CPSC 471 Database Management Systems
Project Proposal: Fictional World Navigation System

T03-7

Fall 2025

1 Introduction

Our team proposes to develop a web-based navigation system for a fictional world, inspired by real-world mapping applications like Google Maps but designed for creative worldbuilding and fantasy settings. This application will allow users to create custom locations, landmarks, and networks within an imaginary world, then calculate optimal routes between destinations based on various criteria such as time of day, chosen mode of transport and any desired pit stops. We will call this web-based navigation system: **Orbis**, Greek for "*around the world.*"

T03-7 Team Members: Yahya Asmara, Abdulrahman Negmeldin, Jason Duong

2 Domain Description

2.1 Overview

The Fictional World Navigation System will serve as a comprehensive mapping and routing platform for fictional worlds, enabling users to:

- Create and manage user accounts with personalized profiles
- Define custom locations with detailed attributes (capacity, parking, accessibility)
- Establish landmarks as points of interest within locations
- Design networks with roads connecting various locations
- Select from multiple modes of transport with varying characteristics
- Plan routes with real-time calculations for distance, time, and cost
- Apply time-based restrictions on road usage for different transport types
- Manage multi-currency systems for location-based transactions

2.2 Application Functionality

The system will be developed as a Single Page Application (SPA) with clear separation between frontend, backend API, and database layers. Development responsibilities are assigned based on team member strengths through Discord coordination and ongoing negotiation.

Frontend-Backend API Design: One team member will design the API contract, defining RESTful endpoints for user authentication, location management, route calculation, and transportation data. This includes specifying request/response formats, HTTP methods, status codes, and error handling patterns.

User Interaction Design: Another team member will map user interactions to API calls, determining which UI components trigger backend requests. This includes defining when data fetches occur (page load, button clicks, form submissions), implementing loading states, and handling asynchronous operations for route visualization and location creation.

Backend-Database Integration: A team member will design the data access layer, specifying how PostgreSQL [4] models interact with the database through ORM operations. This includes query optimization strategies, transaction management, and implementing the pathfinding algorithm with efficient database reads.

API Endpoint Stubs: Initial endpoint stubs will be created early to establish the development contract between frontend and backend teams, allowing parallel development with mock data before full implementation.

SPA Architecture: The SPA navigation structure will be designed with distinct views (dashboard, location creator, route planner, user profile) using client-side routing. State management patterns will be defined to handle navigation without full page reloads while maintaining URL synchronization.

Note: These role assignments are subject to change as the project evolves and additional tasks are identified. Team members will reassign responsibilities as needed to accommodate workload balance and leverage individual expertise.

