UNIVERSITY OF PISA

Master's Degree in Artificial Intelligence and Data Engineering

Project for Large-Scale and Multi-Structured Databases

Smart Parking Finder System (SPFS)

Professors:
Prof. Pietro Ducange
Ing. Alessio Schiavo

Students:
Mohsin Ali (721983)
Chandrakant Yadav (720360)
Pedro Carneiro Junior (724961)

ACADEMIC YEAR 2025/2026

Contents

1 Introduction

The Smart Parking Finder System (SPFS) is a modern, scalable two-sided platform designed to alleviate urban congestion by providing real-time information on parking slot availability in the country of United States. The Mobile User Interface serves the end-user, enabling them to quickly search for, view details of, and reserve an available parking slot for a duration not exceeding 8 hours. The Administrator Dashboard provides the back-office tools for managing lot metadata (CRUD operations) and analyzing system-wide usage metrics.

More details regarding the application architecture are available in the following chapter. This kind of application needs to handle a large amount of data guaranteeing quick response times needed in the modern era of mobile applications, for this reason MongoDB and Redis NoSQL databases were chosen.

GitHub repo of back-end is available on GitHub:

https://github.com/19mohsin58/Smart-Parking-Finding-System

1.1 Key Features

- **Real-Time Availability:** Instant updates on vacant slots using high-performance NoSQL databases.
- **Mobile User Interface:**
  - Search for parking lots based on location.
  - View detailed information (Address, Parking Name, capacity).
  - Reserve slots for up to 8 hours.
- **Administrator Dashboard:**
  - **Metadata Management (CRUD):** View, add, update, or remove parking lot information.
  - **Usage Metrics:** Analyze system-wide data to understand peak hours and high-demand areas, the applied analytics will help the admin to apply the parking rate according to the demand in future.

2 Data Sets

2.1 Data Sources and Acquisition

The final dataset is a consolidated view derived from disparate, real-world public data sources to ensure geographical and informational diversity. This approach was necessary to create a comprehensive and non-uniform testing environment for the SPFS.

The raw data was collected from three primary public sources, providing geographical diversity and key parking facility attributes:

Source File Name	Source URL	Data Focus
Kaggle: US Parking Data	https://www.kaggle.com/datasets/mfaisalqureshi/parking	General US parking lot capacity and location data.
Kaggle: Parking Lots and Garages	https://data.ca.gov/dataset/parking-lots-and-garage	Specific facility data including precise names and addresses in major metropolitan areas.
Data.Gov Datasets	https://catalog.data.gov/dataset/?tags=parking-lot	Used for comprehensive US government and municipal data extraction to supplement core information.

2.2 Data Refactoring and Standardization Process

The raw data was disparate, contained inconsistencies (e.g., varying column names, null capacity values), and lacked the necessary links for a modern database structure. The refactoring process involved a two-stage Python scripting pipeline that focused on cleansing, standardizing, and guaranteeing data integrity.

2.2.1 Cleansing and Core Data Assembly

This initial stage focused on cleaning the raw files and consolidating the foundational parking facility information.

Refactoring Step	Description	Outcome
Data Cleansing	Converted stall/capacity fields to numeric types; coerced errors; removed all records with zero or null capacity values.	Ensured all facilities are valid parking locations.
Field Standardization	Renamed all varying source columns (e.g., DEA_FACILITY_NAME, Bldg City) to standardized project field names (parkingName, city, totalCapacity).	Created a unified, project-specific data language.
Granular Slot Generation	Used the totalCapacity field of each parking lot to synthetically generate individual, unique records for every single parking spot.	Created the most atomic level of parking data required for real-time tracking.

2.3 Python Scripts for Final Data Generation

The following two scripts utilize the bson.objectid.ObjectId class for all primary keys, ensuring a performant and natively compatible MongoDB dataset.

2.3.1 Script 1: Base Collections Generation (Cities, Lots, Slots)

Purpose: Cleans and structures the raw source data into the foundation collections (Cities, ParkingLots, Slots), replacing all primary keys with MongoDB ObjectIDs.

Github Link: https://github.com/19mohsin58/Smart-Parking-Finding-System/blob/main/Python%20Script%20for%20Dataset/cleandAndStructureRawData.py

2.3.2 Script 2: Synthetic Data Generation (Users, Reservations)

Purpose: Generates synthetic Users and Reservations, enforcing strict referential integrity by linking the slotId to the MongoDB ObjectID embedded within the parent ParkingLot document's slotIds array.

Github Link: https://github.com/19mohsin58/Smart-Parking-Finding-System/blob/main/Python%20Script%20for%20Dataset/UsersAndReservationGenerations.py

2.4 The Final Dataset

The dataset consists of:

- 1.8K Cities
- 6.9K Parking Lots
- 567K Slots
- 50K Reservation
- 20K Users

Total Size of Dataset 51 MBs

3 System Requirements and Design

3.1 Functional Requirements

3.1.1 User Module

- **Account Security**: Users register and log in securely via email-based OTP verification.
- **User's City Parking Lots**: User Should see the parking lots on main page right after he logins to the system as per the city he chooses while registration.
- **Parking Search**: Users browse for parking using an optimized state-to-city geographic hierarchy for other cities.
- **Live Availability**: Users view real-time slot counts fetched directly from Redis in-memory sets.
- **Instant Booking**: The system provides atomic, immediate slot assignment upon user request.
- **Single Reservation Limit**: The system prevents double-booking by restricting users to one active reservation at a time.
- **Automatic Release**: Reservations expire and return to the available pool automatically after 8 hours.
- **Booking Cancellation**: Users can manually release their spot instantly, updating the live pool for other drivers.
- **Profile Update**: User should able to update the Name, email and also forgot the password.

3.1.2 Admin Module

- **Dynamic Management**: Admins add lots that auto-sync slots to Redis & MongoDB by uniquely assigning the slot numbers and auto-create missing City documents in MongoDB.
- **Operational Constraint**: The system blocks parking lot creation if Redis is offline to ensure live pool integrity.

- **Advanced Analytics**: Admins access insights on peak usage, duration, and user loyalty rankings.
- **View Parking Lots**: Admin should be able to see the parking lots with available slots also filter by cities by selecting state and respective city.

3.2 Non-Functional Requirements

1. The system must use both a DocumentDB and a Key Value DB

2. The system must expose its functionalities through a complete RESTful API

3. The system must store user's passwords using a secure hash

4. The system should avoid losses to stored data and ensure fault tolerance

5. The system should provide high availability for read operations with low latency responses and Strong Consistency for write operations.

3.3. System Design

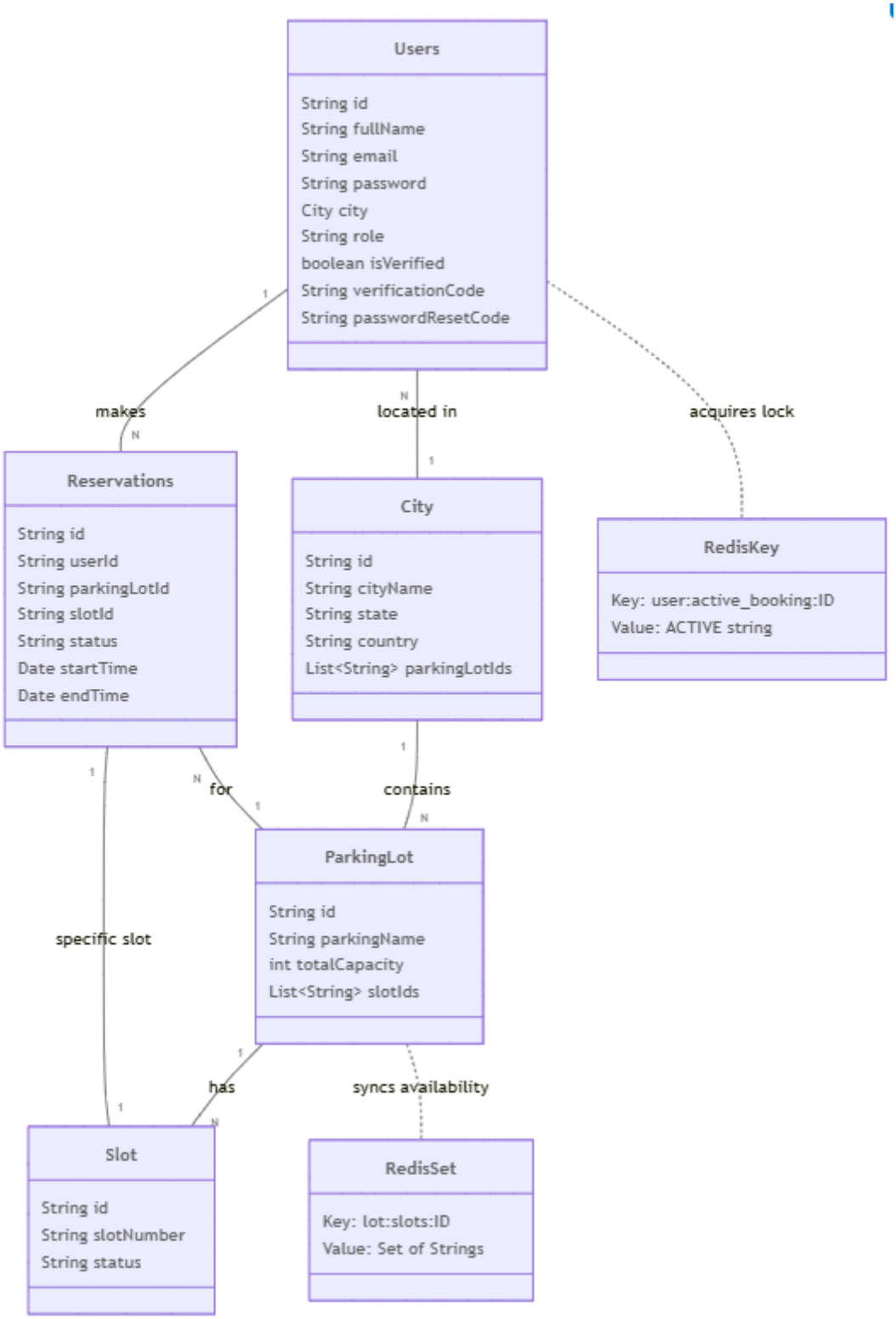

Figure 1: UML Class Diagram – Domain Model

3.4. UI Mockups

3.4.1 Actor User.

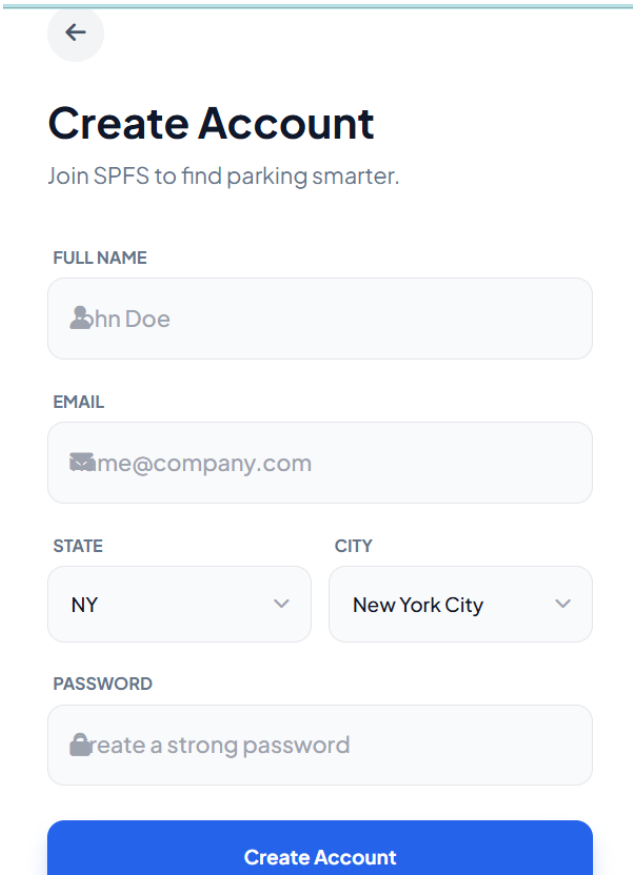

Figure 2: Mobile user account registration.

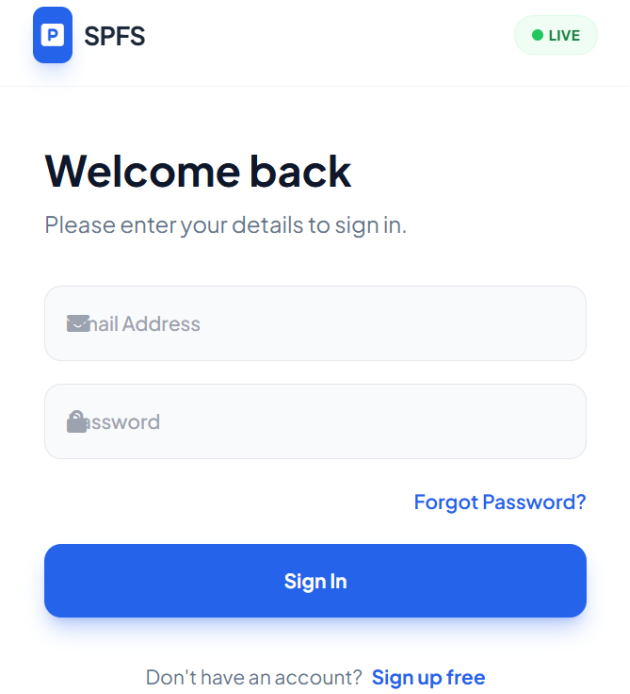

Figure 3: Mobile user login (mockup).

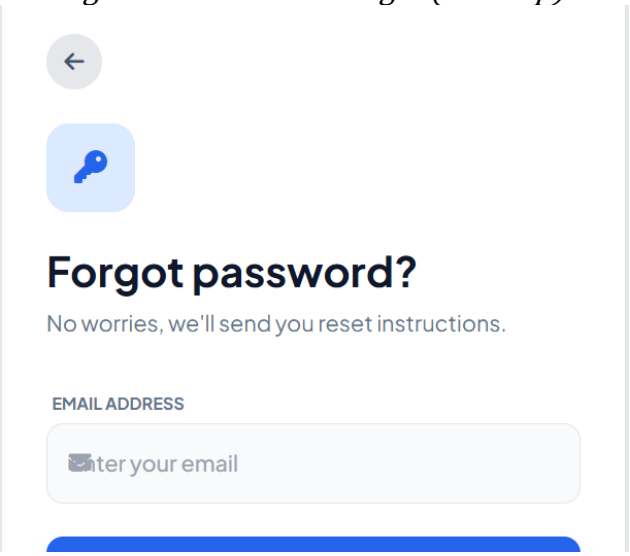

Figure 4: Forgot Password Screen.

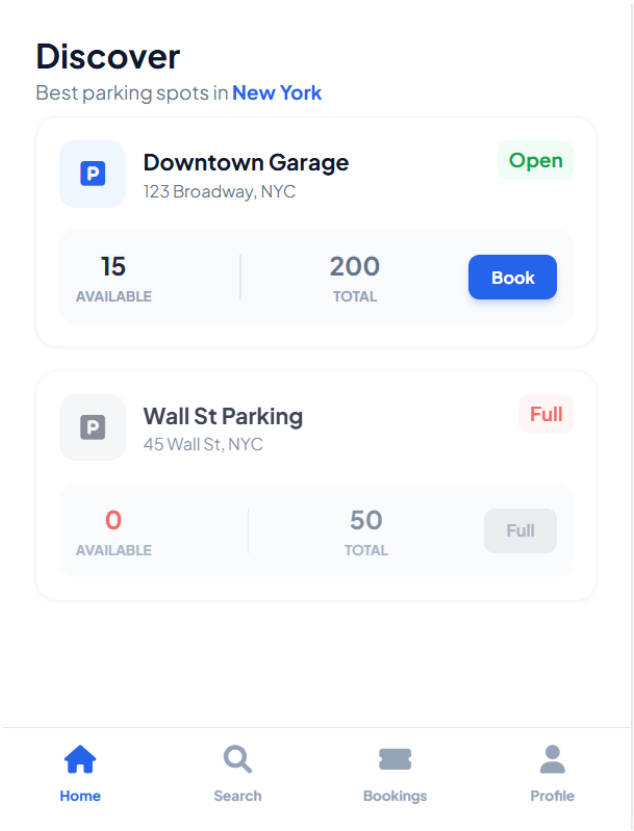

Figure 5: Mobile user main screen.

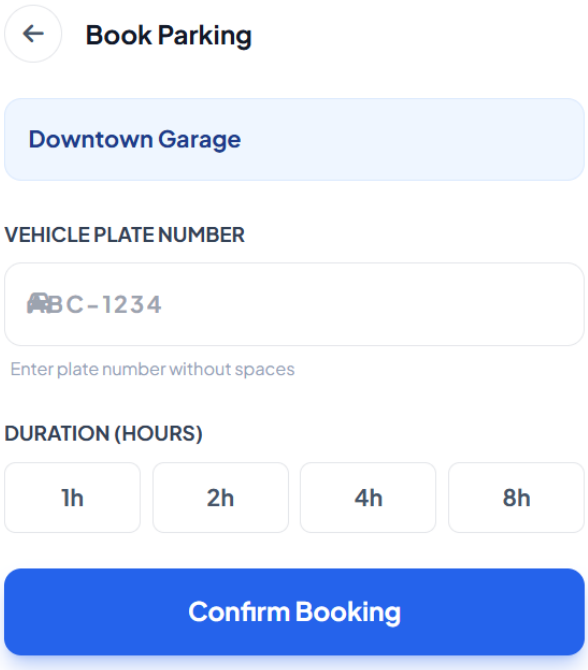

Figure 6: Mobile user real time reservation (mockup).

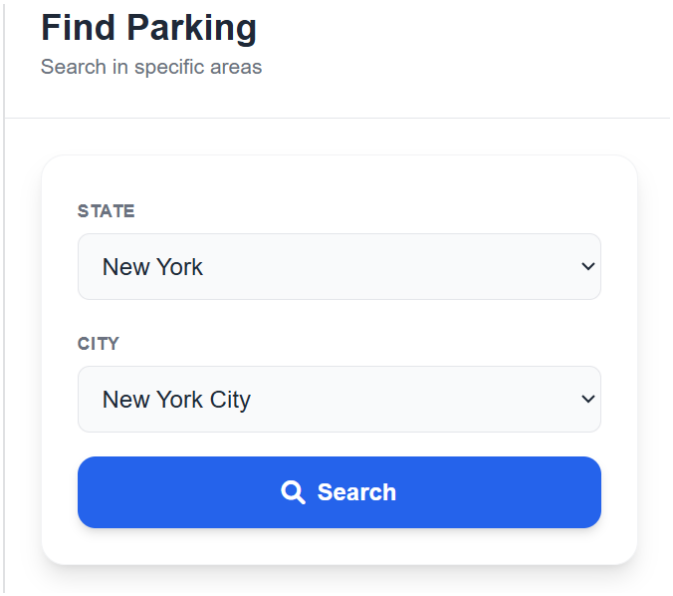

Figure 7: Search for the Parking in other Cities

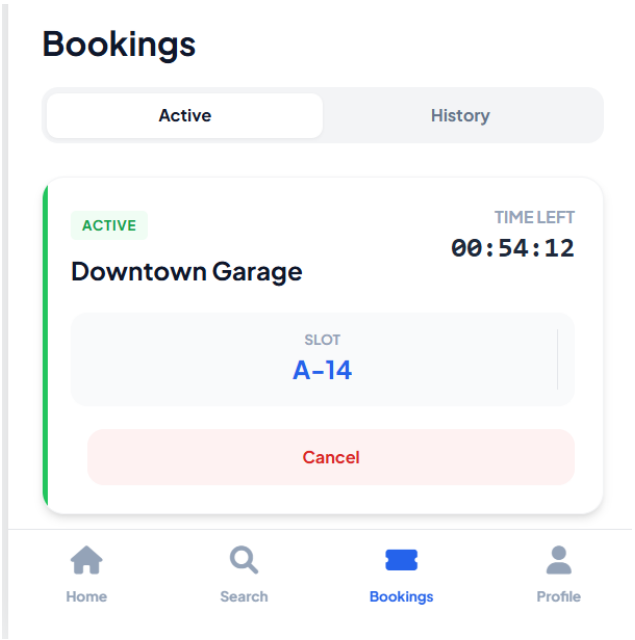

Figure 8: Mobile user real time current reservation information (mockup).

