Technical Specifications and Implementation Guides for Walkable MVP

This document provides detailed technical specifications and implementation guides for key components of the Walkable MVP, focusing on the initial phases of development. These guides aim to provide clarity on architectural choices, technology stack considerations, API designs, and best practices for implementation.

Phase 1: User Authentication - Detailed Technical Specification

1.1 Backend User Authentication API Development

1.1.1 User Model Design

Objective: Define the structure of the user data to be stored in the database, ensuring security and efficiency.

Fields:

- `id` : Unique identifier for the user (Primary Key, UUID recommended).
- `username` : Unique string for user identification (e.g., `VARCHAR(50)` , unique, not null).
- `email` : User's email address (e.g., `VARCHAR(255)` , unique, not null, validated format).
- `password_hash` : Hashed and salted password (e.g., `VARCHAR(255)` , not null).
- `created_at` : Timestamp of user creation (e.g., `DATETIME` or `TIMESTAMP` , default to current time).
- `updated_at` : Timestamp of last update (e.g., `DATETIME` or `TIMESTAMP` , updated on modify).

Security Considerations:

- **Password Hashing:** Passwords MUST NOT be stored in plain text. Use a strong, adaptive one-way hashing function like bcrypt or Argon2. Bcrypt is widely supported and recommended for its resistance to brute-force attacks due to its adjustable work factor [1].

- **Salting:** A unique salt MUST be generated for each password before hashing to prevent rainbow table attacks. Bcrypt inherently handles salting.

1.1.2 API Endpoint for User Registration (`POST /api/register`)

Objective: Allow new users to create an account.

Request Body (JSON):

```json
{
  "username": "string",
  "email": "string (email format)",
  "password": "string (min 8 chars, strong password policy recommended)"
}
```

Response (JSON):

- **Success (HTTP 201 Created):**

  ```
  json
  {
    "message": "User registered successfully",
    "user_id": "uuid",
    "token": "jwt_token_string" // Optional, if auto-login after registration
  }
  ```

- **Error (HTTP 400 Bad Request):** Invalid input (e.g., missing fields, invalid email, weak password).

  ```
  json
  {
    "error": "Invalid input",
    "details": "Email already registered."
  }
  ```

- **Error (HTTP 409 Conflict):** Username or email already exists.

  ```
  json
  {
    "error": "Conflict",
    "details": "Username or email already exists."
  }
  ```

Implementation Details:

1. Validate input: Ensure all required fields are present and meet format/strength requirements.
2. Check for existing username/email: Query the database to ensure uniqueness.
3. Hash password: Use bcrypt to hash the provided password.
4. Create user record: Store the new user in the database.
5. Generate JWT (Optional): If auto-login is desired, generate a JWT and return it.

1.1.3 API Endpoint for User Login (`POST /api/login`)

Objective: Authenticate existing users and provide an access token.

Request Body (JSON):

```
{
  "email": "string",
  "password": "string"
}
```

Response (JSON):

- **Success (HTTP 200 OK):**

  ```
  json
  {
    "message": "Login successful",
    "user_id": "uuid",
    "token": "jwt_token_string",
    "expires_in": 3600 // Token expiration in seconds
  }
  ```

- **Error (HTTP 401 Unauthorized):** Invalid credentials.

  ```
  json
  {
    "error": "Unauthorized",
    "details": "Invalid email or password."
  }
  ```

Implementation Details:

1. Retrieve user: Find the user by email.

2. Verify password: Compare the provided password with the stored hash using bcrypt's `check_password_hash` (or equivalent) function.
3. Generate JWT: If credentials are valid, generate a JSON Web Token (JWT) containing user ID and roles (if applicable). Set an expiration time for the token.
4. Return token: Send the JWT back to the client.

1.1.4 API Endpoint for User Logout (`POST /api/logout`)

Objective: Invalidate the user's session/token on the server-side (if using server-side sessions or a JWT blacklist).

Request Headers:

- `Authorization: Bearer <jwt_token>`

Response (JSON):

- **Success (HTTP 200 OK):**

  ```
  json
  {
    "message": "Logout successful"
  }
  ```

- **Error (HTTP 401 Unauthorized):** Invalid or missing token.

Implementation Details:

1. Token validation: Verify the provided JWT.
2. Invalidate token: If using a JWT blacklist, add the token to the blacklist. If using server-side sessions, destroy the session.

