**National Park Meetup**

**System Design Document**

November 28, 2024

**THIS PAGE INTENTIONALLY BLANK**

**Table of Contents**

[**National Park Meetup** 1](#_Toc183644421)

[**1. Introduction** 4](#_Toc183644423)

[**2. System Architecture** 4](#_Toc183644424)

[**3. System Component Diagram** 5](#_Toc183644425)

[**4. Data Model** 6](#_Toc183644427)

[**5. API Design** 7](#_Toc183644428)

[**6. Scalability and Performance Considerations** 9](#_Toc183644429)

[**7. Security Considerations** 9](#_Toc183644430)

[**8. Testing Frameworks** 9](#_Toc183644430)

[**9. Conclusion** 11](#_Toc183644430)

**1. Introduction**

The National Park Meetup application is designed to allow users to connect based on their proximity to national parks. Users can create accounts, share their locations, and find others nearby within a specified radius. The application uses modern tools and frameworks, with a focus on scalability, data consistency, and performance.

The System Architecture section of this document describes an optimized technology stack designed to ensure scalability. However, due to time constraints, the actual implementation may differ. For instance, the prototype uses an H2 database instead of PostgreSQL. Additionally, a service layer module, ScheduledUserLocationTask, was implemented to simulate user location event messaging in place of Kafka. Finally, a caching layer, such as Redis, has not been included in the current implementation.