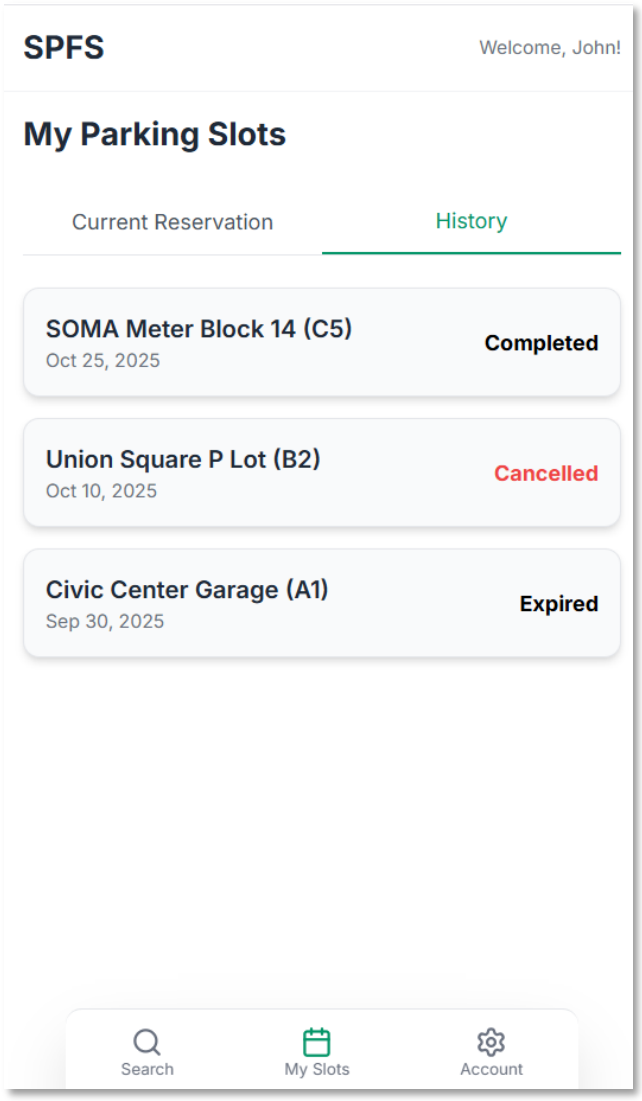

Figure 9: Mobile user reservation history (mockup).

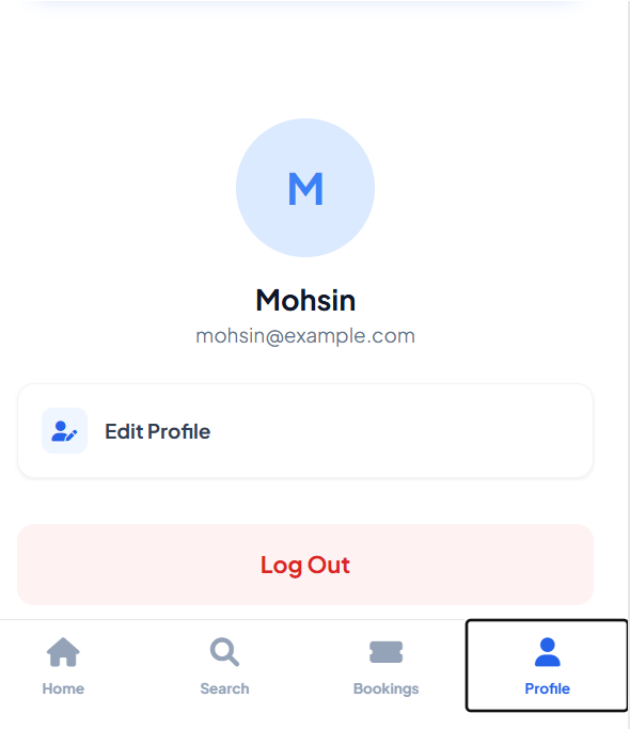

Figure 10: Update Profile screen (mockup)

3.4.2. Actor Admin

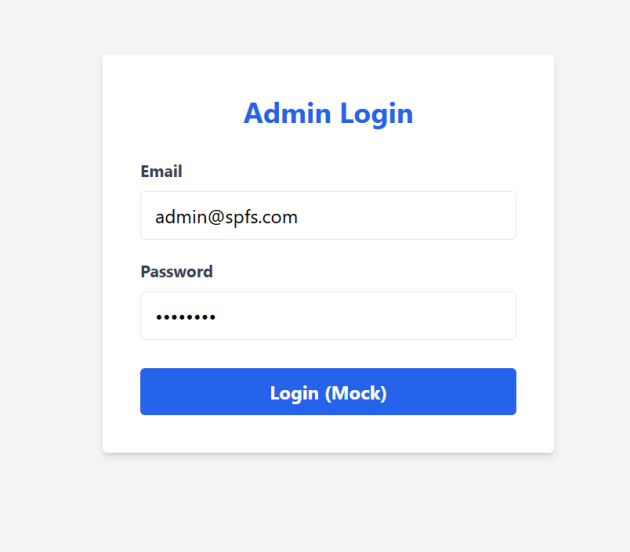

Figure 11: Admin Login screen (mockup)

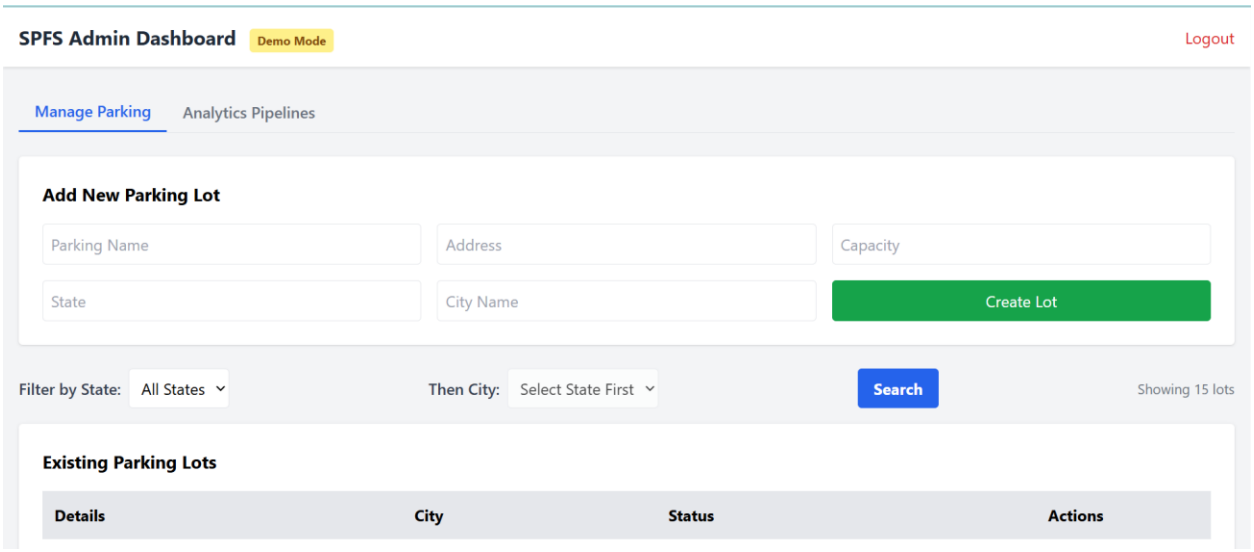

Figure 12: Admin Dashboard screen 1 (mockup)

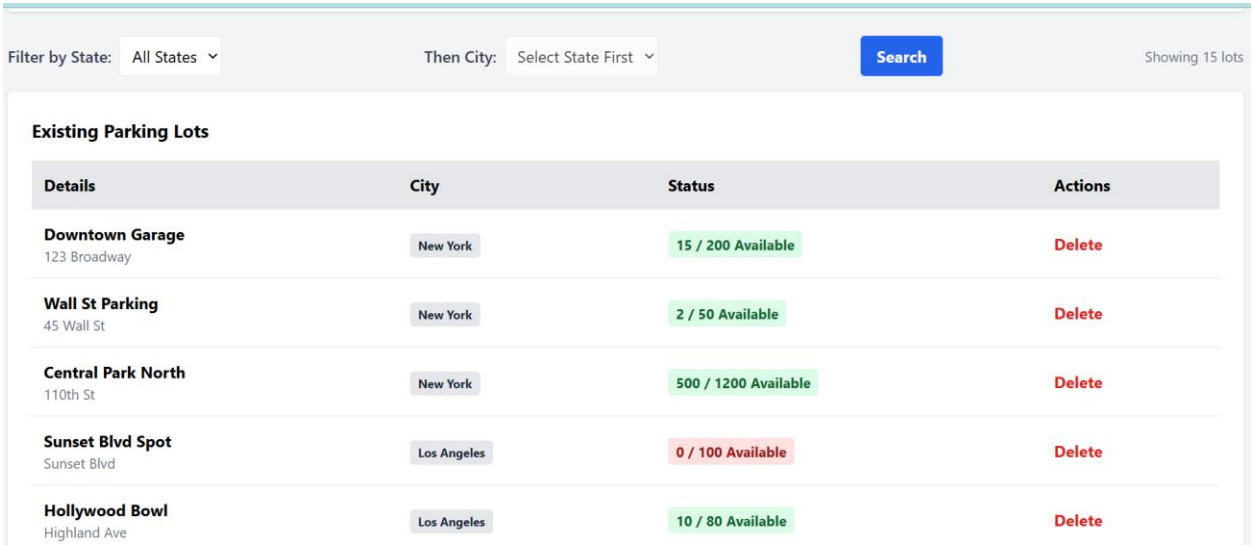

Figure 13: Admin Dashboard screen 2 (mockup)

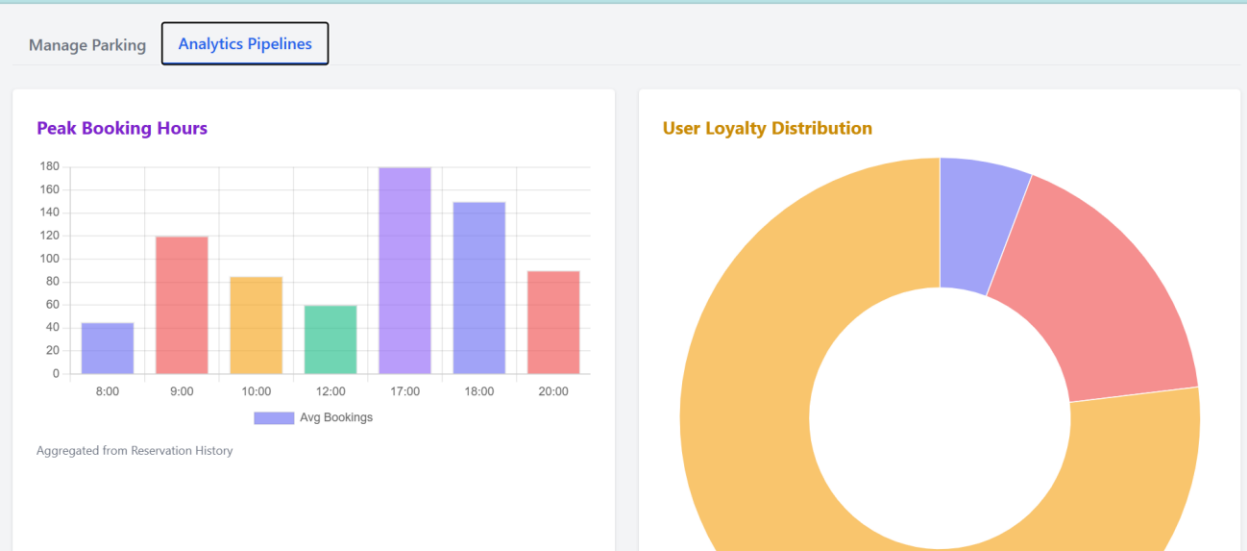

Figure: 14 Admin Analytics screen 1 (mockup)

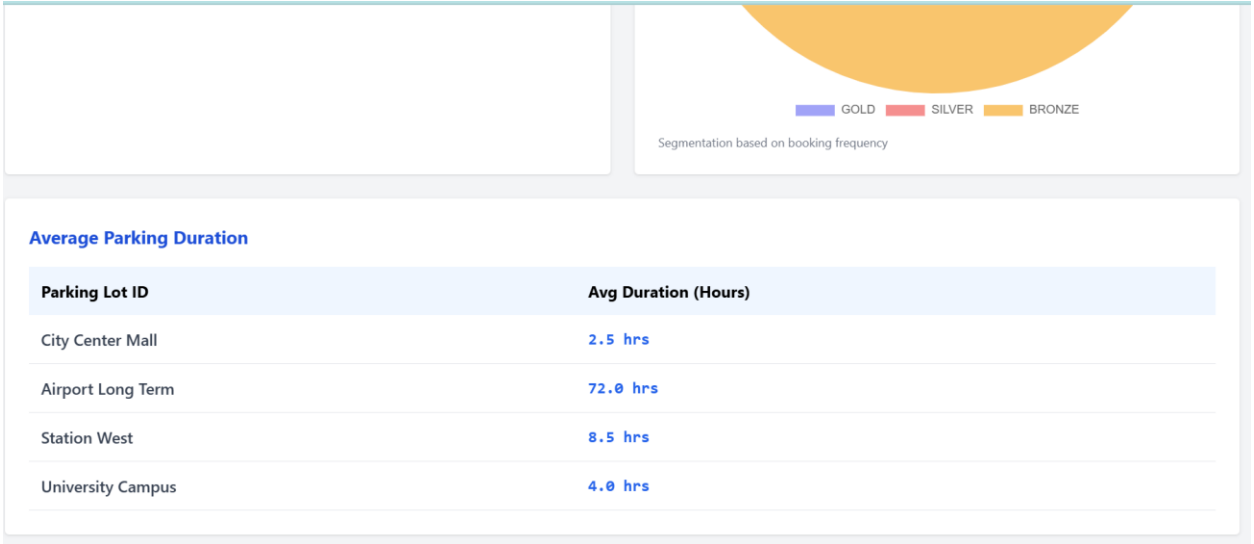

Figure 15: Admin Analytics screen 2 (mockup)

4 Data Models and Implementation

4.1 Document DB Collections

The Document DB is used to store all the data present in the application, in the following collections:

- Users
- Cities
- Parking Lots
- Slots
- Reservations

The implementation of these data structures on the Document DB has some peculiarities, driven by the nature of the queries.

4.1.1 User Entity

The **User** entity represents an individual registered within the system, including both standard users and administrative users. It stores essential authentication and identification information required for secure system access. The user's email address functions as a unique identifier and is used for login, communication, and account recovery. The role attribute defines the authorization level of the user and determines whether the user has administrative control or standard access privileges. Each user is associated with a specific city through a city object refrence, enabling the system to provide localized parking suggestions and default search behavior. To enhance account security, the user entity also stores a verification code that supports two-step authentication mechanisms.

A user is linked to exactly one city, representing the user's primary geographical association within the system.

4.1.2 City Entity

The **City** entity represents a geographical unit that organizes parking infrastructure within the system. It forms part of a hierarchical location structure consisting of state, and city levels, enabling structured and scalable geographic searches. In terms of relationships, a city contains multiple parking lots and serves as a parent entity for parking infrastructure.

4.1.3 ParkingLot Entity

The **ParkingLot** entity models a physical parking facility that contains a collection of parking slots. It stores descriptive information such as the parking lot name for identification and display purposes. To support high-performance availability queries, the entity maintains a cached count of available slots, which is synchronized with a real-time in-memory data store such as Redis. This approach minimizes database load while ensuring accurate availability reporting.

Each parking lot is logically associated with a city and exists within its geographical context. Additionally, a parking lot owns multiple parking slots, which collectively define its total parking capacity.

4.1.4 Slot Entity

The **Slot** entity represents an individual parking space within a parking lot. Each slot is uniquely identified by a slot number and maintains a status attribute that reflects its current state, such as available, booked, or occupied. This status information is essential for enforcing booking constraints and preventing concurrent reservations for the same parking space.

A slot is associated with exactly one parking lot and cannot exist independently outside of it. Furthermore, a slot may be referenced by a reservation entity, ensuring that each reservation corresponds to a specific and uniquely identifiable parking space.

4.1.5 Reservation Entity

The **Reservation** entity represents a transactional record that captures the booking of a parking slot by a user for a specified time interval. It stores temporal information such as the start time and end time of the reservation, along with a reservation status that indicates whether the booking is active, completed, or cancelled. The reservation also records the vehicle number, which is used to authorize access to the allocated parking slot.

Relationally, each reservation is linked to a single user who initiates the booking. It is also associated with a specific slot, guaranteeing exclusive usage during the reserved time period, and references the corresponding parking lot to provide contextual information about the booking location.

4.2 MongoDB Document Examples

This section provides concrete JSON examples of how data is stored in the underlying collections.

4.2.1 Users Collection

```
{
    _id: ObjectId('693ed842999c78508218763b'),
    fullName: 'Angela Bonham',
    email: 'angela.93@colon.com',
    password:
'$2b$12$PKYRtJDjOprJViAWQwX3POswXVgKnuoRrwh0RZ9rwUFDfTEP6SE6q',
    city: {
      _id: ObjectId('693ea977fec7a5540d29688e'),
      cityName: 'CHICO',
      state: 'CA',
      country: 'US'
    },
    role: 'User',
    isVerified: true,
```

```
    _class: 'unipi.lsmdb.SPFS.Entities.Users'
  }
```

4.2.2 Cities Collection

```
{
    _id: ObjectId('693ea977fec7a5540d2961ff'),
    cityName: 'DANBURY',
    state: 'CT',
    country: 'US',
    parkingLotIds:['693ea975fec7a5540d20a189',
'693ea975fec7a5540d20a19f' ],
    _class: 'unipi.lsmdb.SPFS.Entities.City'
  }
```

4.2.3 Parking Lots Collection

```
{
 "_id":"6589a1b2c3d4e5f678909876",
 "parkingName":"Downtown Central Garage",
 "fullAddress":"123 Main St, New York, NY",
 "totalCapacity":50,
 "availableSlots":50,
 "slotIds":[
  "6589a1b2c3d4e5f678901111",
  "6589a1b2c3d4e5f678901112"
 ],
}
```

Note: availableSlots is persisted here but the real-time truth is in Redis.