1.1.5 Password Hashing and Salting (e.g., bcrypt)

Recommendation: Use a library that implements bcrypt (e.g., `Flask-Bcrypt` for Flask, `passlib` for general Python). The work factor (cost parameter) should be chosen carefully; a higher value increases security but also computation time. Start with a value that takes around 100-300ms on your target server hardware.

Example (Conceptual Python with `bcrypt` library):

```python
import bcrypt

def hash_password(password):
    hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
    return hashed_password.decode('utf-8')
```

```
def check_password(password, hashed_password):
    return bcrypt.checkpw(password.encode('utf-8'),
hashed_password.encode('utf-8'))
```

1.1.6 JWT Token Generation and Validation

Recommendation: Use a robust JWT library (e.g., `PyJWT` for Python).

Structure: A JWT consists of three parts separated by dots: Header, Payload, and Signature.

- **Header:** Typically specifies the token type (JWT) and the signing algorithm (e.g., HS256).
- **Payload:** Contains claims about the entity (e.g., user ID, roles) and additional data. Standard claims like `exp` (expiration time) and `iat` (issued at time) are crucial.
- **Signature:** Used to verify the token hasn't been tampered with and is signed by the server.

Example (Conceptual Python with `PyJWT` library):

```
 import jwt
import datetime

SECRET_KEY = "your_super_secret_key_here" # MUST be stored securely (environment
variable)
ALGORITHM = "HS256"

def generate_jwt(user_id, expires_in_seconds=3600):
    payload = {
        "user_id": str(user_id),
        "exp": datetime.datetime.utcnow() +
datetime.timedelta(seconds=expires_in_seconds),
        "iat": datetime.datetime.utcnow()
    }
    token = jwt.encode(payload, SECRET_KEY, algorithm=ALGORITHM)
    return token

def decode_jwt(token):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        return payload
    except jwt.ExpiredSignatureError:
        return {"error": "Token expired"}
    except jwt.InvalidTokenError:
        return {"error": "Invalid token"}
```

Best Practices:

- **Secret Key:** The `SECRET_KEY` MUST be a strong, randomly generated string and stored securely (e.g., environment variable, not hardcoded).
- **Expiration:** Always set an expiration time (`exp`) for JWTs to limit the window of opportunity for token compromise.
- **Refresh Tokens:** For better user experience and security, consider implementing refresh tokens. Short-lived access tokens are used for API requests, and a longer-lived refresh token is used to obtain new access tokens without re-authenticating the user frequently [2].
- **HTTPS:** All API communication, especially authentication, MUST occur over HTTPS to prevent eavesdropping.

1.1.7 Unit Tests for Backend Authentication API

Objective: Ensure each API endpoint and utility function works as expected in isolation.

Test Cases (Examples):

- **Registration:**
  - Successful registration with valid data.
  - Registration with existing email/username.
  - Registration with invalid email format.
  - Registration with weak password.
  - Missing required fields.
- **Login:**
  - Successful login with correct credentials.
  - Login with incorrect password.
  - Login with non-existent email.
  - Missing required fields.
- **Logout:**
  - Successful logout with valid token.
  - Logout with invalid/expired token.
- **Password Hashing:**
  - Verify that `hash_password` produces different hashes for the same password with different salts.
  - Verify that `check_password` correctly validates passwords.
- **JWT:**
  - Verify token generation and decoding.
  - Test token expiration.
  - Test invalid token signature.

1.2 Frontend User Authentication UI Development

1.2.1 User Registration Form

Components: Input fields for username, email, password, and confirm password. A submit button.

Client-side Validation:

- **Required Fields:** All fields are mandatory.
- **Email Format:** Validate using a regular expression.
- **Password Strength:** Minimum length (e.g., 8 characters), presence of uppercase, lowercase, numbers, and special characters. Provide real-time feedback to the user.
- **Password Confirmation:** Ensure password and confirm password fields match.

1.2.2 User Login Form

Components: Input fields for email/username and password. A submit button. Links for

"Forgot Password?" and "Sign Up".

Client-side Validation:

- **Required Fields:** Email/username and password are mandatory.

1.2.3 Integration with Backend API (Registration)

Process Flow:

1. User fills out the registration form and submits.
2. Frontend performs client-side validation.
3. If validation passes, send a `POST` request to `/api/register` with the form data.
4. Handle success response (e.g., redirect to login, show success message).
5. Handle error responses (e.g., display error messages for existing user, invalid input).