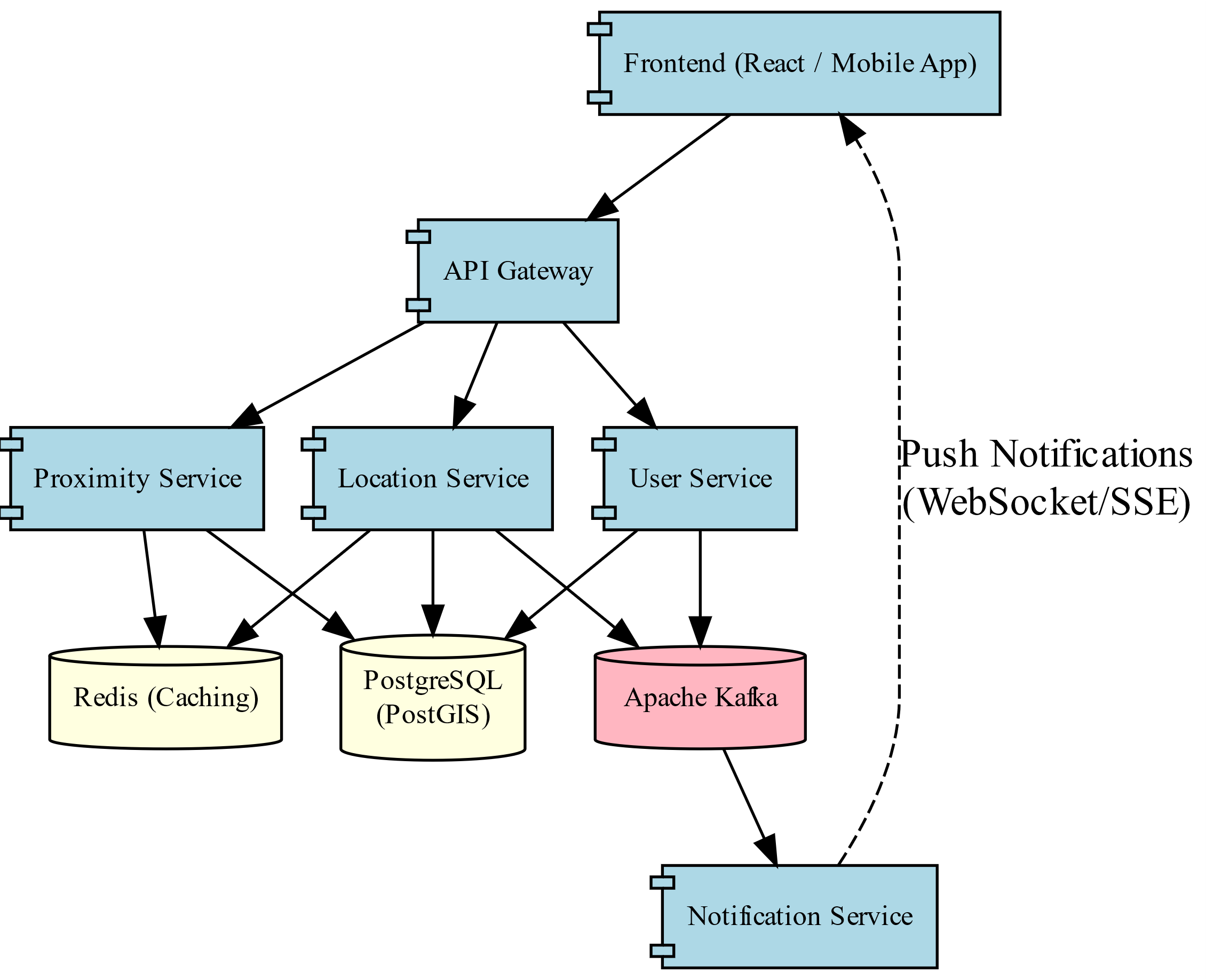
**2. System Architecture**

The system is based on a microservicesarchitecture, leveraging the following technologies:

* **Spring Boot:** For building the backend services.
* **GraphQL:** For flexible querying and mutations of user and location data.
* **PostgreSQL with PostGIS:** For geospatial data storage and spatial queries.
* **Kafka:** For event-driven communication between microservices (e.g., user location updates).
* **Redis:** As a caching layer for frequently accessed data (e.g., static national park locations).
* **Docker and Kubernetes:** For containerization and orchestration, enabling scalable deployments.

**3. System Component Diagram**

**Figure 1 – System Core Components**



The system consists of the following core components:

* **Frontend**: React app or mobile application (not described here but could use RESTful APIs or GraphQL for communication).
* **User Service**: Manages user profiles, mobile numbers, and authentication.
* **Location Service**: Handles the storage and retrieval of user location and park locations.
* **Proximity Service**:Locates nearby users within a radius (in kilometers).
* **Notification Service**: Pushes notifications about nearby users via Kafka or WebSockets/SSE.
* **Database**: PostgreSQL with PostGIS for geospatial data and caching layer via Redis.
* **Kafka**: For streaming location change events to ensure real-time updates.

**4. Data Model**

The application uses the following data models:

**4.1 User**

@Entity

@Table(name = "users")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "user\_id")

private Long id; // Primary key for this table

@Column(name = "first\_name", nullable = false)

@NotEmpty(message = "First name is required")

private String firstName;

@Column(name = "last\_name", nullable = false)

@NotEmpty(message = "Last name is required")

private String lastName;

@Column(name = "password", nullable = false)

@Length(min = 5, message = "Your password must have at least 5 chars")

@NotEmpty(message = "\*Please provide your password")

private String password;

@Column(name = "email", unique = true, nullable = false)

@Email(message = "\*Please provide a valid Email")

@NotEmpty(message = "\*Please provide an email")

private String email;

@Column(name = "mobile\_number", nullable = false)

@NotEmpty(message = "Mobile number is required")

private String mobileNumber;

@Column(name = "status", nullable = false, length = 20)

@Enumerated(EnumType.STRING)

private UserStatus userStatus;

@Column(name = "created\_at", nullable = false, updatable = false)

private ZonedDateTime createdAt;

@Column(name = "updated\_at")

private ZonedDateTime updatedAt;

**4.2 Location**

@Entity

@Table(name = "location")

public class Location {

@Id

@Column(name = "location\_id")

private String id; // Primary key for this table

@Column(name = "park\_name")

private String parkName; // National park name

@Column(name = "state")

private String state;

@Column(name = "latitude")

private Double latitude;

@Column(name = "longitude")

private Double longitude;

**4.3 UserLocation**

@Entity

@Table(name = "user\_location")

public class UserLocation {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name = "user\_location\_id")

private Long id; // Primary key for this table

@ManyToOne

@JoinColumn(name = "user\_id",

referencedColumnName = "user\_id", nullable = false)

private User user;

@ManyToOne

@JoinColumn(name = "location\_id",

referencedColumnName = "location\_id", nullable = false)

private Location location;

@Column(name = "updated\_at")

private ZonedDateTime updatedAt;

**5. API Design**

The application exposes the following REST API endpoints for interacting with user data and locations.