4.2.4 Slots Collection

```
{
  "_id": "6589a1b2c3d4e5f678901111",
  "slotNumber": "A-1",
  "status": "AVAILABLE",
```

```
}
```

4.2.5 Reservations Collection

```
{
  "_id": "6589a1b2c3d4e5f67890aaaa",
  "userId": "6589a1b2c3d4e5f678901234",
  "parkingLotId": "6589a1b2c3d4e5f678909876",
  "slotId": "6589a1b2c3d4e5f678901111",
  "vehicleNumber": "ABC-1234",
  "startTime": ISODate("2023-12-25T14:30:00.000Z"),
  "endTime": ISODate("2023-12-25T18:30:00.000Z"),
  "reservationStatus": "ACTIVE",
}
```

4.3 Indexes

Indexes are used to improve query performance and reduce response time, especially for frequently executed queries. In this project, **compound indexes and Single Indexes** are adopted to efficiently support queries that filter or sort data using multiple fields.

4.2.6 Geospatial Retrieval

The Geospatial Navigation Index (geo_idx) is a single-field B-Tree index on the state field within the cities collection, designed to optimize the application's "Hierarchy Browsing" mechanism across both public and administrative flow. The index allows the Public Registration Flow to instantly retrieve city lists for frontend dropdowns and enables the Admin Lot Creation Flow to rapidly validate city-state pairings without exhaustive database scans. With a cardinality of roughly 50 partitions, this index achieves approximately 98% selectivity, drastically reducing the search space and ensuring high-performance data retrieval for state-specific queries.

4.2.7 Active Reservation Retrieval (user_active_idx)

Definition: {'userId': 1, 'reservationStatus': 1}, This compound index is designed to optimize Read Operations where the system needs to retrieve a user's current status (e.g., enabling the 'Cancel Reservation' button on the dashboard). It allows the database engine to traverse the B-Tree and instantly locate the single 'ACTIVE' reservation amidst potentially thousands of 'COMPLETED' or 'CANCELLED' historical records, ensuring O(1) retrieval time for user profile views.

4.2.8 Temporal Sort Optimization (user_history_idx)

Definition: {'userId': 1, 'endTime': -1}, This index is implemented to optimize the retrieval of user reservation history, leveraging the Sort capability of MongoDB indexes. By indexing the userId followed by the endTime in descending order (-1), the data is effectively stored physically on disk in the exact order required for presentation. When a user requests their history, the database avoids a CPU-intensive "in-memory sort" operation. Instead, it utilizes the index to locate the user's records and streams them directly from the pre-sorted index pointers. This guarantees consistent low-latency response times for the "My Bookings" feature, regardless of how many thousands of past reservations a user may have accumulated.

4.2.9 Efficient Background Maintenance (scheduler_cleanup_idx)

Definition: {'reservationStatus': 1, 'endTime': 1}, This critical index supports the automated Reservation Cleanup Scheduler, optimizing a Range Query. The scheduler must identify reservations that are currently 'ACTIVE' but have expired (where endTime $<$ Current Time). Without this index, the database would perform a full Collection Scan, examining every reservation in the system to check its timestamp, which would degrade performance linearly as data grows. This index allows the query optimizer to isolate the 'ACTIVE' branch of the index and strictly scan the specific range of keys where the time condition is met, reducing the number of documents accessed from hundreds of thousands to merely the handful that require cleanup.

4.3 Redis Data Structures and Real-Time State Management

This section describes the Redis-based key–value schema used to support real-time availability tracking, concurrency control, and low-latency read operations within the parking reservation system. Redis is employed as an in-memory data store to handle high-frequency operations that require atomicity and fast access, which are not efficiently handled by disk-based databases alone.

4.3.1 Parking Lot Availability Pool

The **Parking Lot Availability Pool** is implemented using a Redis Set data structure and serves as the authoritative real-time representation of available parking slots within a parking lot. Each parking lot maintains a dedicated Redis key following a consistent naming convention that uniquely identifies the lot.

The key pattern lot:slots:{parkingLotId} maps to a Redis Set that contains the slot numbers currently available for booking. Since Redis Sets store unordered and distinct values, they are well-suited for managing slot availability without duplication.

This structure acts as the source of truth for slot availability during booking operations. When a user attempts to book a parking slot, the system performs an atomic SPOP operation on the set, which removes and returns a random available slot. This ensures that no two users can reserve the same slot concurrently. If the set is empty, the parking lot is considered full, and the booking request is rejected. For real-time dashboards and search results, the SCARD command provides constant-time access to the total number of available slots.

The lifecycle of this key begins when an administrator creates a new parking lot, at which point the set is initialized with all slot identifiers. During normal operation, slot identifiers are removed from the set when bookings are confirmed and reinserted when reservations are cancelled or expire. The key is deleted entirely if the parking lot is removed from the system.

Example:

If a parking lot with id `693ea975fec7a5540d209efc` has three available slots, the Redis state is represented as:

```
Key: lot:slots: 693ea975fec7a5540d209efc
Value: {"A-1", "B-5", "C-3"}
```

When a booking request is processed, the system executes:

```
SPOP lot:slots: 693ea975fec7a5540d209efc
```

If the returned value is "B-5", that slot is immediately reserved and removed from the availability pool. The remaining set becomes:

```
{"A-1", "C-3"}
```

To display real-time availability on the dashboard, the system executes:

```
SCARD lot:slots: 693ea975fec7a5540d209efc
```

Which returns 2, indicating that two slots remain available.

4.3.2 User Booking Lock

The **User Booking Lock** is implemented using a Redis String data structure and enforces the business rule that a user may have only one active booking at any given time. This mechanism prevents race conditions where a single user might attempt to book multiple slots concurrently through parallel requests.

Each user lock follows the key pattern user:active_booking:{userId} and stores a simple literal value, such as "ACTIVE". The lock is acquired using the Redis SETNX command, which sets the key only if it does not already exist. This operation is atomic, ensuring that concurrent booking attempts by the same user are safely rejected.

If the lock key already exists, the booking request is immediately denied with a client error response, without performing any availability checks. This design significantly reduces unnecessary load on both Redis and MongoDB.

The lifecycle of the lock begins at the start of a booking transaction and ends when the reservation is either completed or cancelled, at which point the key is deleted.

Example:

When a user with id `693ec30b7f31cc1c8f78fdbe` initiates a booking request, the system executes:

```
SETNX user:active_booking: 693ec30b7f31cc1c8f78fdbe "ACTIVE"
```

If the command returns 1, the lock is successfully acquired and the booking process continues. If it returns 0, the lock already exists, and the request is rejected.

The Redis state during an active booking appears as:

```
Key: user:active_booking: 693ec30b7f31cc1c8f78fdbe
Value: "ACTIVE"
```

Once the reservation is cancelled or completed, the system executes:

```
DEL user:active_booking: 693ec30b7f31cc1c8f78fdbe
```

This releases the lock and allows the user to initiate a new booking.

4.4 Redis Data Examples & Relationships

This section illustrates the actual data stored in Redis and how the "Cancellation Return" logic works.

4.4.1 Redis Keys & Values Example

Key Type	Key Name (Example)	Value (Example)	Purpose
Set	lot:slots:6589a1b2c3d4e5f678909876	["A-2", "B-5", "C-1"]	**Available Pool**. Contains *only* slots that are free to book.
String	user:active_booking:6589...1234	"ACTIVE"	**Concurrency Lock**. Prevents the same user from booking twice.

4.4.2 Relationship to MongoDB

Logical Link: The portion 6589a1b2c3d4e5f678909876 in the Redis Key lot:slots:... corresponds exactly to the _id of the **ParkingLot** document in MongoDB.

Sync Logic: The count of elements in the Redis Set (SCARD) should match the availableSlots field in the MongoDB document (eventually).

4.4.3 Cancellation Logic (How slots return)

When a user cancels a reservation, the system performs a "Return to Pool" operation using the Redis SADD command.

Scenario: User cancels reservation for Slot **"A-1"**.
Step-by-Step Flow:

1. **Identify**: System looks up the Reservation in MongoDB to find the parkingLotId and slotId.

2. **DB Update**: MongoDB marks the Reservation as CANCELLED and the Slot status as AVAILABLE.
3. **Redis Return (SADD)**: The system executes: SADD lot:slots:6589a1b2c3d4e5f678909876 "A-1"
    - *Effect*: "A-1" is added back into the Set. It is now instantly available for other users to SPOP.
4. **Unlock User**: The system deletes the user lock: DEL user:active_booking:6589...1234

4.4.4 Why SADD?

SADD (Set Add) is idempotent. If for some reason "A-1" was already in the set (e.g., a race condition or retry), adding it again does nothing. This ensures the pool never contains duplicate "A-1" entries, maintaining data integrity.

4.5 Inter-Database Consistency Strategy

Maintaining consistency between the in-memory cache (Redis) and the persistent data store (MongoDB) represents one of the most critical architectural challenges in the proposed parking finding system. Since Redis operations are executed in real time and MongoDB operations involve disk-based persistence, failures may occur at any stage of the transaction pipeline. To address this challenge, the system adopts a **Redis-First, Compensation-Rollback** consistency pattern.

In this pattern, Redis is treated as the authoritative source for real-time state and concurrency control, while MongoDB serves as the system of record for durable transactional data. Any inconsistency caused by partial failures is resolved through explicit compensation logic rather than distributed transactions, thereby avoiding the overhead and complexity of two-phase commit protocols.

4.5.1 Successful Execution Path (Happy Path)

In the normal execution scenario, a user successfully books a parking slot. The booking process begins by enforcing user-level concurrency constraints. The system first attempts to acquire a booking lock using the Redis SETNX command. If the lock is successfully obtained, the system proceeds to reserve a parking slot.

Next, an atomic SPOP operation is executed on the parking lot's availability pool in Redis. This operation removes and returns an available slot, such as `"A-1"`, ensuring that no other concurrent request can reserve the same slot. At this point, Redis reflects the updated real-time state by excluding the selected slot from the availability pool.

Following the successful Redis operations, the system persists the transaction in MongoDB by inserting a new reservation record and updating the corresponding slot status to BOOKED. Once this step completes, both Redis and MongoDB reflect a consistent state: the slot is removed from the Redis pool and marked as booked in persistent storage.

This execution path results in strong operational consistency, where Redis and MongoDB are synchronized without requiring additional corrective actions.

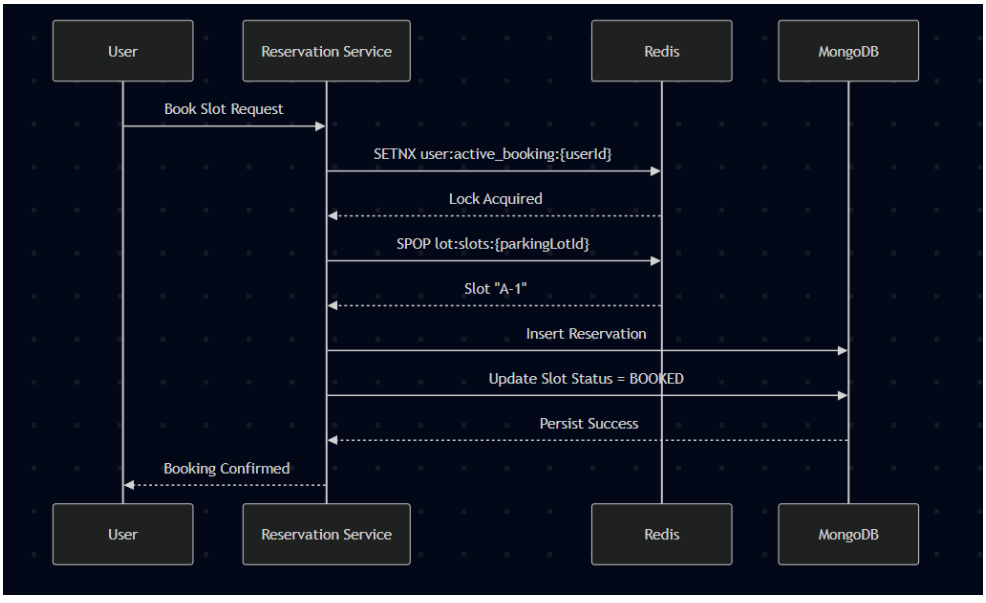

Figure 16: Successful Booking (Happy Path).

4.5.2 Failure and Rollback Scenario (Compensation Path)

The rollback scenario addresses the case where a failure occurs after Redis has already updated its state but before MongoDB persistence is completed. This situation may arise due to network interruptions, database crashes, or write timeouts.

In this scenario, the system successfully acquires the user booking lock using `SETNX` and removes a slot from the availability pool using `SPOP`. However, the subsequent attempt to insert the reservation into MongoDB fails. At this point, Redis reflects a partially updated state, while MongoDB remains unchanged.

To resolve this inconsistency, the system immediately triggers a compensation mechanism within the service layer, specifically inside the exception handling block of the `ReservationService`. The first corrective action reinserts the previously removed slot back into the Redis availability pool using the `SADD` command. This operation safely restores the slot to the pool and guarantees idempotency in case of retries. The second corrective action releases the user-level lock by deleting the corresponding Redis key.

After compensation, the system returns to a logically consistent state equivalent to the pre-booking condition. The user receives a failure response indicating that the booking could not be completed, and the parking slot becomes instantly available for other users.

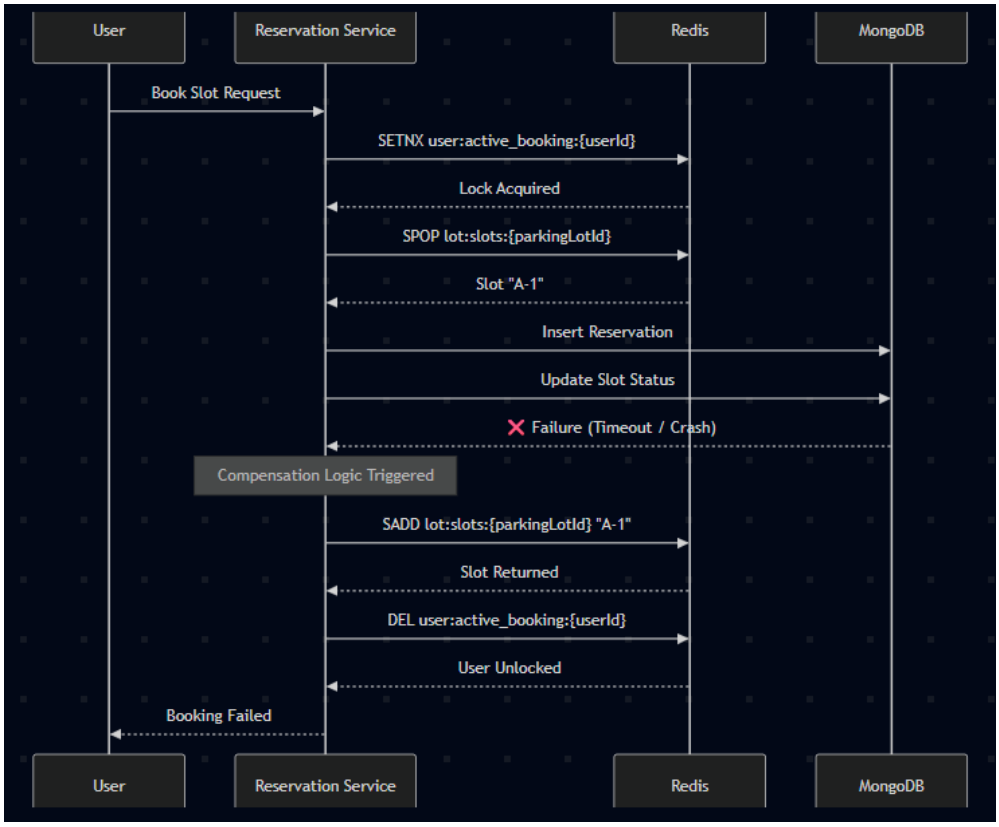

Figure 17: Failure with Compensation (Rollback Scenario).

4.5.3 Consistency Guarantees and Design Rationale

This strategy provides **eventual consistency with immediate correction**, ensuring that transient failures do not result in resource leakage or phantom bookings. By avoiding distributed locking or two-phase commits, the system remains highly performant and resilient under load. Redis acts as the fast-moving control layer, while MongoDB preserves long-term correctness and auditability.

The Redis-First, Compensation-Rollback pattern thus offers a practical and scalable solution for maintaining consistency across heterogeneous data stores in real-time reservation systems.

4.5.4 CAP Theorem in the SPFS

In the proposed Smart Parking Finder System (SPFS), Redis and MongoDB play complementary roles. Redis is responsible for real-time slot allocation and concurrency control, while MongoDB serves as the persistent system of record for reservations and historical data. According to the CAP theorem, a distributed system can guarantee at most two of the following properties: Consistency (C), Availability (A), and Partition Tolerance (P). Since the system is distributed and relies on networked services, **partition tolerance is a mandatory requirement**.

For **write operations**, such as booking and cancelling parking slots, the system prioritizes **consistency over availability**. Slot allocation is performed using atomic Redis operations (SETNX for user locking and SPOP for slot assignment), ensuring that no parking slot can be double-booked. If Redis is unavailable, booking requests are rejected and no write operation is performed. If MongoDB is unavailable, a rollback mechanism restores the slot in Redis. This behavior corresponds to a **CP (Consistency + Partition Tolerance)** choice, as correctness is preserved even at the cost of temporary unavailability.

For **read operations**, particularly the display of available parking slots, the system adopts a more availability-oriented approach. Under normal conditions, availability data is retrieved directly from Redis for real-time accuracy. However, MongoDB maintains a synchronized backup value of the available slots counter, updated during every booking and cancellation. If Redis becomes unavailable, the system gracefully falls back to MongoDB, allowing users to continue viewing parking availability. Although this value may be slightly stale in rare edge cases, the system remains operational. This behavior aligns with an **AP (Availability + Partition Tolerance)** model and ensures service continuity.

Overall, SPFS implements a **hybrid CAP strategy**, applying **CP guarantees to critical write operations** and **AP guarantees to read operations**. This selective application of CAP principles ensures strong consistency where correctness is essential, while maintaining high availability and resilience for user-facing read operations.

4.5.5 Replicas

The database layer of the application is deployed on a cluster that supports replication to improve availability and fault tolerance. In this project, replication is implemented only for **MongoDB**, while **Redis is used as a single in-memory data store for real-time operations**.

The final setup is:

- **MongoDB**: 3 replicas (1 Primary, 2 Secondaries)
- **Redis**: Single instance for availability and concurrency control

4.5.6 Read Operations

Read operations are the most frequent in the system. To ensure fast response times, **real-time parking slot availability is read directly from Redis**, which provides very low latency access.

MongoDB is allowed to read from secondary replicas. Users do not require strictly up-to-date data for these operations, therefore **eventual consistency is acceptable**.

This approach reduces load on the primary replica and improves system availability.

4.5.7 Write Operations

All write operations are handled by the MongoDB primary replica to ensure data consistency. A write is considered successful when it is stored on the primary node, while replication to secondary nodes happens asynchronously.

For booking operations, Redis is used first to reserve a slot atomically. If the MongoDB write fails, a rollback mechanism restores the slot in Redis, keeping the system consistent.

5 Implementation

The project has been implemented following the most modern technologies, Spring boot as Back-end framework and MongoDB and Redis as DBMSs.

The full implementation code can be found on the SPFS github repository.