3 Entity-Relationship Model

3.1 ER Diagram

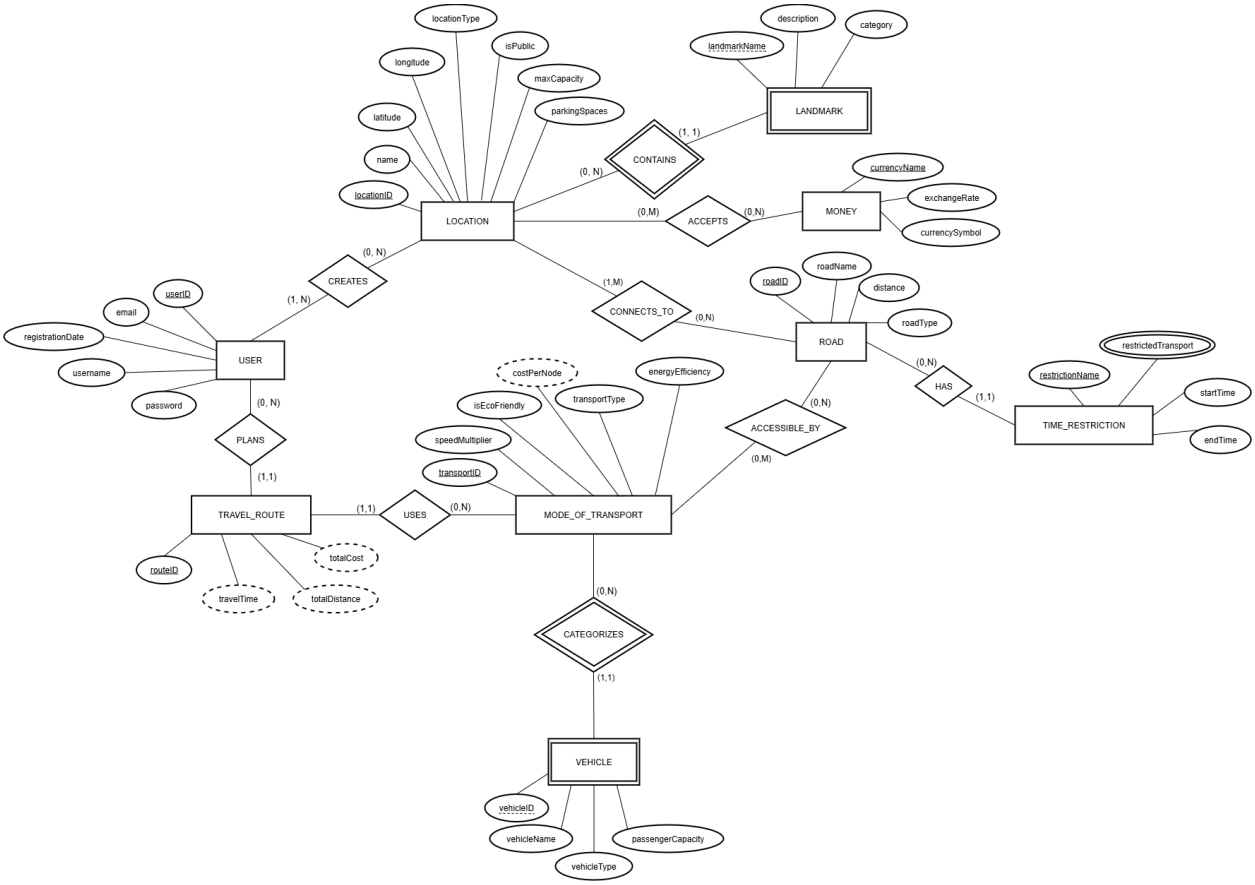


Figure 1: Entity-Relationship Diagram

Note that this is subject to change. The diagram is in parenthetical (min, max) notation.

3.2 Entities and their Attributes

Note that this is subject to change.

3.2.1 User and Travel Management

USER

- userID (Primary Key)
- email
- password

- username
- registrationDate

TRAVEL_ROUTE

- routeID (Primary Key)
- userID (Foreign Key)
- startLocationID (Foreign Key)
- endLocationID (Foreign Key)
- transportID (Foreign Key)
- travelTime (Derived)
- totalDistance (Derived)
- totalCost (Derived)

3.2.2 Location and Infrastructure

LOCATION

- locationID (Primary Key)
- name
- latitude
- longitude
- locationType
- isPublic
- maxCapacity
- parkingSpaces

ROAD

- roadID (Primary Key)
- roadName
- startLocationID (Foreign Key)
- endLocationID (Foreign Key)
- distance

- roadType

LANDMARK (Weak Entity)

- landmarkName (Partial Key)
- locationID (Foreign Key - identifying relationship)
- category
- description

3.2.3 Transportation and Economics

MODE_OF_TRANSPORT

- transportID (Primary Key)
- transportType
- speedMultiplier
- costPerNode (Derived)
- isEcoFriendly
- energyEfficiency

VEHICLE (Weak Entity)

- vehicleID (Partial Key)
- transportID (Foreign Key - identifying relationship)
- vehicleName
- vehicleType
- passengerCapacity

MONEY

- currencyName (Primary Key)
- currencySymbol
- exchangeRate

TIME_RESTRICTION

- restrictionName (Primary Key)
- roadID (Foreign Key)
- startTime
- endTime
- restrictedTransport (Multi-valued)