1.2.4 Integration with Backend API (Login)

Process Flow:

1. User fills out the login form and submits.
2. Frontend performs client-side validation.
3. If validation passes, send a `POST` request to `/api/login` with the form data.

4. Handle success response:
    - Store the received JWT (or session token) securely (e.g., in `localStorage` or `sessionStorage` for JWT, or rely on cookies for session management). `localStorage` is generally preferred for JWTs for persistence across browser sessions, but be aware of XSS vulnerabilities [3].
    - Redirect the user to a protected route (e.g., the Discover page).
5. Handle error responses (e.g., display error messages for invalid credentials).

1.2.5 Client-side Validation Best Practices

- **Immediate Feedback:** Provide real-time feedback to users as they type (e.g., green checkmark for valid input, red X for invalid).
- **Clear Error Messages:** Error messages should be specific and actionable (e.g.,

"Email format is invalid" instead of "Invalid input").
* **Prevent Submission:** Disable the submit button until all client-side validations pass.

1.3 Auth-aware Navbar Implementation

Objective: Dynamically change the navigation bar content based on the user's authentication status.

Implementation Details:

1. **Authentication State Management:** The frontend application needs a mechanism to track the user's authentication state. This can be achieved using:
    - **Context API (React), Vuex (Vue), Redux (React/Redux):** For larger applications, a centralized state management solution is recommended.
    - **Simple State Variables:** For smaller applications, local state variables or React Hooks (`useState` , `useContext`) can suffice.
2. **Conditional Rendering:**
    - If the user is authenticated (i.e., a valid JWT is present in `localStorage` or a session exists), display links like

"Profile" and "Logout".
  * If the user is not authenticated, display links like "Login" and "Sign Up".
3. **Logout Functionality:**
  * When the "Logout" link is clicked, clear the JWT from `localStorage` (or invalidate the session on the backend if applicable).
  * Redirect the user to the login page or home page.

1.4 Integration Testing (Phase 1)

Objective: Verify that all components of the authentication system work together seamlessly.

Testing Tools/Frameworks:

- **Frontend:** Jest, React Testing Library, Cypress (for end-to-end).
- **Backend:** Pytest, unittest (Python).

Key Scenarios to Test:

- **Full Registration Flow:**
  - User navigates to registration page.
  - Fills in valid details.
  - Submits form.
  - Verifies success message/redirection.
  - Attempts to log in with newly created credentials.
- **Full Login Flow:**
  - User navigates to login page.
  - Enters valid credentials.
  - Submits form.
  - Verifies successful login (e.g., sees profile link, redirected to dashboard).
  - Refreshes page to ensure authentication persistence.
- **Full Logout Flow:**
  - User logs in.
  - Clicks logout button.
  - Verifies logout (e.g., sees login/signup links, redirected to home/login).
  - Attempts to access protected routes (should be denied).
- **Edge Cases:**
  - Incorrect password attempts.
  - Attempting to register with an already existing email/username.
  - Navigating directly to protected routes without authentication.

Phase 2: Discover Page + Map - Detailed Technical Specification

2.1 Map Integration

Objective: Integrate Leaflet.js to display an interactive map.

Technology Choice: Leaflet.js is an open-source JavaScript library for mobile-friendly interactive maps. It is lightweight, has a simple API, and a large plugin ecosystem [4].

Implementation Details:

1. **Installation:** Include Leaflet via CDN or npm package. `html <!-- Example using CDN --> <link rel="stylesheet" href="https://unpkg.com/leaflet@1.7.1/dist/leaflet.css" /> <script src="https://unpkg.com/leaflet@1.7.1/dist/leaflet.js"></script>` `bash # Example using npm npm install leaflet`
2. **Map Initialization:** Create a `div` element in your HTML with a specific ID and set its height. `html <div id="mapid" style="height: 500px;"></div>` Initialize the map in your JavaScript: `javascript var mymap = L.map("mapid").setView([51.505, -0.09], 13); // Centered at London, zoom level 13`
3. **Tile Layer:** Add a tile layer to display the map tiles. OpenStreetMap is a common choice. `javascript L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png", { attribution: "&copy; <a href=\"https://www.openstreetmap.org/copyright\">OpenStreetMap</a> contributors" }).addTo(mymap);`

2.2 Geolocation Implementation

Objective: Obtain the user's current location using the browser's Geolocation API and display it on the map.

Technology: Browser's built-in `navigator.geolocation` API.