5.1 Software Modules

5.1.1 Overall Architecture

The application is developed as a modular Spring Boot project using the Maven build system. Its design follows the conventional multi-layered architecture typical of enterprise Java applications, where different concerns are separated into distinct packages: controllers (for HTTP interface), services (for business logic), repositories for data access), configuration (for security and application setup), models (for MongoDB documents), and data transfer objects (DTOs, for structuring API responses and requests). This architecture ensures maintainability, estability, and separation of concerns.

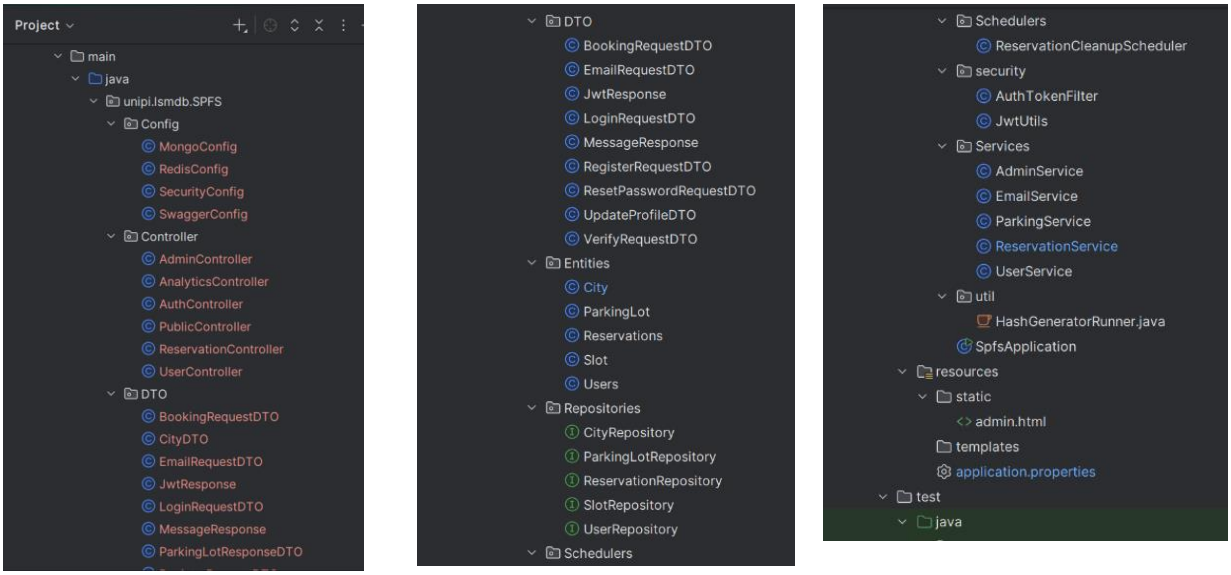

Figure 18: Package Structure unipi.lsmdb.SPFS: Root package.

- **Config**: Configuration classes (Redis, Security, MongoDB).
- **Controller**: REST Controllers handling HTTP requests (PublicController, ReservationController).
- **Services**: Business logic layer (ReservationService, EmailService, Admin Services).
- **Repositories**: Data access interfaces extending MongoRepository.
- **Entities**: Data objects mapping to MongoDB collections (Users, ParkingLot, Reservations).
- **DTO**: Data Transfer Objects for API requests/responses.

5.1.1.1 Maven and Spring Boot Integration

The project uses Maven as the build automation and dependency management tool. All required libraries are defined in the pom.xml file, including dependencies for Spring Boot, Spring Data MongoDB, Data-Redis Spring Security, Lombok, Swagger/OpenAPI, and testing frameworks.

Spring Boot simplifies application development by providing starter dependencies, which automatically configure the application based on the libraries present in the classpath and the defined properties.

The application is started through the main() method in the SpfsApplication class, located in the unipi.lsmdb.SPFS root package. This class is annotated with @SpringBootApplication, enabling component scanning, auto-configuration, and automatic registration of configuration classes.

The application.properties file defines the access details for MongoDB, Redis, and the email service, allowing Spring to automatically handle all required connections. It also includes configuration parameters for logging and runtime behavior, ensuring that errors and consistency issues can be properly monitored. Additionally, the file allows the activation of different application profiles, making it possible to enable or disable specific components,

such as database initialization or data-loading modules, depending on the execution environment.

5.1.1.2 Package structure

The main packages are structured as follows:

- **Controller**: Located in unipi.lsmdb.SPFS.Controller, this package contains REST controllers for different user roles and functionalities: AuthController (public access), PublicController (search), ReservationController (booking), UserController (profiles), AdminController (management), and AnalyticsController (stats). Each controller maps HTTP requests to service calls, returning data encapsulated in standard ResponseEntity<T> objects.
- **Service**: The unipi.lsmdb.SPFS.Services package implements the business logic. It includes classes like ReservationService, UserService, EmailService, and AdminService. Each service interacts with multiple repositories (Mongo & Redis) and implements core operations such as Atomic Booking, aggregation logic, and email notifications. Application logic ensures data consistency between the two databases.
- **Entities (Model)**: Document classes used for MongoDB persistence reside in unipi.lsmdb.SPFS.Entities. These include: Users, ParkingLot, Slot, Reservations, and City. Each model is annotated with Spring Data's @Document annotation and defines the structure stored in MongoDB. Relationships such as users referencing cities (cityCollectionId) or slots referencing lots are maintained via ID references to support distributed scaling.
- **DTO (Data Transfer Objects)**: Located in unipi.lsmdb.SPFS.DTO, these classes mediate between internal models and the external API representation. Examples include RegisterRequestDTO, LoginRequestDTO,

  BookingRequestDTO, JwtResponse, and UpdateProfileDTO. They are data carriers annotated with Lombok to reduce boilerplate and support automatic JSON serialization/deserialization by Jackson. DTOs are used strictly for API inputs and outputs.

- **Repository**: The unipi.lsmdb.SPFS.Repositories package contains interfaces that extend MongoRepository. These repositories (UserRepository, ParkingLotRepository, SlotRepository, ReservationRepository, CityRepository) manage data persistence and custom queries over collections, providing out-of-the-box CRUD operations and custom finders (e.g., findByEmail, findByCityId).
- **Config**: The unipi.lsmdb.SPFS.Config package (along with unipi.lsmdb.SPFS.security) holds the application's configuration classes. This includes:
  - **SecurityConfig**: Configures Spring Security, defining public routes (/api/public/**) vs protected routes and JWT filters.
  - **RedisConfig**: Configures the LettuceConnectionFactory and RedisTemplate for high-performance interaction with the Redis standalone instance.
  - **MongoConfig**: Custom MongoDB configurations if needed.
  - **SwaggerConfig**: Configures the OpenAPI/Swagger UI documentation.
  - **JwtUtils** (in security): Handles the generation, parsing, and validation of JWT tokens.

5.2 Queries

In this section we will describe the most complex queries and aggregation on both MongoDB and Redis. We will not describe simple CRUD operations. The full implementation of all methods, including the ones not listed here, can be found in the application source code.

5.2.1 MongoDB aggregations

All aggregations are presented in MongoDB query language. The actual implementation in the application is done using Spring Framework's construction functions, avoiding as much as possible the use of direct json translation. For technical reasons, the resulting aggregations can thus differ slightly from the ones presented here, but the behaviour is the same.

5.2.1.1 Average Parking Duration (Analytics)

- **Description**: Calculates the average time users spend in each parking lot. It filters out invalid records, sanitizes data, computes duration in hours, and groups by parking lot.

Average Parking Duration

Parking Lot ID	Avg Duration (Hours)
City Center Mall	2.5 hrs
Airport Long Term	72.0 hrs
Station West	8.5 hrs
University Campus	4.0 hrs

Figure 19: Average Parking Duration Plot mockup.

Pipeline:

```
db.reservations.aggregate([
    // 1. Filter completed records
    { $match: { startTime: { $exists: true }, endTime: { $exists: true
} } },
    // 2. Project Duration (End - Start)
    { $project: {
        parkingLotId: 1,
        durationHours: { $divide: [ { $subtract: ["$endTime",
"$startTime"] }, 3600000 ] }
    }},
    // 3. Group and Average
    { $group: {
        _id: "$parkingLotId",
        avgDuration: { $avg: "$durationHours" }
    }},
    // 4. Sort Descending
    { $sort: { avgDuration: -1 } },
    { $limit: 10 }
])
```

Java Code

```
public List<Map> getAverageParkingDuration() {
        Aggregation aggregation = Aggregation.newAggregation(
                        // Step 1: Filter out invalid entries
```

```
Aggregation.match(Criteria.where("startTime").exists(true).ne(null)

.and("endTime").exists(true).ne(null)),

                        // Step 2: Sanitize startTime
                        Aggregation.project("parkingLotId", "endTime")
                                        .and(ConditionalOperators.when(
                                                        context -> new
org.bson.Document("$eq",

java.util.Arrays.asList(

new org.bson.Document(

"$type",

"$startTime"),

"string")))

.then(org.springframework.data.mongodb.core.aggregation.StringOperators.Subst
r

.valueOf("$startTime").substring(0, 19))

.otherwise("$startTime"))
                                        .as("sanitizedStartTime"),

                        // Step 3: Sanitize endTime (keep sanitizedStartTime)
                        Aggregation.project("parkingLotId",
"sanitizedStartTime")
                                        .and(ConditionalOperators.when(
                                                        context -> new
org.bson.Document("$eq",

java.util.Arrays.asList(

new org.bson.Document(

"$type",

"$endTime"),

"string")))

.then(org.springframework.data.mongodb.core.aggregation.StringOperators.Subst
r

.valueOf("$endTime").substring(0, 19))

.otherwise("$endTime"))
                                        .as("sanitizedEndTime"),

                        // Step 4: Convert both to Dates
                        Aggregation.project("parkingLotId")

.and(org.springframework.data.mongodb.core.aggregation.ConvertOperators.ToDat
```

```
e

.toDate("$sanitizedStartTime"))
                                        .as("start")

.and(org.springframework.data.mongodb.core.aggregation.ConvertOperators.ToDat
e

.toDate("$sanitizedEndTime"))
                                        .as("end"),

                        // Step 5: Calculate duration in hours
                        Aggregation.project("parkingLotId")
                                        .andExpression("(end - start) /
3600000").as("durationHours"),

                        // Step 6: Group and Average

Aggregation.group("parkingLotId").avg("durationHours").as("avgDuration"),

                        // Step 7: Sort

Aggregation.sort(org.springframework.data.domain.Sort.Direction.DESC,
"avgDuration"),

                        // Step 8: Top 10 Limit
                        Aggregation.limit(10));

        AggregationResults<Map> results =
mongoTemplate.aggregate(aggregation, "reservations", Map.class);
        List<Map> rawResults = results.getMappedResults();

        // 1. Extract IDs (The group result puts the ID in "_id")
        List<String> lotIds = rawResults.stream()
                        .map(m -> (String) m.get("_id"))
                        .filter(id -> id != null)
                        .map(String::trim)
                        .collect(java.util.stream.Collectors.toList());

        // 2. Bulk Fetch ParkingLots
        List<ParkingLot> lots = parkingLotRepository.findAllById(lotIds);

        // 3. Create Lookup Map (ID -> ParkingLot)
        Map<String, ParkingLot> lotMap = lots.stream()

.collect(java.util.stream.Collectors.toMap(ParkingLot::getId, lot -> lot));

        // 4. Merge Data
        List<Map> enrichedResults = new java.util.ArrayList<>();
        for (Map original : rawResults) {
                String id = (String) original.get("_id");
                if (id != null)
                        id = id.trim();

                ParkingLot lot = lotMap.get(id);

                // Create new map to ensure mutability
```

```
                Map<String, Object> enriched = new
java.util.HashMap<>(original);

                // Normalize the ID field for the frontend
                enriched.put("parkingLotId", id);

                if (lot != null) {
                        enriched.put("parkingName", lot.getParkingName());
                        enriched.put("fullAddress", lot.getFullAddress());
                } else {
                        enriched.put("parkingName", "Unknown Lot");
                        enriched.put("fullAddress", "Unknown Address");
                }
                enrichedResults.add(enriched);
        }

        return enrichedResults;
}
```

The aggregation is composed of the following stages:

1. **match**: filters out invalid records where startTime or endTime might be missing to ensure data integrity.
2. **project**: computes the duration for each reservation. It subtracts the start time from the end time and divides by 3,600,000 to convert milliseconds to Hours.
3. **group**: groups the results by parkingLotId and calculates the average duration ($avg) for each lot.
4. **sort**: sorts the parking lots in descending order of duration to show the longest-stay lots first.
5. **limit**: restricts the result to the top 10 entries.

5.2.1.2 User Loyalty Tier Segmentation

- **Description**: Segments users into tiers (GOLD, SILVER, BRONZE) based on their total booking count using conditional logic ($cond / $switch).

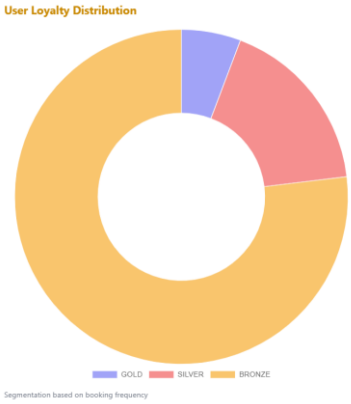

Figure 20: User Loyalty Tier Segmentation Plot mockup.

Pipeline:

```
db.reservations.aggregate([
    // 1. Count bookings per user
    { $group: { _id: "$userId", bookingCount: { $count: {} } } },
    // 2. Assign Tier
    { $project: {
        loyaltyTier: {
            $cond: { if: { $gte: ["$bookingCount", 20] }, then: "GOLD",
            else: { $cond: { if: { $gte: ["$bookingCount", 5] }, then:
"SILVER", else: "BRONZE" } } }
        }
    }},
    // 3. Count Users per Tier
    { $group: { _id: "$loyaltyTier", userCount: { $count: {} } } }
])
```

Java Code

```
public List<Map> getUserLoyaltyDistribution() {
        // Since $bucket can be complex with MongoTemplate, we use a
Facet or
        // Conditional
        // Project
        // approach which is clearer for "well-done" segmentation.

        Aggregation aggregation = Aggregation.newAggregation(
                        // Step 1: Calculate total bookings per user

Aggregation.group("userId").count().as("bookingCount"),

                        // Step 2: Categorize into Tiers using
Conditional Logic ($switch / $cond)
                        Aggregation.project("bookingCount")

.and(ConditionalOperators.when(Criteria.where("bookingCount").gte(20))
                                                        .then("GOLD")

.otherwise(ConditionalOperators

.when(Criteria.where("bookingCount")
```

```
.gte(5))

.then("SILVER")

.otherwise("BRONZE")))
                                        .as("loyaltyTier"),

                        // Step 3: Group by the new Tier field to get
the specific counts

Aggregation.group("loyaltyTier").count().as("userCount"),

                        // Step 4: Rename _id to loyaltyTier and add
Criteria Description
                        Aggregation.project("userCount")
                                        .and("_id").as("loyaltyTier")

.and(ConditionalOperators.when(Criteria.where("_id").is("GOLD"))
                                                        .then("20 or
more bookings")

.otherwise(ConditionalOperators

.when(Criteria.where("_id")

.is("SILVER"))

.then("5 to 19 bookings")

.otherwise("Less than 5 bookings")))
                                        .as("criteria")
                                        .andExclude("_id"));

        AggregationResults<Map> results =
mongoTemplate.aggregate(aggregation, "reservations", Map.class);
        return results.getMappedResults();
}
```

The aggregation is composed of the following stages:

1. **group**: groups all reservations by userId and counts the total number of bookings per user.
2. **project**: applies conditional logic ($cond) to assign a tier. Users with >= 20 bookings become "GOLD", those with >= 5 become "SILVER", and others "BRONZE".
3. **group**: regroups the data by the newly assigned loyaltyTier to count how many users fall into each category.

5.2.1.3 Peak Booking Hours

- **Description**: Extracts the hour of day from reservation start times to identify peak usage periods.

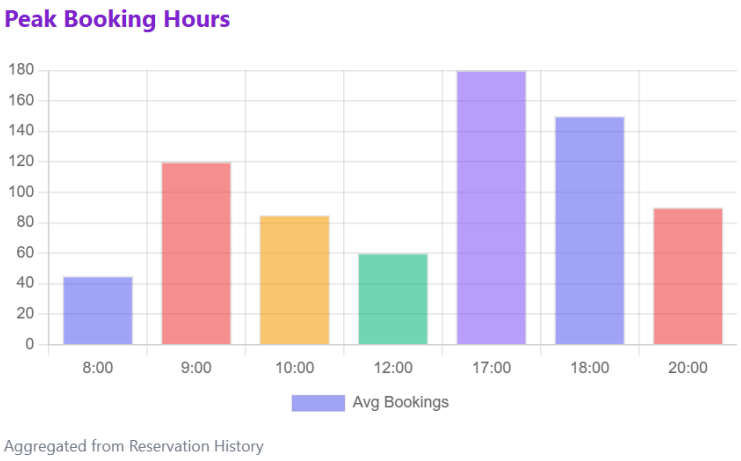

Figure 21: Peak Booking Hours Report mockup.

- **Pipeline**:

```
db.reservations.aggregate([
    { $project: { hourOfDay: { $hour: "$startTime" } } },
    { $group: { _id: "$hourOfDay", count: { $count: {} } } },
    { $sort: { count: -1 } }
])
```

- **Java Code**

```
public List<Map> getPeakBookingHours() {
        Aggregation aggregation = Aggregation.newAggregation(
                        // Step 1: Filter out naturally invalid entries

Aggregation.match(Criteria.where("startTime").exists(true).ne(null)),

                        // Step 2: Sanitize and Convert
                        // We truncate the string to 19 chars (yyyy-MM-
ddTHH:mm:ss) to remove
                        // malformed microseconds/timezones like
".692096.000+00:00"
                        Aggregation.project()
                                        .and(ConditionalOperators.when(
                                                        // Custom
expression: { $eq: [ { $type: "$startTime" },
                                                        // "string" ] }
context -> new org.bson.Document("$eq",

java.util.Arrays.asList(
```

```
new org.bson.Document(

"$type",

"$startTime"),

"string")))

.then(org.springframework.data.mongodb.core.aggregation.StringOperators
.Substr

.valueOf("$startTime").substring(0, 19))

.otherwise("$startTime"))
                                        .as("sanitizedDate"),

                        // Step 3: Convert to actual Date object
                        Aggregation.project()

.and(org.springframework.data.mongodb.core.aggregation.ConvertOperators
.ToDate

.toDate("$sanitizedDate"))
                                        .as("convertedDate"),

                        // Step 4: Extract the hour
                        Aggregation.project()

.andExpression("hour(convertedDate)").as("hourOfDay"),

                        // Step 5: Group and Sort

Aggregation.group("hourOfDay").count().as("count"),

Aggregation.sort(org.springframework.data.domain.Sort.Direction.ASC,
"_id"),

                        // Step 6: Rename _id to hour for cleaner JSON
response

Aggregation.project("count").and("_id").as("hour").andExclude("_id"));

        AggregationResults<Map> results =
mongoTemplate.aggregate(aggregation, "reservations", Map.class);
        return results.getMappedResults();
}
```

The aggregation is composed of the following stages:

1. **project**: extracts the hour of the day (0-23) from the startTime timestamp.
2. **group**: groups by the extracted hour and counts the number of reservations initiated in that hour.