3.3 Relationships

1. **USER creates LOCATION** (1:N): User can create multiple locations; constraint enforces that new locations must connect to existing roads
2. **USER plans TRAVEL_ROUTE** (1:N): User can plan multiple routes through the system
3. **LOCATION contains LANDMARK** (1:1): Locations can only have one landmark; landmarks are weak entities dependent on locations
4. **LOCATION connects_from/to ROAD** (N:M): Roads connect two locations bidirectionally
5. **MODE_OF_TRANSPORT categorizes VEHICLE** (1:N): Each mode of transport can have multiple specific vehicle types
6. **MODE_OF_TRANSPORT uses TRAVEL_ROUTE** (1:N): Each route uses one mode of transport
7. **ROAD has TIME_RESTRICTION** (1:N): Roads can have multiple time-based restrictions
8. **LOCATION accepts MONEY** (N:M): Locations can accept multiple currencies
9. **ROAD accessible_by MODE_OF_TRANSPORT** (N:M): Roads may be accessible by certain transport modes only

3.4 ER Diagram Assumptions

The following assumptions guide our entity-relationship design and implementation approach:

3.4.1 Assumptions Organized by Entity

USER

- All values for attributes affiliated with the USER entity cannot be null
- Each instance of TRAVEL_ROUTE represents a travel route created for one user

TRAVEL_ROUTE

- `totalDistance` has its value derived from the `longitude` and `latitude` values from LOCATION instances whose IDs are `startLocationID` and `endLocationID`
- `travelTime` has its value derived from stored data like the longitude and latitude values from LOCATION instances whose IDs are `startLocationID` and `endLocationID` and the `speedMultiplier` value from a MODE_OF_TRANSPORT instance whose ID is `transportID`

- `totalCost` has its value derived from stored data like the `longitude` and `latitude` values from LOCATION instances whose IDs are `startLocationID` and `endLocationID`, the `costPerNode` value from a MODE_OF_TRANSPORT instance whose ID is `transportID`. `totalCost` is calculated to lower as travel time decreases when the MODE_OF_TRANSPORT instance is of `transportType` Car or Bus, or as travel time increases when the instance is of `transportType` Walking

LOCATION

- `locationType` will be one of the following values: {Hotel, Park, Cafe, Restaurant, Landmark, Gas_Station, Electric_Charging_Station}
- `isPublic` can be only one of {yes, no}

ROAD

- `startLocationID` and `endLocationID` belong to instances of LOCATION who are connected by a ROAD instance
- `roadType` can only be one of {blocked, unblocked} and its value represents whether a user can create a TRAVEL_ROUTE instance that utilizes this ROAD instance

LANDMARK

- `category` will be one of the following values: {Mountain, River, Lake, In_City, Other}

MODE_OF_TRANSPORT

- `transportType` can only be one of {Car, Bicycle, Bus, Walking}
- The value of `speedMultiplier` represents the number of nodes the instance can travel in one tick
- The value of `costPerNode` represents the amount of fuel required to travel past a node and is derived from `energyEfficiency`
- The value of `isEcoFriendly` represents whether a vehicle is an electric vehicle and can only be one of the following values: {yes, no}
- The value of `energyEfficiency` represents a rate that is used to determine `costPerNode`

VEHICLE

- `vehicleType` can only be one of: {Car, Bus}

TIME_RESTRICTION

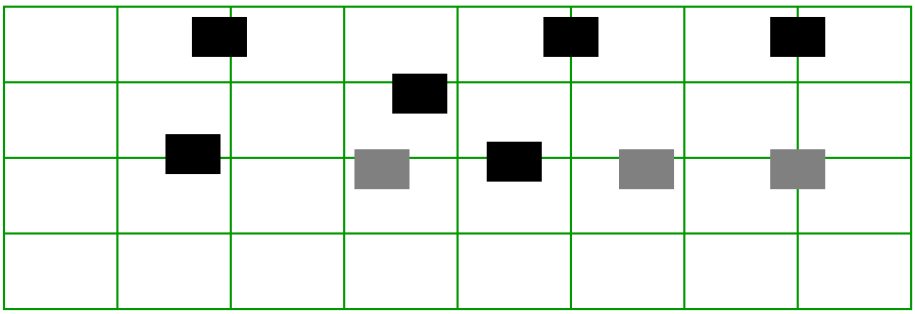
- `roadID` is the `roadID` of a ROAD instance where this TIME_RESTRICTION instance applies to
- The value set of `restrictedTransport` can be any of {Car, Bicycle, Bus, Walking}

3.4.2 User-Specific Graph Extension

Users can extend the navigation graph by adding custom locations to a predefined base map. Each user maintains their own local version of the map with their additions stored in the database associated with their userID. Key constraints include:

- Any newly created location must connect to at least one pre-existing location via a road
- Route calculations require valid road connections between locations
- User-created content (locations, roads, landmarks) is stored locally to that user's profile
- The base map provides the initial graph structure that all users begin with

NOTE: This map is subject to change. Below is a $m \times n$ map with the grid as ROADS. We may possibly have a **graph** of ROAD entities as edges and LOCATION entities as vertices.



