List of events with geolocation details.
  2. GET /event/{id}: Returns detailed information about a specific event.
     + **Parameters**: id (event ID).

**Step 4: Build Interactive Map**

* Use a frontend framework (React, Vue.js, or plain JavaScript).
* Integrate a mapping library:
  + **Markers**: Display events as clickable markers.
  + **Popups**: Show event details when a user clicks a marker.
  + **Search Box**: Allow users to input their location or select filters.

**Step 5: Geospatial Queries**

* Use geospatial capabilities for filtering events by proximity:
  + PostgreSQL: ST\_DWithin for calculating distance between user location and event venues.
  + MongoDB: $geoNear for similar geospatial queries.

**Step 6: Notifications**

* Add optional user notifications:
  + Store user preferences for events they want to follow.
  + Send email or app notifications using tools like Twilio SendGrid or Firebase.

**Step 7: Testing**

* Write unit tests for each API endpoint using tools like Postman or Pytest.
* Test map interactivity and geospatial filtering.

**Step 8: Deployment**

* Host the API and frontend:
  + Backend: Deploy on Heroku, AWS, or Render.
  + Frontend: Host on Netlify or Vercel.
  + Mapping API: Ensure appropriate API keys and quotas for public use.

**Bonus Features (Optional Enhancements):**

* Add support for event reviews or ratings.
* Allow users to bookmark or share events.
* Show route suggestions to venues using mapping APIs.

**Blueprint Files in resources/ Folder**

**1. movies.py**

Handles CRUD operations for the movies table.

**Endpoints:**

* GET /movies: Retrieve all movies.
* GET /movies/<movie\_id>: Retrieve a specific movie by its ID.
* POST /movies: Add a new movie.
* PUT /movies/<movie\_id>: Update an existing movie by ID.
* DELETE /movies/<movie\_id>: Delete a movie by ID.

**2. programs.py**

Handles CRUD operations for the programs table.

**Endpoints:**

* GET /programs: Retrieve all programs.
* GET /programs/<program\_id>: Retrieve a specific program by ID.
* POST /programs: Add a new program.
* PUT /programs/<program\_id>: Update an existing program by ID.
* DELETE /programs/<program\_id>: Delete a program by ID.

**3. venues.py**

Handles CRUD operations for the venues table.

**Endpoints:**

* GET /venues: Retrieve all venues.
* GET /venues/<venue\_id>: Retrieve a specific venue by ID.
* POST /venues: Add a new venue.
* PUT /venues/<venue\_id>: Update an existing venue by ID.
* DELETE /venues/<venue\_id>: Delete a venue by ID.

**4. agenda.py**

Handles CRUD operations for the agenda table.

**Endpoints:**

* GET /agenda: Retrieve all agenda entries.
* GET /agenda/<agenda\_id>: Retrieve a specific agenda entry by ID.
* POST /agenda: Add a new agenda entry.
* PATCH/agenda/<agenda\_id>: Update an existing agenda entry by ID.
* DELETE /agenda/<agenda\_id>: Delete an agenda entry by ID.
* GET /agenda/venue/<venue\_id>: Retrieve agenda entries for a specific venue.
* GET /agenda/movie/<movie\_id>: Retrieve agenda entries for a specific movie.

**Examples of Additional Endpoints**

**Custom Endpoint Ideas**

1. **GET /programs/<program\_id>/movies**
   * Retrieve all movies in a specific program.
2. **GET /venues/<venue\_id>/agenda**
   * Retrieve all agenda entries for a specific venue.
3. **GET /movies/<movie\_id>/agenda**
   * Retrieve all agenda entries for a specific movie.
4. **GET /movies/search?title=<title>**
   * Search for movies by title.
5. **GET /venues/type/<type>**
   * Retrieve venues filtered by type (cinema, theatre, etc.).

Ah, I see! You're asking how to return the **JWT token** to the client and store it in a **cookie** (instead of showing it in the console or response body). This is a common practice to enhance security and usability, especially when dealing with authentication for web apps.

**Storing JWT in Cookies**

You can store the JWT token in an **HTTP-only cookie**, which is not accessible to JavaScript (for security purposes) but can be sent automatically with every request to the server.

Here’s how to modify the response so the JWT is stored in a cookie:

**Modified Code Example: Storing JWT in a Cookie**

1. **Set JWT in a Cookie**:

python

Copy code

from flask import Flask, jsonify, request, make\_response

from flask\_jwt\_extended import JWTManager, create\_access\_token, set\_access\_cookies

app = Flask(\_\_name\_\_)

app.config["JWT\_SECRET\_KEY"] = "your\_secret\_key" # Strong secret key

jwt = JWTManager(app)

# Mock database

USERS\_DB = {

"admin": {"password": "adminpass", "role": "admin"},

"user": {"password": "userpass", "role": "user"},

}

@app.route("/login", methods=["POST"])

def login():

username = request.json.get("username")

password = request.json.get("password")

# Check if user exists and password matches

user = USERS\_DB.get(username)

if not user or user["password"] != password:

return jsonify({"msg": "Invalid username or password"}), 401

# Create JWT with user's role

access\_token = create\_access\_token(identity={"username": username, "role": user["role"]})

# Create a response object

response = make\_response(jsonify({"msg": "Login successful"}))

# Set the JWT as an HTTP-only cookie

set\_access\_cookies(response, access\_token)

return response

if \_\_name\_\_ == "\_\_main\_\_":

app.run(debug=True)

**Key Points**

1. **set\_access\_cookies(response, access\_token)**:
   * This function sets the access\_token in the response as an **HTTP-only cookie**.
2. **Response**: The client will receive the JWT in a cookie called access\_token\_cookie, instead of in the response body. Example response:

http

Copy code

HTTP/1.1 200 OK

Set-Cookie: access\_token\_cookie=eyJhbGci...; HttpOnly; Path=/; SameSite=Lax

Content-Type: application/json

1. **Cookie Properties**:
   * **HttpOnly**: Prevents JavaScript access (defense against XSS attacks).
   * **SameSite=Lax**: Mitigates CSRF attacks by restricting cookie usage across sites.

**Accessing JWT from Cookies**

When the client makes requests to protected endpoints, the JWT token in the cookie is automatically sent. For example:

1. **Protecting Routes**: Use the @jwt\_required() decorator to require the JWT.

python

Copy code

from flask\_jwt\_extended import jwt\_required, get\_jwt\_identity

@app.route("/protected", methods=["GET"])

@jwt\_required()

def protected():

user = get\_jwt\_identity() # Retrieves the user from the token

return jsonify({"msg": f"Hello, {user['username']}! Your role is {user['role']}."})

1. **Client Request**: The cookie will be automatically sent by the browser:

http

Copy code

GET /protected HTTP/1.1

Cookie: access\_token\_cookie=eyJhbGci...

**Testing in the Browser Console**

If you want to confirm that the cookie has been set:

1. Open **Developer Tools** (usually F12 or Ctrl+Shift+I in your browser).
2. Go to the **Application** tab (or **Storage** tab in Firefox).
3. Look under **Cookies** for your application’s domain. You’ll see the access\_token\_cookie.

**Advantages of Using Cookies**

 **HttpOnly** cookies cannot be accessed by JavaScript, protecting against XSS.

 CSRF protections can be applied using SameSite or additional CSRF tokens.