3. **sort**: sorts the results in order of count to identify the busiest times of day.

5.3 Redis Atomic Operations

Redis is used for high-concurrency operations where atomicity is critical.

5.3.1 Atomic Slot Allocation (SPOP)

The operation is composed of the following stages:

1. **key construction**: constructs the unique key for the parking lot's slot set (e.g., lot:slots:123).
2. **pop**: calls opsForSet().pop() which atomically removes and returns a random member from the set. This ensures concurrency safety; two threads cannot pop the same slot.
3. **null check**: checks if the returned object is null. If it is, the set is empty, meaning the lot is full, and an exception is thrown.

```
// 1. Construct Key
String slotKey = "lot:slots:" + parkingLotId;
// 2. Atomic Pop (Critical Section)
Object slotObj = redisTemplate.opsForSet().pop(slotKey);
// 3. Validation
if (slotObj == null) {
throw new RuntimeException("Parking Lot is Full");
}
String slotNumber = slotObj.toString();
```

5.3.2 User Locking (SETNX)

The operation is composed of the following stages:

1. **key construction**: constructs a unique lock key for the user (e.g., user:active_booking:456).

2. **setIfAbsent**: calls opsForValue().setIfAbsent("ACTIVE"). This executes the Redis SETNX command. It returns true ONLY if the key did not exist.
3. **boolean check**: verifies the result. If false, it means the key existed (user has a lock), so the transaction is rejected immediately.

```
// 1. Construct Key

String userLockKey = "user:active_booking:" + userId;

// 2. Try to Acquire Lock (Atomic SETNX)

Boolean lockAcquired =
redisTemplate.opsForValue().setIfAbsent(userLockKey,
"ACTIVE");

// 3. Verify Result

if (Boolean.FALSE.equals(lockAcquired)) {

throw new RuntimeException("User already has an active
booking. Please cancel it before booking again.");

}
```

5.4 Data Classes Documentation

This document describes the Data Models (Entities) and Data Transfer Objects (DTOs) used in the Smart Parking Finding System.

5.4.1 Entities (MongoDB Collections)

Located in com.example.SPFS.Entities. These classes map directly to MongoDB collections.

5.4.1.1 Users

- **Collection:** User
- **Purpose:** Stores user account information and role.
- **Code**

```
@Data
@Document(collection = "users")
public class Users {
    @Id
    private String id;
    private String fullName;
    private String email;
```

```
    @JsonIgnore
    private String password; // Stored as Hashed value
    private City city; // Embedded City Object (Lite version)

    public void setCity(City city) {
        if (city != null) {
            // Create a "lite" version of the city to avoid embedding
heavy data like
            // parkingLotIds
            City liteCity = new City();
            liteCity.setId(city.getId());
            liteCity.setCityName(city.getCityName());
            liteCity.setState(city.getState());
            liteCity.setCountry(city.getCountry());
            // Do NOT copy parkingLotIds
            this.city = liteCity;
        } else {
            this.city = null;
        }
    }

    private String role; // always User

    private boolean isVerified = false; // Default to false
    @JsonIgnore
    private String verificationCode;
    @JsonIgnore
    private String passwordResetCode;
}
```

5.4.1.2 City

- **Collection:** cities
- **Code**

```
@Data
@CompoundIndex(name = "geo_idx", def = "{'state': 1}")
@Document(collection = "cities")
@JsonInclude(JsonInclude.Include.NON_NULL)
public class City {
    @Id
    private String id;

    private String cityName;
    private String state;
    private String country;

    private List<String> parkingLotIds;
    // Getters and Setters
}
}
```

5.4.1.3 ParkingLot

- **Collection:** parking_lots
- **Purpose:** Represents a physical parking facility.
- **Code**

```
@Data
@Document(collection = "parking_lots")
public class ParkingLot {
    @Id
    private String id;
    private String parkingName;
    private String fullAddress;
    private int totalCapacity;
    private int availableSlots;
    private List<String> slotIds; // Document Linking
(Strategy I)
}
```

5.4.1.4 Slot

- **Collection:**
- **Purpose:** Represents an individual parking space.
- **Code**

```
@Data
@Document(collection = "slots")
public class Slot {
    @Id
    private String id;
    private String slotNumber; // e.g., "A-1"
    private String status; // e.g., "AVAILABLE", "BOOKED"
}
```

5.4.1.5 Reservations

- **Collection:** reservations
- **Code**

```
@Data
@Document(collection = "reservations")
@org.springframework.data.mongodb.core.index.CompoundIndexes({

@org.springframework.data.mongodb.core.index.CompoundIndex(nam
e = "user_active_idx", def = "{'userId': 1,
'reservationStatus': 1}"),

@org.springframework.data.mongodb.core.index.CompoundIndex(nam
e = "user_history_idx", def = "{'userId': 1, 'endTime': -1}"),

@org.springframework.data.mongodb.core.index.CompoundIndex(nam
e = "scheduler_cleanup_idx", def = "{'reservationStatus': 1,
'endTime': 1}")
})
public class Reservations {
    @Id
    private String id;
    private String userId;
    private String parkingLotId;
    private String slotId; // Stores MongoDB _id of the Slot
document
    private String vehicleNumber;

    // Using LocalDateTime for easier calculations
    private LocalDateTime startTime;
    private LocalDateTime endTime;

    private String reservationStatus; // "ACTIVE",
"COMPLETED", "CANCELLED"
}
```

5.4.2 Data Transfer Objects (DTOs)

Data Transfer Objects (DTOs) are fundamental to the application's architecture, serving as the strict contract for data exchange between the client interface and the server-side controllers. By encapsulating data into specialized objects, the system decouples the internal database entities from the external API, ensuring that the database schema can evolve independently without breaking client integrations. This layer also acts as a security filter, preventing the accidental exposure of sensitive internal fields.

5.4.2.1 Authentication and User Management

The **RegisterRequestDTO** functions as the initial data payload for user onboarding. It aggregates the user's personal identity information, specifically their full name and email address, along with the raw password for subsequent hashing. Crucially, it also captures the user's initial geographical preference by including the state, and city names.

The **LoginRequestDTO** is a streamlined credential carrier designed solely for the authentication handshake. It transmits the user's email and raw password to the AuthController, where they are validated against the stored credentials. Upon successful authentication, the system responds with the **JwtResponse**. This complex response object establishes the client's session state. It delivers the generated JSON Web Token (JWT) for authorizing future requests, alongside essential user context such as the unique MongoDB ID, email, assigned role, full name, and verification status. This allows the frontend client to construct a personalized user interface immediately upon login without needing subsequent data fetches.

For post-registration modifications, the **UpdateProfileDTO** encapsulates the editable subset of user data. It allows users to submit changes to their full name, phone number, and physical address. By restricting updates to this specific object, the system strictly controls which fields can be modified by the user, protecting immutable attributes like the unique ID or assigned role.

5.4.2.2 Booking and Operations

The **BookingRequestDTO** represents the explicit intent of a user to reserve a parking resource. It carries the necessary foreign keys to link the transaction: the userId of the requester and the parkingLotId of the target facility. Additionally, it includes the vehicleNumber for enforcement purposes and the hours field to define the duration of the stay, which the service uses to calculate the expiration time.

To maintain a clean and efficient API response structure, the **CityDTO** provides a sanitized view of the City entity. Unlike the full database document, which includes a potentially massive list of associated parking lot IDs, this DTO contains only the essential location descriptors: the unique identifier, city name, state. This design prevents circular reference issues and reduces payload size when embedding city details within other responses.

The **ParkingLotResponseDTO** serves as the primary data structure for the parking inventory feed. It is a composite object that enriches the static ParkingLot entity with dynamic context. While it mirrors the core fields of the entity—such as the ID, name, and total capacity—it critically replaces the raw availableSlots value with real-time data fetched from the Redis cache. Furthermore, it embeds the CityDTO defined above, providing clients with a complete, self-contained representation of the parking facility and its location in a single HTTP response.

5.4.2.3 System Utilities

The system employs several specialized DTOs to handle auxiliary workflows. The **MessageResponse** is a generic wrapper used to standardize simple server communications, delivering clear success or error notifications to the client in a consistent JSON format. The **VerifyRequestDTO** facilitates the email verification process by transmitting the user's email alongside the one-time verification code. Similarly, the **EmailRequestDTO** is a single-field carrier used to initiate the "Forgot Password" and "Resend Code" flows, verifying the user's existence before triggering email events. Finally, the **ResetPasswordRequestDTO** completes the recovery cycle by securely bundling the user's

email, the validation code, and the new password, allowing the system to verify the request's legitimacy before updating the credential store.

The controller exposes its functionalities through RESTful APIs. RESTful APIs (Representational State Transfer) represent a standardized architectural style for designing networked applications. They allow client-side applications to interact with the backend services through HTTP methods, each of which corresponds to a specific type of action: GET to retrieve resources, POST to create new ones, PUT to update existing ones, and DELETE to remove them. This approach offers a stateless and scalable method for accessing and manipulating resources, and is widely adopted due to its simplicity, readability, and compatibility with web technologies.

In this project, RESTful APIs are implemented using the Spring Boot frame- work, which offers extensive support for defining controllers and endpoints through declarative annotations. Each controller in the application is annotated with @RestController, a specialized version of @Controller that automatically seri- alizes return values into JSON. The base path of the controller is defined using @RequestMapping("/api/admin"), and individual methods are mapped to specific end- points using annotations such as @PostMapping, @PutMapping, and @DeleteMapping. These annotations define the HTTP verb to handle, and often include path variables or query parameters for identifying or modifying specific resources. For example, when updating or deleting a venue, the identifier is passed as a @PathVariable, while the data to be updated is received as a JSON object mapped to a @RequestBody.

Authentication and authorization are managed using Spring Security, configured through a custom SecurityConfig class. In this setup, users are assigned roles: USER for regular users and ADMIN for administrators. These roles are enforced at the endpoint level by specifying access restrictions, ensuring that only users with administrative privileges can perform sensitive actions such as modifying database records. The authentication mechanism ensures that the system is secure, distinguishing between users who merely interact with content and those who have the authority to modify its structure.

5.4.2.4 Admin Controller

The **AdminController** exposes a set of privileged REST APIs dedicated to system administrators. Its primary responsibility is to manage core system resources, including parking lots, users, and cache synchronization mechanisms. All endpoints under this controller are protected and accessible only to users with administrative privileges.

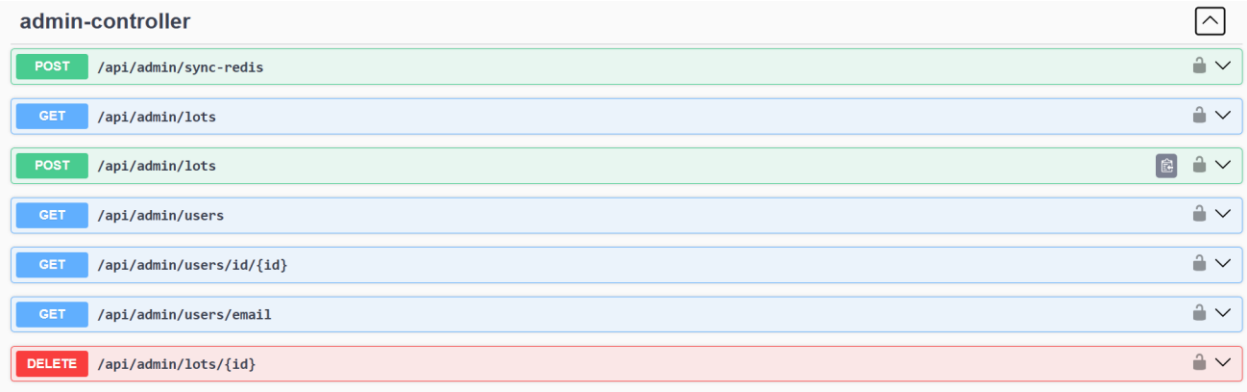

Figure 22: All the APIs under Admin Controller.

5.4.2.5 Parking Lot Management APIs

The endpoint for creating a new parking lot is implemented through the createParkingLot() method, which listens to **POST** requests at the path /api/admin/lots. This operation requires a JSON request body representing the ParkingLot entity, along with query parameters that specify the geographic context of the parking facility, namely cityName, and State. The request handling logic is delegated to the AdminService.createLot() method, which persists the parking lot data in MongoDB and initializes the corresponding availability structure in Redis. During this process, all parking slots associated with the new lot are created and marked as available. Upon successful execution, the newly created ParkingLot object is returned to the client.

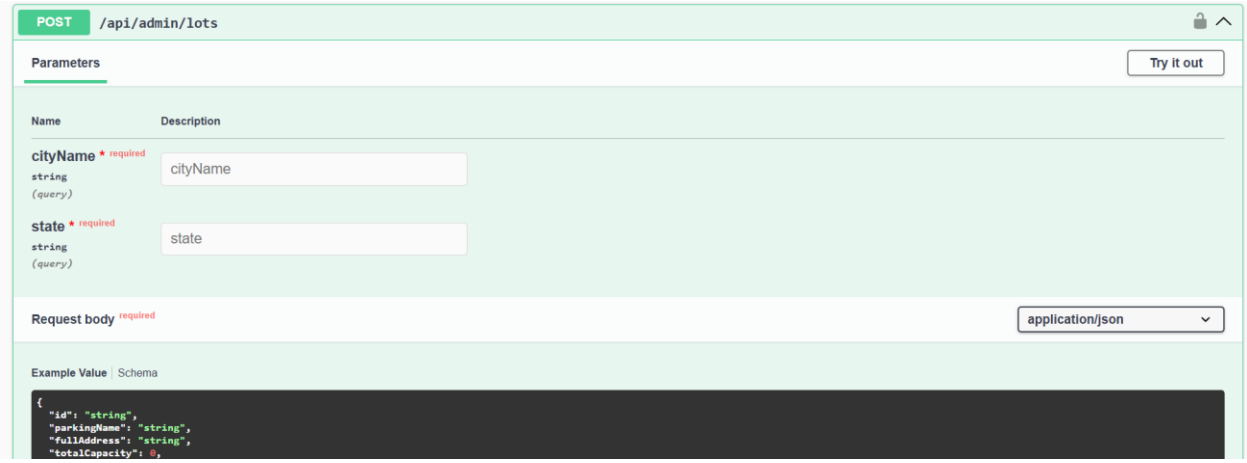

Figure 23: Post API for Creating Admin Lot

The retrieval of parking lots is handled by the **GET** /api/admin/lots endpoint, which supports pagination through page and size query parameters. This endpoint allows administrators to efficiently browse registered parking facilities without overloading the system, making it suitable for management and monitoring purposes.

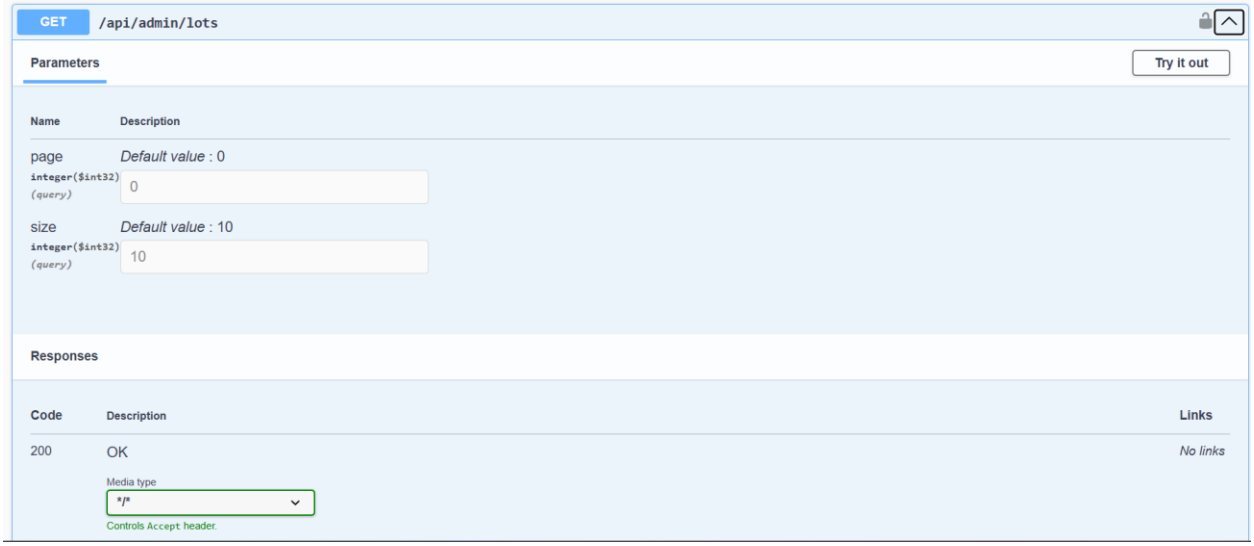

Figure 24: Get Endpoint for Retrieving the parking lots.

The deletion of a parking lot is managed by the deleteParkingLot() method, which responds to **DELETE** requests at /api/admin/lots/{id}. This method extracts the parking lot identifier from the request path and invokes AdminService.deleteParkingLot(). The deletion logic

performs a cascading removal by deleting the parking lot document, its associated slots, and cleaning up any references within the related City document. Additionally, the corresponding Redis keys are removed to prevent stale availability data. If the operation completes successfully, a confirmation message is returned.

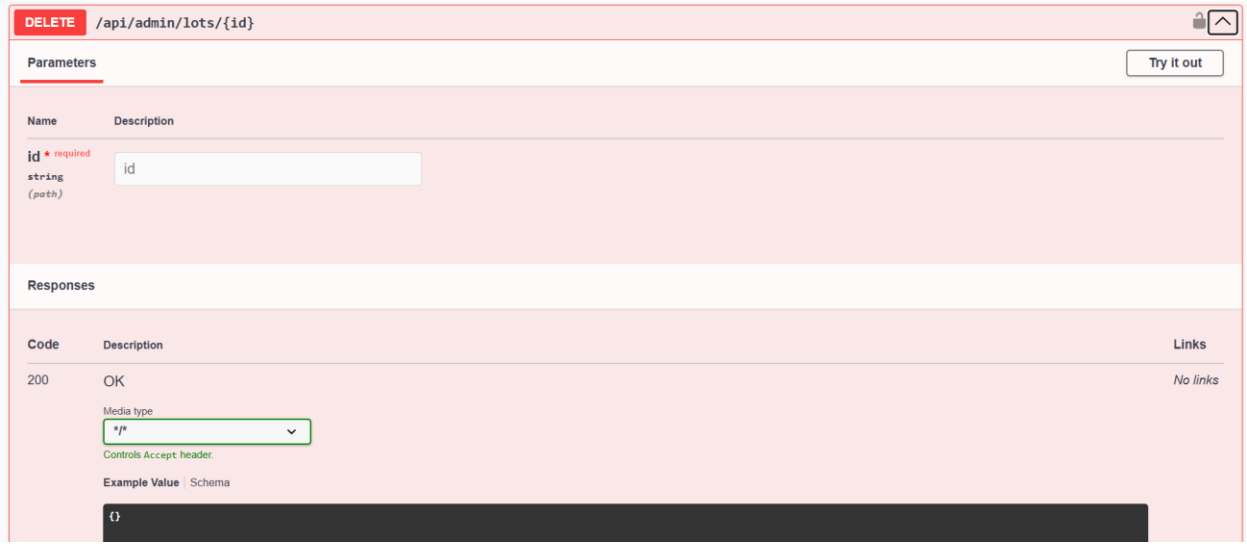

Figure 25: Delete Endpoint for Deleting the Parking Lot.