Legend:
Predefined Location User-Created Location Road Network

3.4.3 Map Architecture and Scalability

NOTE: this section is more for to answer the question of app scalability. It is a feature that could be implemented but might not.

If implementing a "global map" feature where users can contribute to a shared world:

- Limit contributions to one location per user on the global map
- Define maximum location capacity based on map dimensions ($m \times n$ grid)

- Once all grid positions are filled, no additional locations can be placed
- This prevents unbounded growth and maintains map manageability

3.4.4 Real-Time Communication

For any real-time features (if global changes are implemented):

- WebSocket [10] connections between frontend and Flask [1, 2] backend enable push notifications
- When database changes occur, Flask pushes messages over WebSocket
- Frontend interprets messages and updates the UI accordingly
- Expected latency of approximately 2 seconds for updates is acceptable for our use case
- If real-time global updates prove complex, the application defaults to client-side only changes

3.4.5 Deployment and Multi-User Access

The application will be hosted on Render [9] or similar platforms:

- Flask [1, 2] serves frontend files (HTML [5], CSS [5], JavaScript [5]/React [7])
- User interactions that modify data send API requests to Flask
- Flask processes requests and updates the database via PostgreSQL [4]
- Multiple concurrent users are supported through standard HTTP request handling

3.4.6 Obstacle and Terrain Management

NOTE: Obstacles may not be in the final application. Depending on implementation, we may not include obstacles. The map includes predefined obstacles (mountains, terrain features, roadblocks):

- Obstacles are represented in a grid array structure
- Null values or special objects denote impassable locations
- Obstacles are preset and do not change during normal operation
- Frontend provides visual indicators (different colors, icons) for impassable terrain
- Obstacles prevent unrealistic pathfinding scenarios and add navigational complexity
- Without obstacles, the map would become cluttered with user-created paths and lack realism

3.4.7 Implementation Simplification

If extending the network proves difficult to implement:

- Fall back to the predefined base map without user extensions
- Focus on other route calculation and visualization features
- User contributions limited to route planning rather than map modification

4 Technical Framework: Flask

As the project requirement, we have selected Python + Flask [1, 2] as our web framework for this project. Flask is a lightweight and flexible Python web framework that provides the essential tools needed for web application development without imposing rigid structure. As part of experimenting however, we may in addition to Flask serve the React [7] framework to employ the Three.js [6] library in order to develop the mapping system for the application.

4.1 Reference One

Reference: Grinberg, M. (2018). *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media. ISBN: 978-1491991732.

Justification: This comprehensive guide provides extensive documentation on Flask's routing system, template rendering with Jinja2, and database integration with PostgreSQL. The book's focus on RESTful API design and form handling makes it particularly relevant for our transaction-based web application. Grinberg's work is widely recognized in the Flask community and offers practical examples that align with our project's requirements for user authentication, session management, and database operations.

4.2 Reference Two

Reference: Pallets Projects. (n.d.). *Flask Documentation (Version stable)*. Retrieved October 1, 2025, from https://flask.palletsprojects.com/en/stable/

Justification: The official Flask documentation is a good primary reference to refer to as our team is already comfortable with modern web frameworks, and a member has prior, hands-on experience building apps in Flask. The docs are the most up-to-date source for API behaviour, configuration, blueprints, Jinga templating, request/response lifecycles, and deployment guidance. This makes it far more reliable than blog posts or outdated tutorials and books. Using them allows us to be aligned with Flask's best and most secure practices and error handling. This shortens onboarding, reduces rework, and ensures that all design decisions we make are consistent with the framework's intended pattern.