Implementation Details:

1. **Requesting Location:** Use `navigator.geolocation.getCurrentPosition()` to get the user's current position. `javascript if ("geolocation" in navigator) { navigator.geolocation.getCurrentPosition(function(position) { var lat = position.coords.latitude; var lon = position.coords.longitude; // Use lat, lon }, function(error) { // Handle errors console.error("Error getting geolocation: ", error); }); } else { console.log("Geolocation is not supported by this browser."); }`
2. **Displaying User Marker:** Create a Leaflet marker at the obtained coordinates. `javascript var userLocationMarker = L.marker([lat, lon]).addTo(mymap) .bindPopup("You are here!").openPopup(); mymap.setView([lat, lon], 15); // Center map on user location`
3. **Error Handling:** Gracefully handle cases where the user denies permission or geolocation is unavailable.
    - `PERMISSION_DENIED (1)`: User denied access to location.
    - `POSITION_UNAVAILABLE (2)`: Network error or satellites unavailable.
    - `TIMEOUT (3)`: The request timed out.

2.3 Tour Data Display on Map

Objective: Fetch tour data from the backend and display tour pins on the map.

2.3.1 Backend API Endpoint for Nearby Tours (`GET /api/tours/nearby?lat=&lon=&radius=`)

Objective: Provide a list of tours within a specified radius of given coordinates.

Parameters:

- `lat` : Latitude (required, float).
- `lon` : Longitude (required, float).
- `radius` : Search radius in kilometers (optional, float, default e.g., 10km).

Response (JSON):

- **Success (HTTP 200 OK):**

  `json`
  ```
  [
    {
      "id": "uuid",
      "title": "string",
      "description": "string",
      "latitude": "float",
      "longitude": "float",
      "category": "string",
      "creator_name": "string"
    },
    // ... more tours
  ]
  ```

Implementation Details:

1. **Database Query:** Use spatial queries (if your database supports it, e.g., PostGIS for PostgreSQL) or calculate distances manually to filter tours within the radius.
2. **Indexing:** Ensure geographic coordinates are indexed for efficient querying.

2.3.2 Frontend Logic to Call Nearby Tours API

Process Flow:

1. Once the user's location is obtained (from Task 2.2), construct the API URL with `lat`, `lon`, and `radius`.
2. Make an asynchronous `GET` request to `/api/tours/nearby`.
3. Handle the response, parsing the JSON data.

2.3.3 Display Tour Pins/Markers on Leaflet Map

Process Flow:

1. Iterate through the fetched tour data.
2. For each tour, create a Leaflet marker at `[tour.latitude, tour.longitude]`.
3. Add the marker to the map.

2.3.4 Basic Popup/Tooltip for Tour Pins

Implementation Details:

1. Use `bindPopup()` method on each marker to attach a popup. `javascript L.marker([tour.latitude, tour.longitude]).addTo(mymap) .bindPopup("<b>" + tour.title + "</b><br>" + tour.description.substring(0, 50) + "...");`
2. Consider adding a link within the popup to navigate to the tour detail page.

2.4 Integration Testing (Phase 2)

Objective: Verify that the map, geolocation, and tour display functionalities work together correctly.

Key Scenarios to Test:

- **Map Loading:** Verify that the Leaflet map initializes and displays correctly upon page load.
- **Geolocation Accuracy:**
  - Test with user granting permission: Verify user's marker appears at the correct location.
  - Test with user denying permission: Verify appropriate fallback or error message is displayed.
  - Test with simulated locations (if possible in development environment).
- **Nearby Tour Display:**
  - Verify that tours within the specified radius are fetched and displayed as pins.
  - Verify that tours outside the radius are not displayed.

    - Test with varying numbers of tours (none, few, many).
- **Tour Pin Interaction:**
    - Click on a tour pin: Verify that the popup appears with the correct tour title and description snippet.
    - Clicking outside the popup closes it.

References

[1] OWASP Foundation. (n.d.). Password Storage Cheat Sheet. Retrieved from https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
[2] Auth0. (n.d.). Refresh Tokens: When to Use Them and How They Work. Retrieved from https://auth0.com/docs/secure/tokens/refresh-tokens
[3] OWASP Foundation. (n.d.). Cross-Site Scripting (XSS). Retrieved from https://owasp.org/www-community/attacks/xss/
[4] Leaflet. (n.d.). About. Retrieved from https://leafletjs.com/about.html