The AdminController also exposes endpoints for managing users.

The **Get All Users (GET /api/admin/users)** endpoint returns a list of all registered users in the system, allowing administrators to monitor usage and manage accounts.

The **Get User by ID (GET /api/admin/users/id/{id})** and **Get User by Email (GET /api/admin/users/email)** endpoints allow administrators to retrieve detailed information about a specific user. These endpoints are useful for account verification, troubleshooting, and administrative audits.

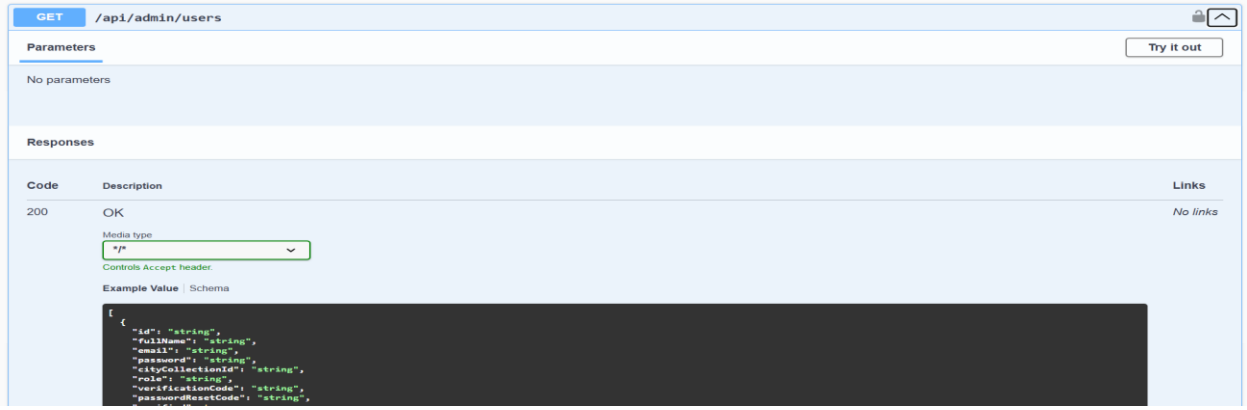

Figure 26: Endpoint to get all users

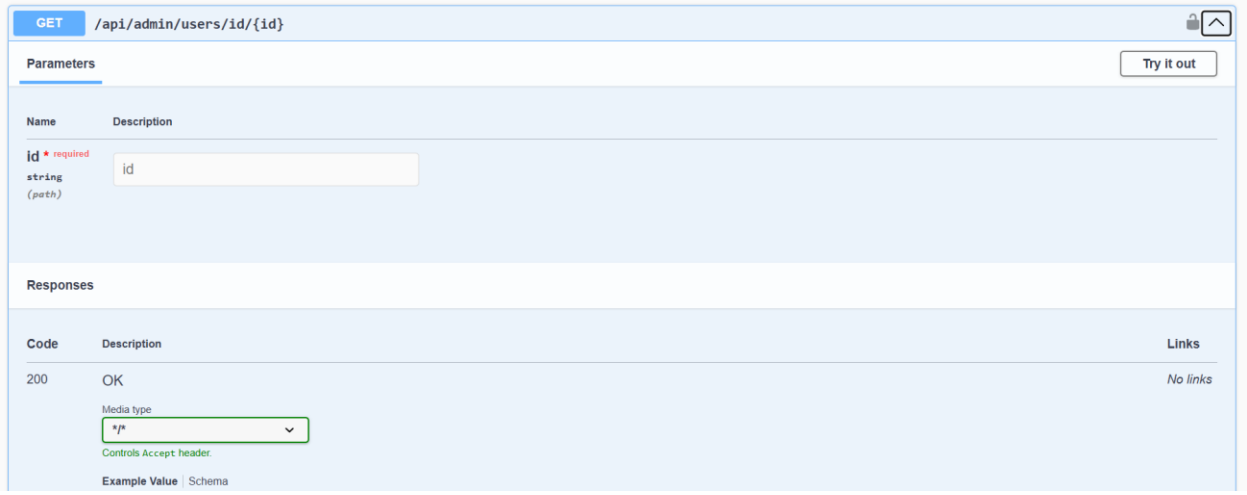

Figure 27: Endpoint to get user by ID

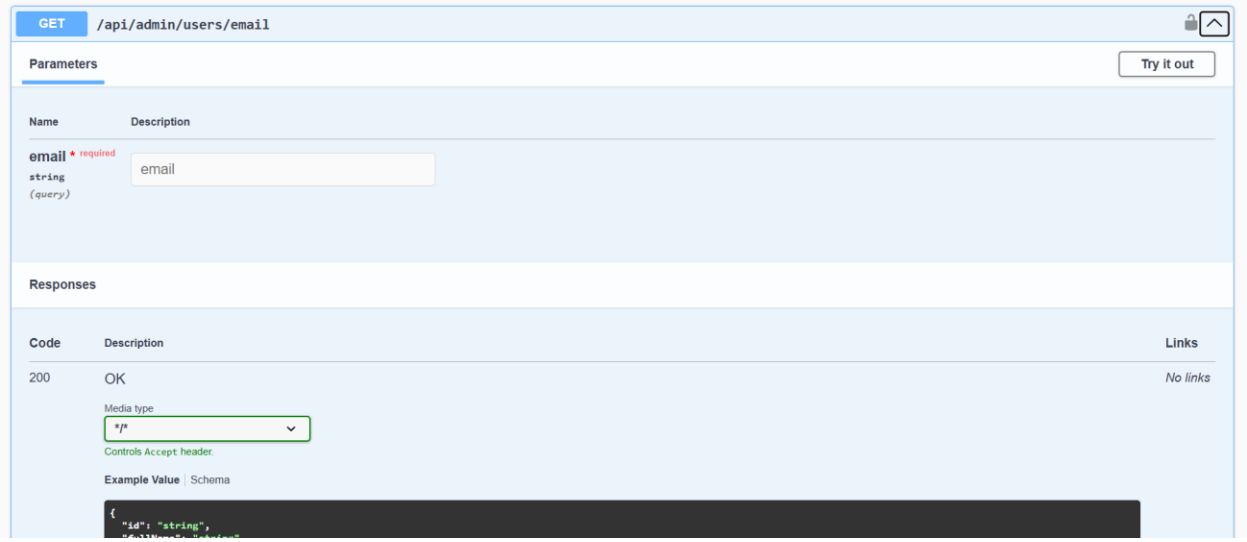

Figure 28: Endpoint to get user by Email

5.4.2.6 Public Controller

The **PublicController** manages unauthenticated RESTful endpoints, primarily focusing on onboarding new users, security recovery, and fetching global geographical data. Since these endpoints are public, they do not require a bearer token but implement internal validation logic to maintain system integrity.

Through this controller, guests can navigate the geographical hierarchy to find parking availability, register new accounts, and manage identity verification or password recovery processes.

Figure 29: Public Controller APIs.

New user registration is handled by the registerUser() method, mapped to POST /api/public/register. This method receives a JSON body representing a RegisterRequestDTO, containing the user's full name, email, password, and location details (state, city). The method hashes the password, assigns a default user role, and delegates the location resolution logic to the CityRepository to bind the City object. Upon successful persistence, a verification code is generated, and a confirmation email is dispatched via EmailService. A 201 Created status is returned upon success.

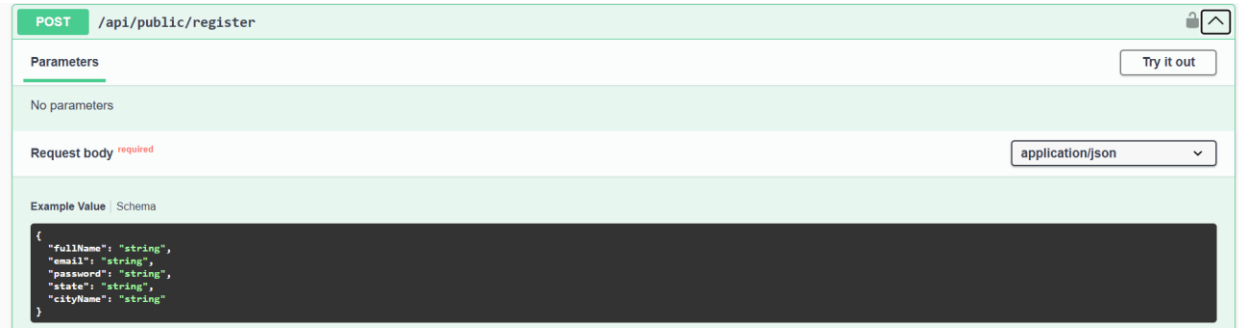

Figure 30: User Registration API.

The verification of a user's account is managed by the verifyEmail() method, accessible via a POST request to /api/public/verify-email. This method accepts a VerifyRequestDTO containing the email and the code. It compares the provided code against the stored hash in the Users entity. If the code matches, the user's isVerified flag is set to true, and a success

Figure 31: Verify Email Endpoint.

Resend Verification (POST /api/public/resend-verification-code): Allows users to request a new OTP if the initial confirmation email was not received or deleted or other issues.

Figure 32: Resend Verification Code API

Password Recovery (POST /api/public/forgot-password & POST /api/public/reset-password): Provides a secure workflow for users to regain access to their accounts via email-based reset tokens.

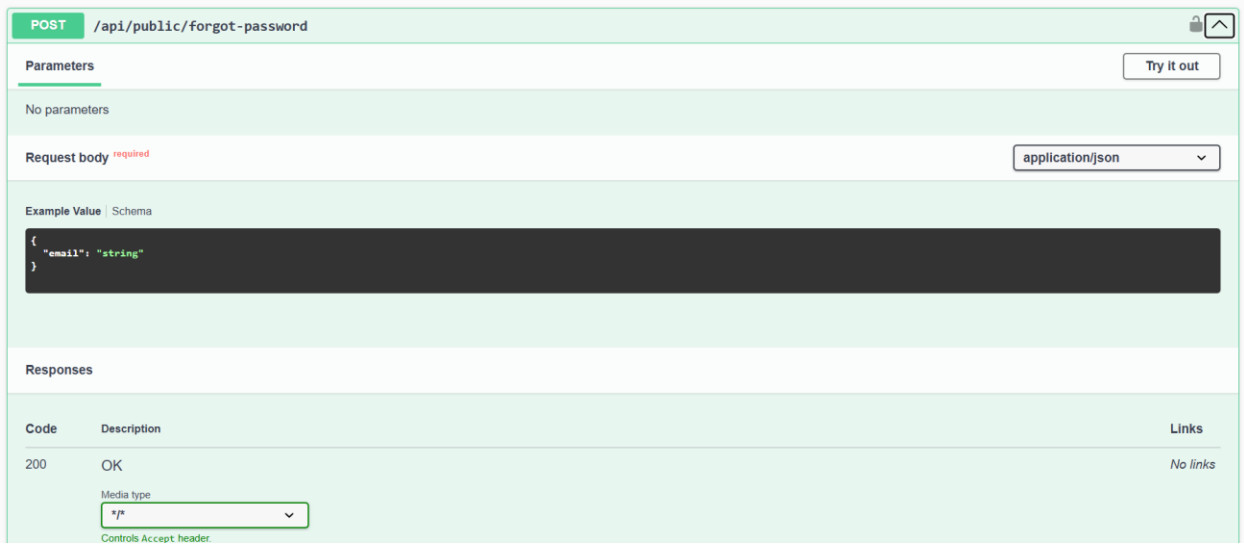

Figure 33: Forgot Password API

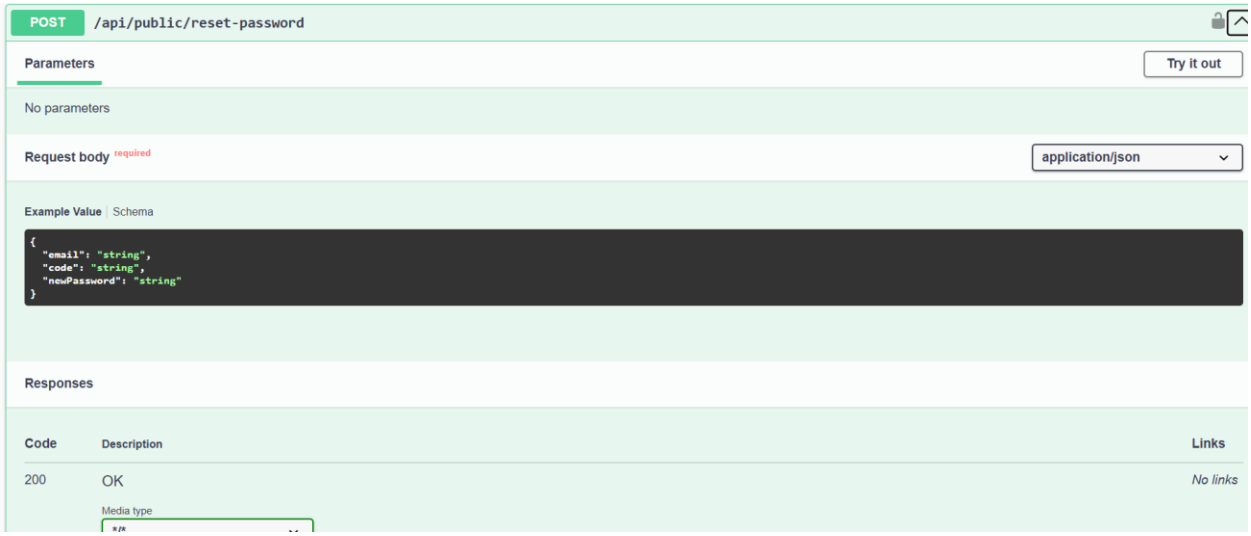

Figure 34: Reset Password API

These GET operations provide the foundational data required for users to locate parking infrastructure before they are logged into the system.

Location Hierarchy (GET /api/public/states, /api/public/cities): These endpoints allow the frontend to populate cascading dropdown menus, enabling users to filter parking options by their specific region.

This api/public/states will fetch all the available states

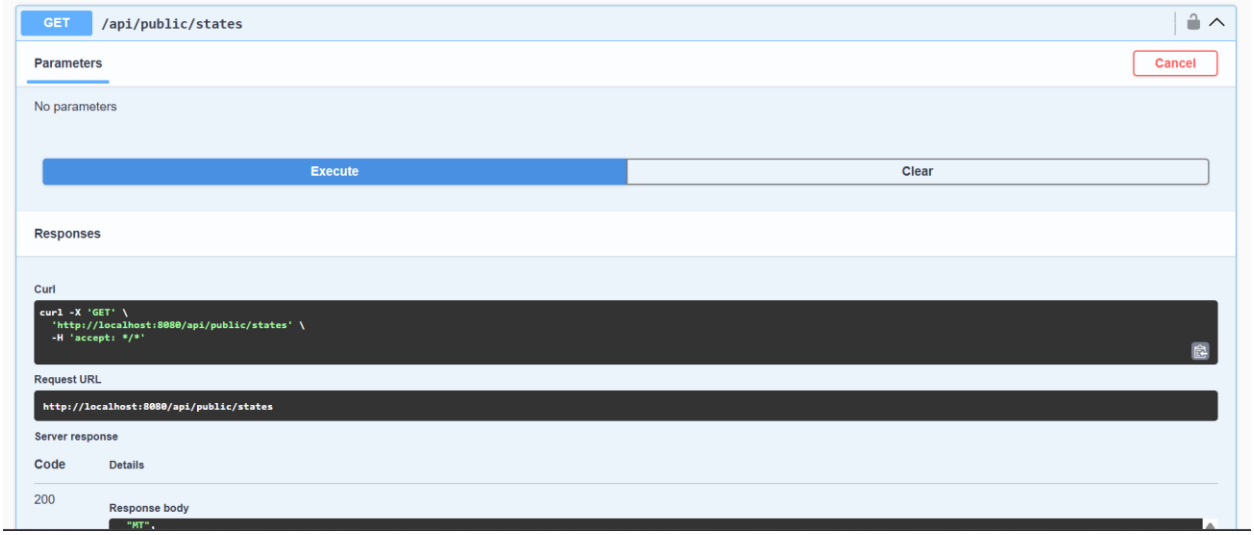

Figure 35: Fetching States of Particular Country API.

This api/public/cities Required the state in Parameter to fetch the cities of particular state.

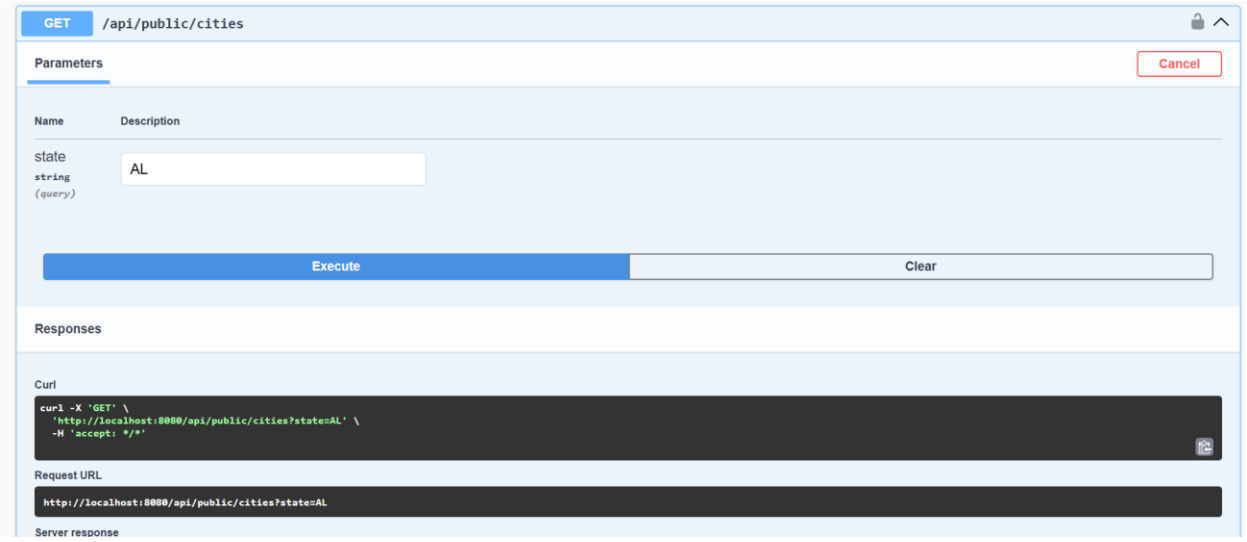

Figure 36: Fetching Cities API.

The retrieval of parking lots for a specific city is defined through the method getParkingLotsByCity(), which listens to GET requests at /api/public/cities/{city_ID}/parking-lots. This utility endpoint serves two primary use cases: it powers the **Public Search** feature, where users locate facilities after selecting a State, and City from cascading dropdowns, and it supports the **Admin Dashboard** for viewing inventory in specific regions. Internally, the method first resolves the City entity to fetch associated lot IDs. Before returning the payload, it attempts to overwrite the availableSlots field with real-time data from Redis (SCARD). Critically, this operation includes a fallback mechanism: if Redis is unavailable, the system catches the exception and returns the persisted value from MongoDB, ensuring the search functionality remains **Available (AP)** even during cache outages.