4.3 Reference Three

Reference: GeeksforGeeks. (2025, August 5). *Flask Tutorial*. Retrieved October 1, 2025, from https://www.geeksforgeeks.org/python/flask-tutorial/

Justification: While other references can be utilized to learn the capabilities of Flask, this reference can be used to learn how to implement specific features. Topics such as defining endpoints, serving static files and handling redirects and errors are taught in a simpler way, with many more guides available to aid in creating a Flask application. GeeksforGeeks is additionally a reputable community-made resource and has been used as references in other university classes for their digestible explanations.

5 Implementation Plan

5.1 Database Design

- Design and normalize relational schema based on ER diagram
- Implement database creation scripts for MySQL/MariaDB/PostgreSQL [4]
- Define foreign key constraints and integrity rules
- Create indexes for optimal query performance

5.2 Backend Development

- Set up Flask [1, 2] application structure with PostgreSQL [4] integration
- Implement user authentication and session management
- Develop RESTful API endpoints for CRUD operations
- Create pathfinding algorithm [8] for route calculation
- Implement business logic for time restrictions and cost calculations

5.3 Frontend Development

- Design responsive UI with HTML5 [5] and CSS3 [5]
- Implement interactive map visualization with JavaScript [5] (with React [7] framework and Three.js [6] library)
- Create forms for location and route management
- Develop dynamic route display with cost and time breakdowns

5.4 Integration and Testing

- Connect frontend to backend API endpoints
- Test database transactions and data integrity
- Validate pathfinding algorithm accuracy
- Perform user acceptance testing

6 Timeline

Milestone	Target Date
Complete ER diagram and relational schema	October 15, 2025
Database scripts and basic Python connection	October 22, 2025
Milestone Demo (Tutorial Week of Oct 27)	October 27-31, 2025
User authentication and location CRUD	November 5, 2025
Route calculation algorithm implementation	November 12, 2025
Team Evaluation submission	November 18, 2025
Frontend interface completion	November 24, 2025
Testing and refinement	November 25-30, 2025
Final Demo (Tutorial Week of Dec 1)	December 1-5, 2025
Individual Reflection submission	December 5, 2025

7 Challenges and Considerations

7.1 Technical Challenges

- Implementing efficient pathfinding algorithms [8] for potentially large graphs
- Managing time-based restrictions in route calculations
- Ensuring data integrity when users create interconnected locations and roads
- Optimizing database [4] queries for complex multi-table joins

7.2 Design Considerations

- Normalization vs. query performance tradeoffs
- User interface design for intuitive location and route creation
- Scalability of the system for large fictional worlds
- Currency conversion logic for multi-currency support

8 Conclusion

The Fictional World Navigation System or **Orbis** represents a comprehensive database application that demonstrates complex entity relationships, transaction management, and search algorithm implementation. By building this system, our team will gain hands-on experience with database design, web development, and the integration of multiple technologies to create a functional, user-friendly application.

9 References

[1] Miguel Grinberg. 2018. *Flask Web Development: Developing Web Applications with Python* (2nd ed.). O'Reilly Media, Inc., Sebastopol, CA.

[2] Pallets Projects. 2025. Flask Documentation. Retrieved October 1, 2025 from https://flask.palletsprojects.com/en/stable/

[3] GeeksforGeeks. 2025. Flask Tutorial. Retrieved October 1, 2025 from https://www.geeksforgeeks.org/python/flask-tutorial/

[4] PostgreSQL Global Development Group. 2025. PostgreSQL 17.0 Documentation. Retrieved October 1, 2025 from https://www.postgresql.org/docs/17/index.html

[5] Mozilla Developer Network. 2025. MDN Web Docs. Retrieved October 1, 2025 from https://developer.mozilla.org/

[6] Three.js. 2025. Three.js Documentation. Retrieved October 1, 2025 from https://threejs.org/docs/

[7] Meta Platforms, Inc. 2025. React Documentation. Retrieved October 1, 2025 from https://react.dev/

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* 4, 2 (July 1968), 100–107. DOI:https://doi.org/10.1109/TSSC.1968.300136

[9] Render Services, Inc. 2025. Render Documentation. Retrieved October 1, 2025 from https://render.com/docs

[10] Socket.IO. 2025. Socket.IO Documentation. Retrieved October 1, 2025 from https://socket.io/docs/v4/