**5.1** **Authentication API**

* **URL:** http://localhost:8070/auth/token
* **POST:** Generate REST API access token via JWT utility.
  + Request: {"username": "prototypeUser"}
  + Response: { eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJwcm90b3R5cGVVc2VyIiwiaWF0IjoxNzMyNzczMTAwLCJleHAiOjE3MzI4NTk1MDB9.uO33wV\_kFP95M6sib\_oLGILvEAv\_GCIVoveBQGosEXQ}
  1. **User API**
* **URL:** http://localhost:8070/api/users
* **POST:** Create a new user.
  + Request: {"firstName": "John", "lastName": "Doe", "password": "changeitpwd", "email": "john@mail.com","mobileNumber": "999-999-9999", "userStatus": "ACTIVE"}
  + Response: {"id": 31, "firstName": "John", "lastName": "Doe", "email": "john@mail.com", "mobileNumber": "9999999999", "userStatus": "ACTIVE", "createdAt": "2024-11-27T21:21:17.9957977-08:00", "updatedAt": "2024-11-27T21:21:17.9957977-08:00"}
* **URL:** http://localhost:8070/api/users/{mobileNumber}
* **GET:** Retrieves user details by mobile number.
  + Response: {"id": 31, "firstName": "John", "lastName": "Doe", "email": "john@mail.com", "mobileNumber": "9999999999", "userStatus": "ACTIVE", "createdAt": "2024-11-27T21:21:17.995798-08:00", "updatedAt": "2024-11-27T21:21:17.995798-08:00"}
  1. **User Location API**
* **URL:** http://localhost:8070/api/location/current
* **GET:** Get a user’s current location
  + Request: {"mobileNumber": “9999999999”}
  + Response: {"id": 30, "user": {"id": 30, "firstName": "John", "lastName": "Doe", "email": "john@mail.com", "mobileNumber": "9999999999","userStatus": "ACTIVE", "createdAt": "2024-11-27T21:16:51.966683-08:00", "updatedAt": "2024-11-27T21:16:51.966683-08:00"}, "location": {"id": "BADL", "parkName": "Badlands", "state": "South Dakota", "latitude": 43.7480316, "longitude": -102.4983748},"updatedAt": "2024-11-27T21:21:48.371166-08:00"}

**5.4 Proximity API**

* **URL:** http://localhost:8070/api/proximity
* **GET:** Get nearby users within a radius (in kilometers).
  + Request: {"mobileNumber": “9999999999”}
  + Response: [{"id": 21,"user": {"id": 21, "firstName": "Marie", "lastName": "Montana", "email": "montana@mail.com", "mobileNumber": "5555555521", "userStatus": "ACTIVE", "createdAt": "2024-11-27T21:16:51.965682-08:00", "updatedAt": "2024-11-27T21:16:51.965682-08:00"}, "location": {"id": "BADL", "parkName": "Badlands","state": "South Dakota", "latitude": 43.7480316, "longitude": -102.4983748}, "updatedAt": "2024-11-27T21:16:52.360279-08:00"}]

**6. Scalability and Performance Considerations**

To handle high traffic and ensure optimal performance, the following measures will be implemented:

* **Database Scalability**: PostgreSQL with PostGIS supports geospatial indexing for fast queries. Sharding or replication may be considered for horizontal scaling if needed.
* **Caching**: Redis will be used to cache frequently accessed data, such as national park locations.
* **Event-Driven Architecture**: Kafka will handle asynchronous communication and real-time updates for user location changes and notifications.
* **Horizontal Scaling**: Docker and Kubernetes will be used to containerize the application and ensure it can scale horizontally on cloud infrastructure.
* **WebSockets / SSE for Notifications**: For real-time location-based notifications to users.