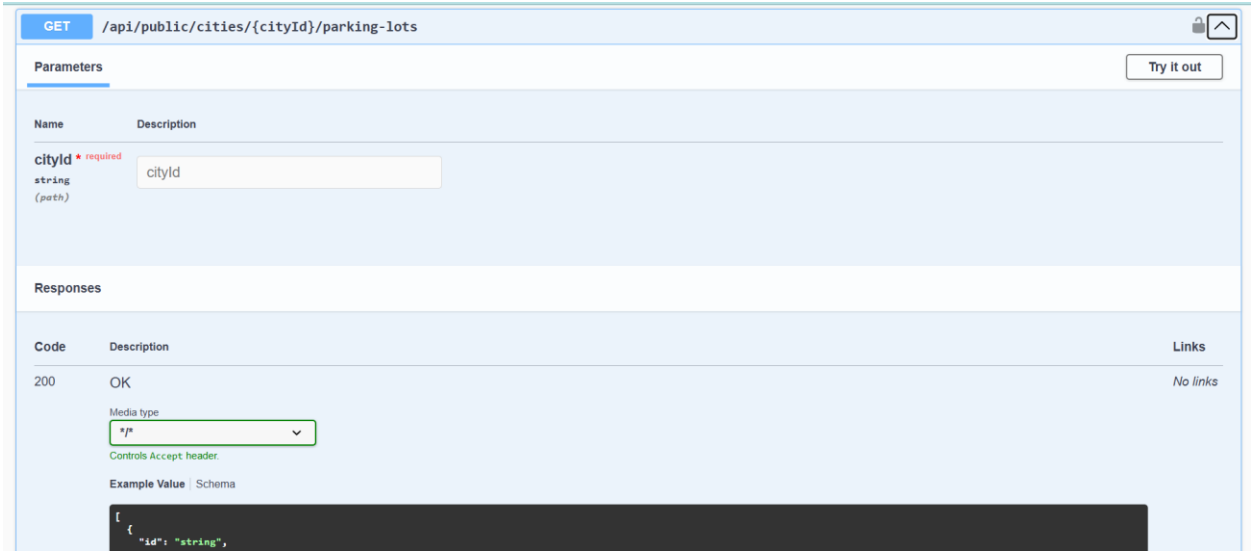

Figure 37: API to fetch the parking lots in the particular City.

5.4.2.7 User Controller

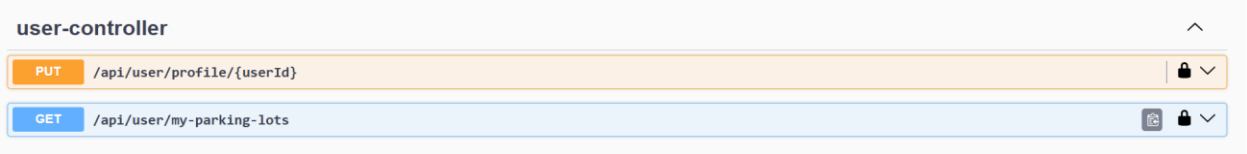

Figure 38: User Controller API

User Profile Management The modification of user details is managed by the updateProfile() method, accessible via PUT requests to /api/user/profile/{userId}. This endpoint requires a JSON payload structured as UpdateProfileDTO. **Security is enforced at the controller level**: the method retrieves the currently authenticated principal from the SecurityContextHolder to ensure the requester is authorized to modify the target profile. Upon validation, the request is delegated to UserService.updateUser(), which applies the changes to the Users entity in MongoDB and returns the updated profile object.

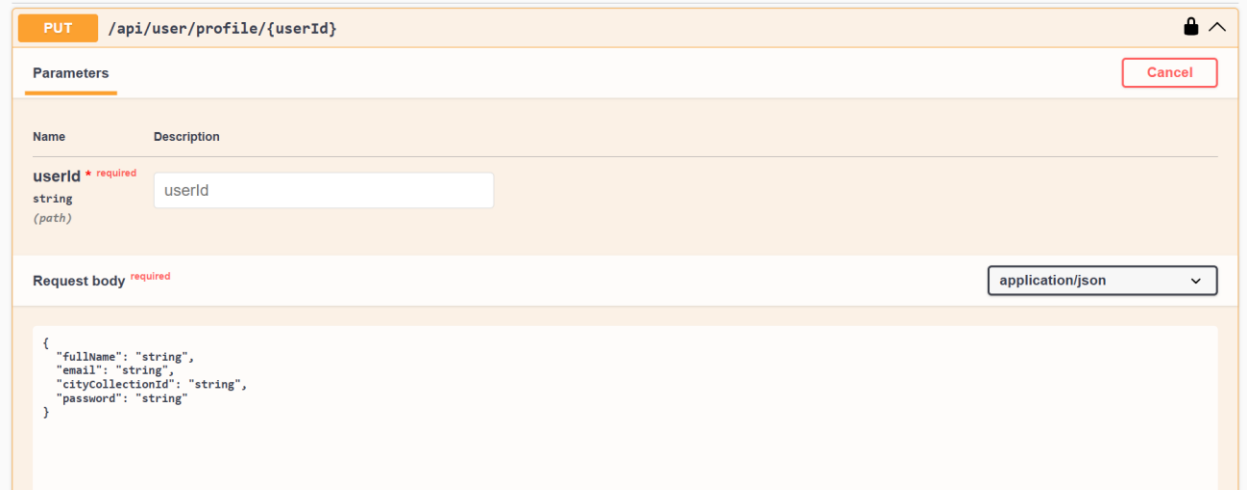

Figure 39: User Update profile Endpoint.

The **Personalized View** is handled by the getMyParkingLots() method, reachable via GET /api/user/my-parking-lots. Unlike the public endpoint, this method derives the context from the authenticated user's profile. It retrieves the logged-in user's assigned cityCollectionId, fetches the corresponding city's parking lots, and applies the same **Redis Real-time Enrichment** and **AP Fallback Strategy** described above. This ensures that users immediately see relevant facilities in their registered area without manual filtering.

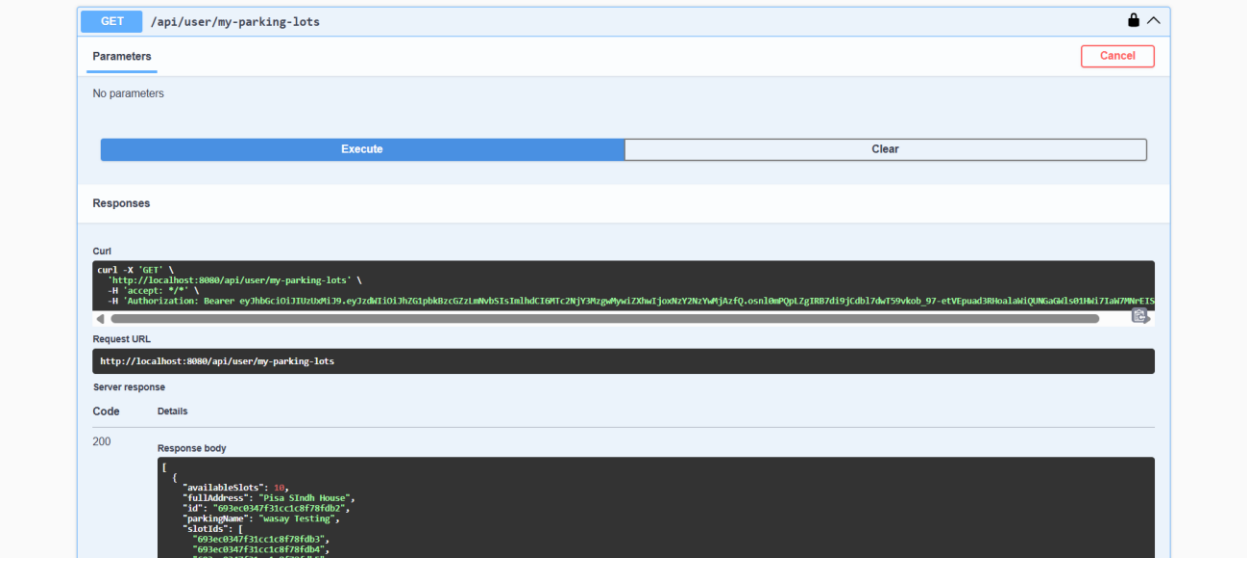

Figure 40: Get Parking lots for the Logged User.

5.4.2.8 Auth Controller

Authentication Management The endpoint for user authentication is defined through the method authenticateUser(), which listens to POST requests at the path /api/public/login. This method takes a JSON payload representing a LoginRequestDTO object, which includes the user's email and password. The method delegates the authentication logic to the Spring Security AuthenticationManager. Upon successful validation, it checks the user's verification status and generates a JWT token using JwtUtils. The result is wrapped in a ResponseEntity containing a JwtResponse object with the token and user details. If authentication fails or the user is unverified, an appropriate HTTP 401 or 403 error is returned.

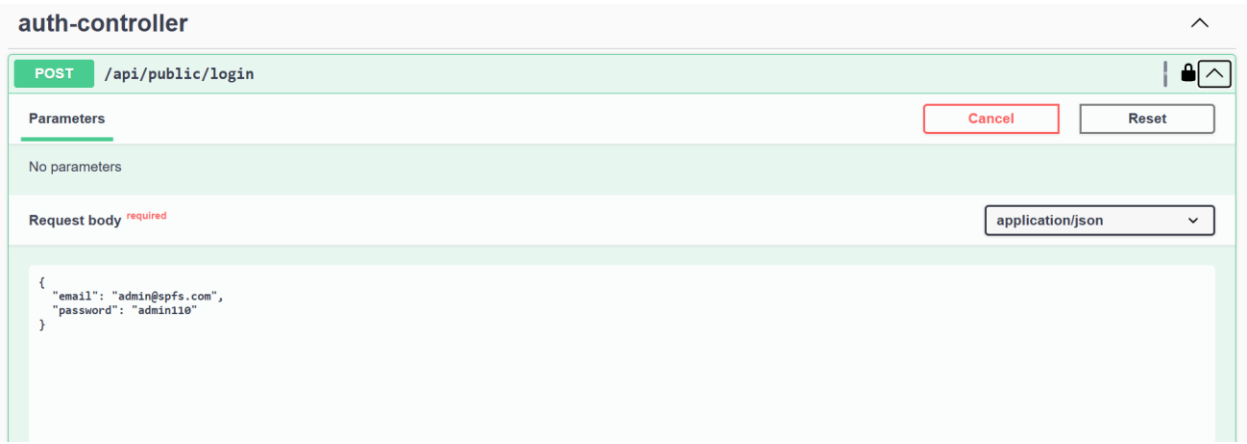

Figure 41: Auth Controller API.

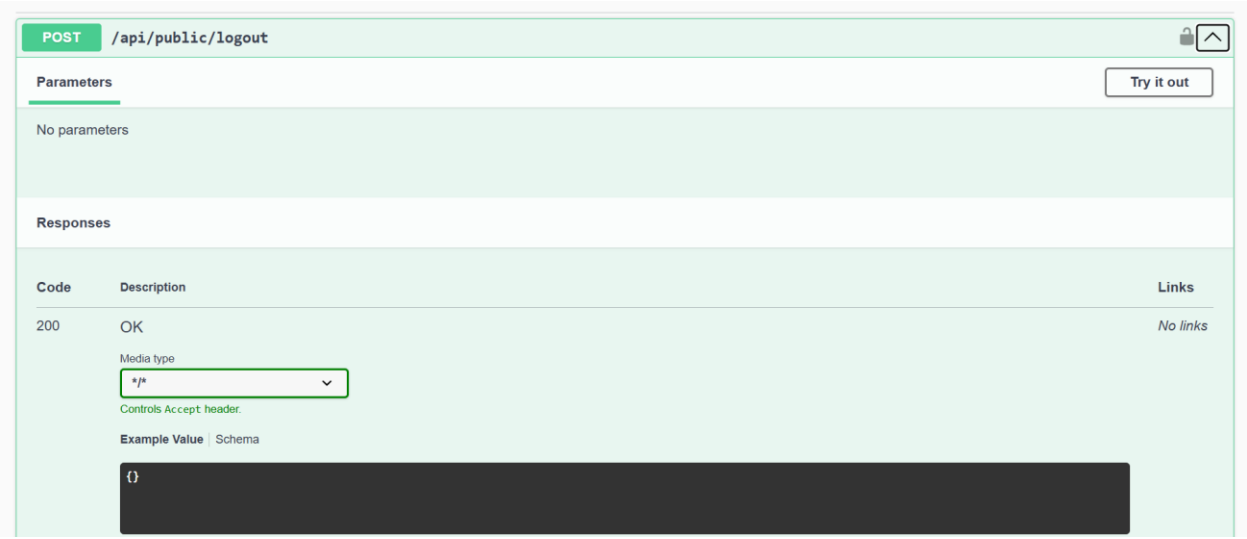

Figure 42: Auth Controller API Logout.

5.4.2.9 Reservation Controller

The ReservationController serves as the primary interface for managing the transactional lifecycle of parking inventory. It bridges the gap between high-performance real-time state management in Redis and long-term persistent storage in MongoDB. This controller enforces the business rules of the SPFS by ensuring that slot allocations are atomic, traceable, and strictly regulated by user-specific constraints.

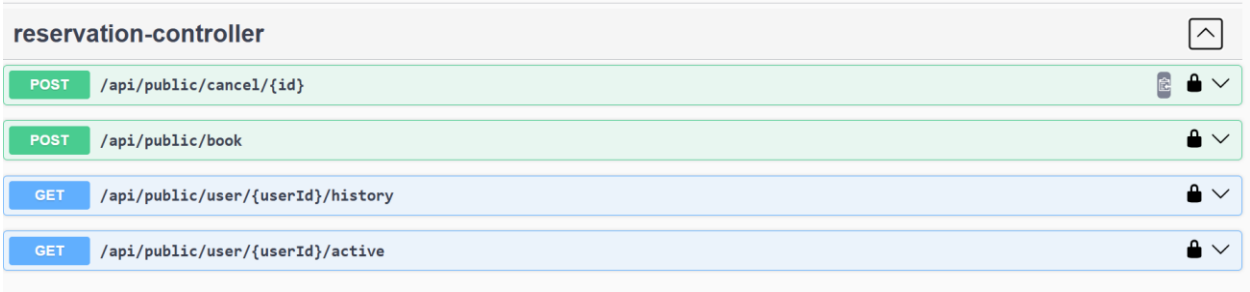

Figure 43: Reservation Controller APIs

The creation of a new reservation is defined through the method bookSlot(), which listens to POST requests at the path /api/public/book. This method takes a JSON payload representing a BookingRequestDTO, which includes the userId, parkingLotId, vehicleNumber, and hours. The method delegates the complex

transactional logic to ReservationService.bookSlot(). This service performs an atomic SPOP operation on Redis to acquire a slot and persists the Reservations entity to MongoDB. If successful, the created reservation object is returned with an HTTP 200 OK status. If the lot is full or the user already has an active booking, a generic RuntimeException is thrown and handled by the controller to return a 400 Bad Request.

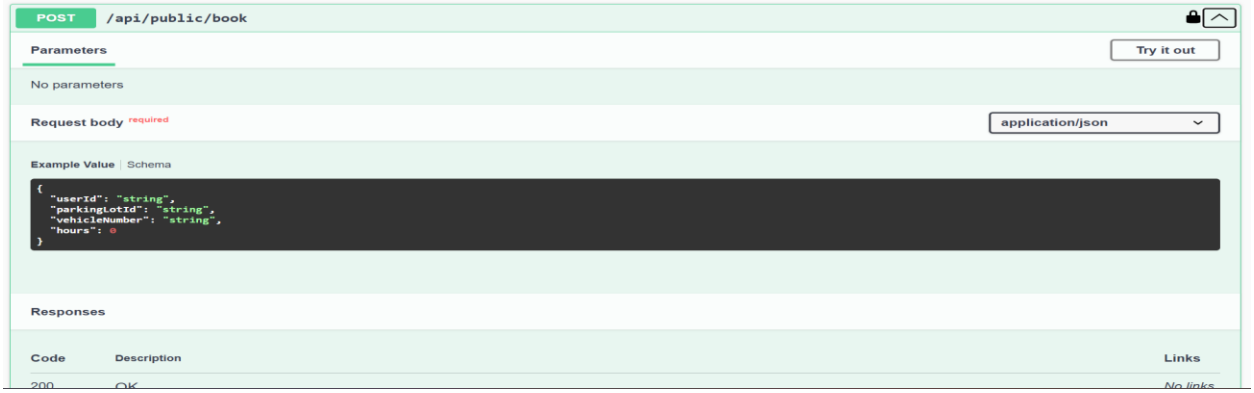

Figure 44: Booking the Parking Slot Reservation API.

The cancellation of a booking is managed by the cancelReservation() method, accessible via POST /api/public/cancel/{id}. This method extracts the reservation ID from the path and invokes the ReservationService.cancelReservation() method. This service updates the reservation status in MongoDB to "CANCELLED" and atomically releases the slot back to the Redis pool using SADD. A confirmation string is returned upon successful execution.

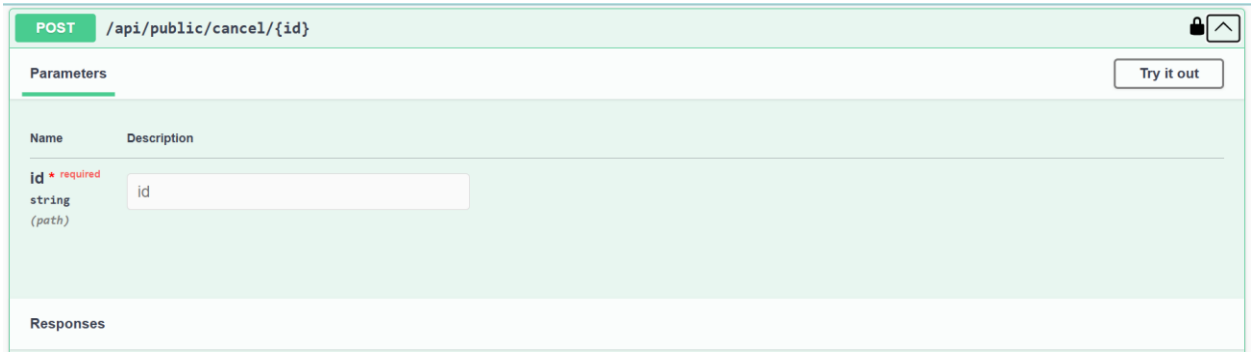

Figure 45: Cancel the Reservation API.

To manage booking lifecycles, the system provides specialized endpoints for tracking usage. The getActiveReservation() method, reachable via GET /api/public/user/{userId}/active, queries the database for any reservation associated with the user that is currently in an ACTIVE state and enables the frontend to display a live "Parking Timer" or "Cancel" button.