**7. Security Considerations**

* **Authentication**: The app will use OAuth2 or JWT tokens for secure authentication.
* **Data Validation**: Input data will be validated using Spring’s validation framework to prevent injection attacks and ensure data integrity.
* **Data Encryption**: Sensitive data, such as passwords, will be hashed (e.g., using bcrypt) and stored securely in the database.
* **Access Control**: Role-based access control (RBAC) will be enforced on sensitive endpoints.

**8. Testing Frameworks**

To ensure the reliability and robustness of the application, a combination of testing frameworks is used to validate functionality, integration, and performance.

**8.1 Unit Testing**

* **Framework Used:** JUnit 5  
  JUnit is used for writing and executing unit tests to validate the correctness of individual components and methods within the application.
* **Example:** Testing the LocationServiceImpl to ensure location retrieval functions correctly.

**Sample Test Case:**

@ExtendWith(SpringExtension.class)

@SpringBootTest

public class LocationServiceImplTest {

@Autowired

private LocationService locationService;

@Test

public void testFindCurrentUserLocation\_UserLocationExists() {

// Arrange

String mobileNumber = "5555555530";

User mockUser = new User(30L, "Francisco", "Test", "changeitpwd", "francisco@mail.com", "5555555530", UserStatus.ACTIVE);

Location mockLocation = new Location("PINN", "Pinnacles", "California", 36.4808829, -121.1593104);

UserLocation mockUserLocation = new UserLocation(30L, mockUser, mockLocation);

// Mock user location respoitory findCurrentUserLocation response with mockUserLocation object

when(userLocationRepository.findCurrentUserLocation(mobileNumber)).thenReturn(Optional.of(mockUserLocation));

// Act

Optional<UserLocation> foundUserLocation = null;

try {

foundUserLocation = locationService.findCurrentUserLocation(mobileNumber);

}

catch (DataStoreException e) {

LogUtil.testLogger(TAG, e.getMessage());

fail("Expected no exception, but got: " + e.getMessage());

}

// Assert

assertTrue(foundUserLocation.isPresent());

assertEquals("Francisco", foundUserLocation.get().getUser().getFirstName());

assertEquals("PINN", foundUserLocation.get().getLocation().getId());

// Verify interaction

verify(userLocationRepository, times(1)).findCurrentUserLocation(mobileNumber);

}

* 1. **Mocking**
* **Framework Used:** Mockito  
  Mockito is used for mocking dependencies to isolate components during testing. This ensures the tests are focused on the unit under test without relying on external systems like databases or APIs.
* **Example:** Mocking the LocationRepository to test LocationServiceImpl independently as shown in JUnit test above.

**8.3 Integration Testing**

* **Framework Used:** JUnit + [Spring Boot Test](https://docs.spring.io/spring-boot/docs/current/reference/html/features.html#features.testing)  
  Integration tests validate the interaction between multiple components of the application, such as REST controllers, services, and repositories.
* **Example:** Testing the LocationServiceFunctionalTest to ensure location retrieval functions correctly by directly accessing the database service and repository.

@Test

public void testFindCurrentLocation() {

// Arrange

String mobileNumber = "5555555530";

// Act

Optional<UserLocation> foundUserLocation = null;

try {

foundUserLocation = locationService.findCurrentUserLocation(mobileNumber);

}

catch (DataStoreException e) {

LogUtil.testLogger(TAG, e.getMessage());

fail("Expected no exception, but got: " + e.getMessage());

}

// Assert

assertTrue(foundUserLocation.isPresent());

assertEquals("Francisco", foundUserLocation.get().getUser().getFirstName());

assertEquals("PINN", foundUserLocation.get().getLocation().getId());

}

**9. Conclusion**

This design outlines the core architecture of the National Park Meetup application, focusing on scalability, performance, and simplicity. Key components like PostgreSQL with PostGIS, Redis, and Kafka ensure that the application can handle growth and efficiently deliver location-based services to users.