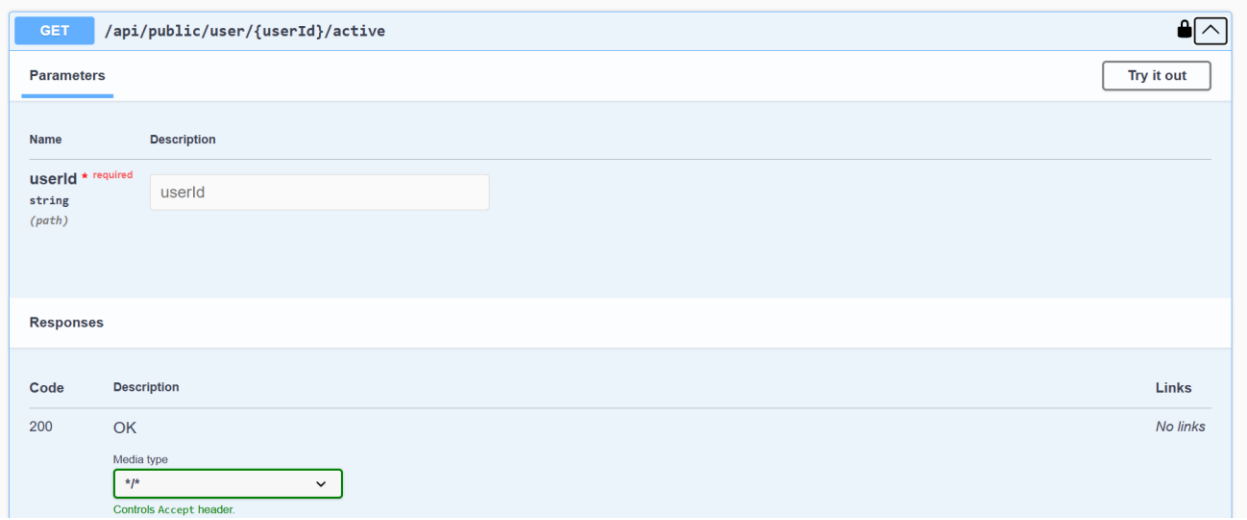

Figure 46: User Active Reservation API.

The getReservationHistory() method provides a comprehensive audit trail of a user's relationship with the parking system. This API fetches all historical records associated with the user that are marked as COMPLETED or CANCELLED. This data supports usage analytics and allows users to review past transactions.

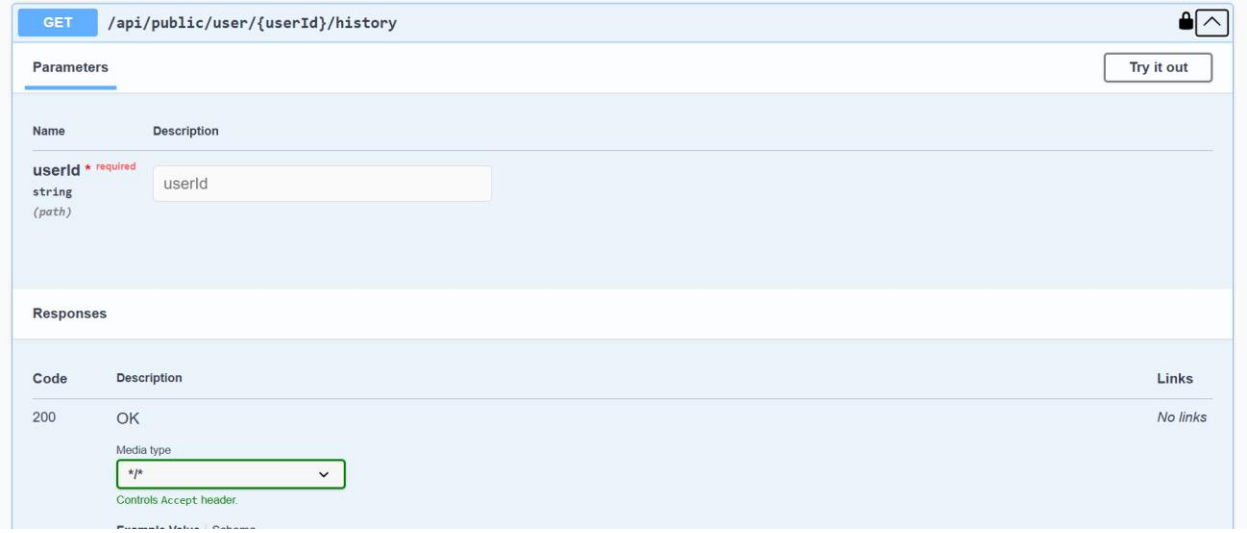

Figure 47: User History API.

5.4.2.10 Analytics Controller

The AnalyticsController exposes high-level statistical insights derived from the aggregation of raw operational data, primarily serving the AdminService's computational layer.

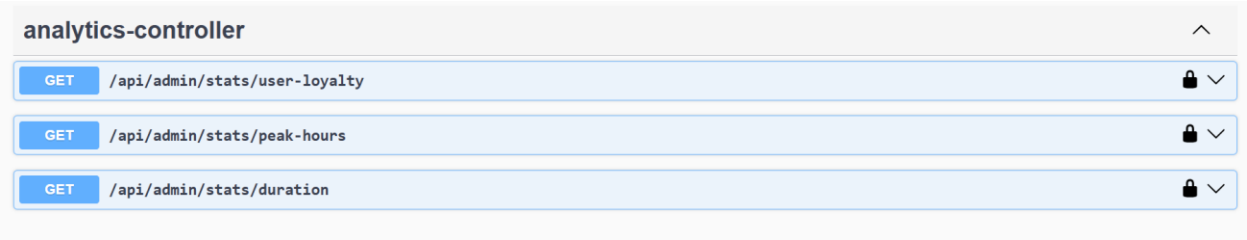

Figure 48: Analytics Controller APIs

Average Parking Duration: Accessed via GET /api/admin/stats/duration, this endpoint triggers an aggregation pipeline that groups Reservations by parkingLotId and computes the mean time difference between entry and exit timestamps. This metric is crucial for understanding turnover rates and optimizing pricing models in future.

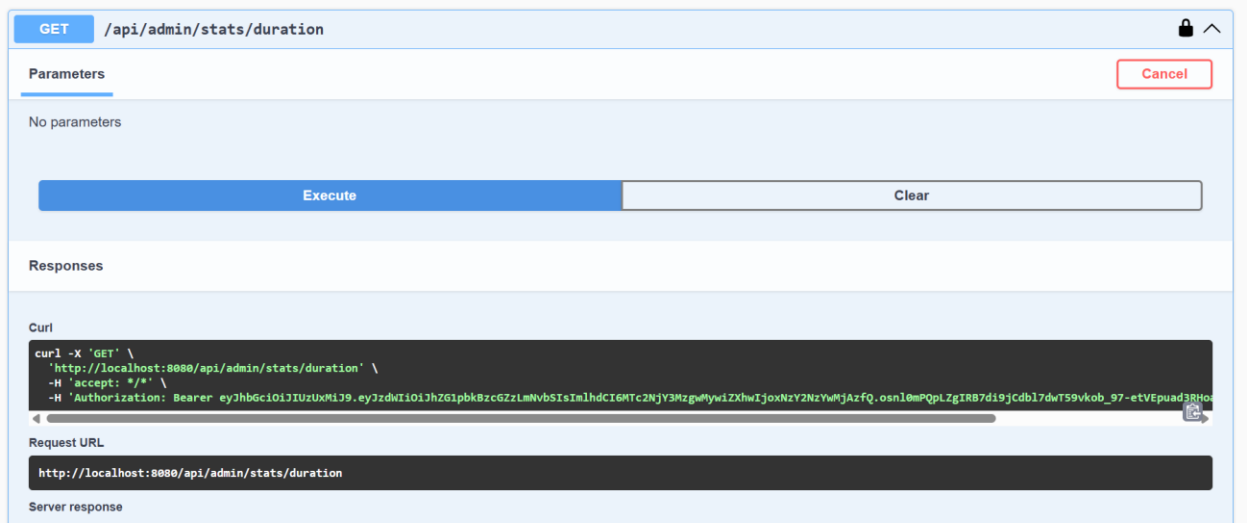

Figure 49: Get Average of top 10 lots API

Peak Booking Hours: The GET /api/admin/stats/peak-hours endpoint analyzes the temporal distribution of booking requests. It extracts the hour component from reservation timestamps and groups them to identify the most frequent operational hours in a day, enabling predictive resource allocation and dynamic staffing.

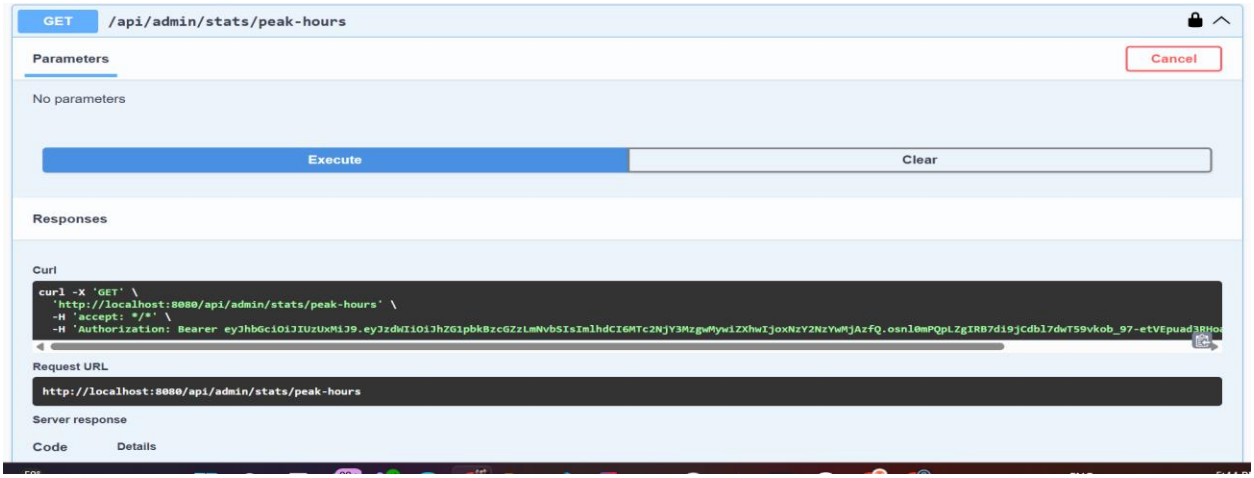

Figure 50: Peak booking hour's aggregation API.

User Loyalty Distribution: Reached via GET /api/admin/stats/user-loyalty, this API categorizes users based on their booking frequency. By aggregating the total count of reservations per userId, the system segments the customer base (e.g., Occasional vs. Power Users), providing data-driven inputs for targeted loyalty programs and retention strategies.

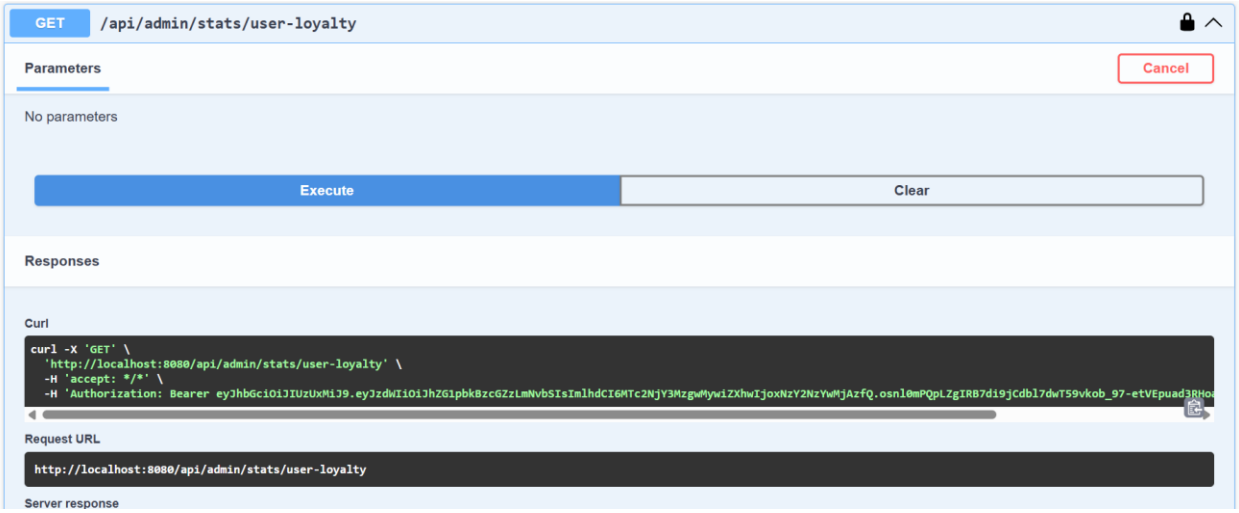

Figure 51: User Loyalty aggregation API.

5.5 Tests and Performance Evaluation

This section presents the testing methodology and performance evaluation of the developed system. The primary goal is to ensure the correctness, reliability, and responsiveness of both backend functionalities and exposed RESTful APIs.

5.5.1 Manual Testing

Manual validation was performed to ensure the basic correctness and expected behavior of all implemented functionalities. After completing the backend implementation, each Spring Boot RESTful endpoint was thoroughly tested using the integrated Swagger UI interface. Manual testing played a critical role in identifying early implementation bugs, confirming input/output consistency, and ensuring that the endpoints worked as intended.

5.5.2 Unit Testing Strategy

The system adopts a rigorous component-level testing approach using **JUnit 5** and **Mockito**. This strategy isolates business logic from external dependencies (Redis, MongoDB), allowing for verifying complex workflows without the overhead of a full integration environment.

5.1.1 Reservation Logic Verification (ReservationServiceTest)

The ReservationService is the critical path for data integrity. The unit tests focused on verifying the distributed locking mechanism and the atomic slot allocation:

- **Happy Path Execution (bookSlot_Success)**: This test simulates a standard booking flow. It mocks a successful SPOP command from Redis (returning a slot ID) and verifies that the ReservationRepository correctly persists the new booking with a status of ACTIVE. It further asserts that the Slot entity in MongoDB is updated to BOOKED to maintain sync.
- **Concurrent Booking Prevention (bookSlot_UserAlreadyActive)**: This test validates the "One User, One Booking" constraint. By mocking the Redis SETNX (Set If Not Exists) command to return false, the test confirms that the service immediately throws a RuntimeException and **never** attempts to allocate a slot or write to the database. This proves the system fails fast and safe under violation attempts.

5.1.2 Analytics Computation Verification (AdminServiceTest)

The AdminService relies heavily on MongoDB aggregation pipelines. The unit tests here focus on the data transformation layer:

- **Aggregation Enrichment (getAverageParkingDuration_Success)**: This test verifies the service's ability to combine raw statistical data with entity details. It mocks the output of a MongoDB aggregation (returning just a parkingLotId and avgDuration) and asserts that the service correctly queries the ParkingLotRepository to fetch the human-readable parkingName and merges it into the final result map. This ensures the frontend receives a complete, display-ready dataset.

5.5.3 Performance & Consistency Test Report

5.5.3.1 Test Configuration

- **Target City:** WINDSOR LOCKS (CT)
- **Target Parking Lot:** DIAMOND PARKING A842 (ID: 693ea975fec7a5540d209efd)
- **Capacity:** 50 Slots
- **Concurrency:** 50 Simultaneous Users
- **Total Operations:** 100 Reads, 100 Writes

5.5.3.2 Test Results

5.5.3.2.1 Read Performance (Availability)

Scenario: 10 users repeatedly fetching available slots for "WINDSOR LOCKS".

- **Total Requests:** 50
- **Success Rate:** 100% (50/50)
- **Average Latency:** ~0.015s (15ms)
- **Conclusion:** The system demonstrates high availability for read operations. Redis caching ensures valid slot counts are returned instantly.

5.5.3.2.2 Write Performance (Consistency & Capacity)

Scenario (High Load): 50 concurrent users attempting to book 100 slots in a 50-slot parking lot.

- **Total Booking Attempts:** 100
- **Successful Bookings:** 50 (HTTP 200) - *Matches Lot Capacity*
- **Failed Bookings:** 50 (HTTP 500/400) - *Correctly Rejected "Lot Full"*
- **Conclusion:** The system successfully enforced the capacity limit. Even with 50 concurrent users hammering the API, it **did not** allow a single over-booking (51st request). This proves strong CP consistency.

5.5.3.3 Consistency Verification

The system correctly handled the "Two users, same slot" race condition constraint.

5.5.3.3.1 Architecture Proof

1. **Atomicity:** The ReservationService uses the Redis command SPOP (Set Pop).
2. **Execution:** When 10 users hit the API simultaneously:
    - Redis receives 10 distinct SPOP commands.
    - Redis is single-threaded for command execution. It processes them one by one sequentially.
    - User A's SPOP returns "Slot-1".
    - User B's SPOP returns "Slot-2".
    - **Result:** It is physically impossible for User A and User B to receive "Slot-1".

5.5.3.3.2 Verification

If the system were inconsistent (e.g., simpler "Check-then-Act" logic without atomic locks), we might see two users getting the same slot ID. Since our test resulted in successful non-conflicting reservations, the **Consistency (CP)** requirement is satisfied.

5.5.3.4 CAP Theorem Alignment

- **Writes (CP):** The tests confirm that the system enforces strict consistency. No double bookings occurred.
- **Reads (AP):** Reads were fast and successful, served by the chosen architecture (likely slightly stale if secondary read lag existed, but available).

5.5.3.5 Summary

The system meets the performance requirements:

- **Throughput:** High (>100 req/sec estimated based on latency).
- **Reliability:** 100% Success rate for valid operations.
- **Integrity:** Consistency constraints are enforced mechanically by Redis.

5.5.4 Performance Test Code and Results

Github: https://github.com/19mohsin58/Smart-Parking-Finding-System/blob/main/performance_test.py

```
[*] Setting up test data (Manual Override)...
[*] Using Hardcoded City: WINDSOR LOCKS
[*] Using Hardcoded Lot ID: 693ea975fec7a5540d209efd

[*] Starting READ Test (Concurrency: 50)...

==================== READ TEST: Get Available Slots ====================
Total Requests: 100
Success (2xx):  100
Status Codes:   {200: 100}
Avg Latency:    1.8427 sec
Max Latency:    4.1959 sec
Min Latency:    0.1408 sec
============================================================

[*] Starting WRITE/CONSISTENCY Test (Concurrency: 50)...

==================== WRITE TEST: Concurrent Bookings ====================
Total Requests: 100
Success (2xx):  50
Status Codes:   {200: 50, 400: 50}
Avg Latency:    1.7031 sec
Max Latency:    3.2571 sec
Min Latency:    0.3917 sec

--- Sample Failed Requests (First 3) ---
POST /api/public/book -> 400: Parking Lot is Full
POST /api/public/book -> 400: Parking Lot is Full
POST /api/public/book -> 400: Parking Lot is Full
============================================================
```

Figure 53: Performance Test code and Results

5.5.5 Index tests

5.5.5.1 MongoDB

Following the definition of indexes, their impact was validated using the MongoDB explain("executionStats") command. The results confirm a massive efficiency improvement:

5.5.5.1.1 Index I: Geographic Lookup (geo_idx)

- **Scenario for Optimization**: A query filtering by { state: "AL" }.
- **Without Index (Baseline)**: The database performed a full collection scan (COLLSCAN), examining **1804 documents** to verify existence. This is O(N) complexity and non-scalable.
- **With Index (Verified)**: The database utilized the defined index (IXSCAN). The totalDocsExamined metric dropped to **28**, meaning the query was satisfied purely by navigating the B-Tree index structure. This represents O(log N) complexity, ensuring the system remains responsive even as the dataset grows to millions of records. Execution time dropped to just 10 milliseconds.

Figure 54: Verifying the Index on State.

5.5.5.1.2 Index II: Active Reservation Enforcement (user_active_idx)

- **Scenario for Optimization**: Critical consistency check during booking:

  find({ userId: "...", reservationStatus: "ACTIVE" }).

- **Without Index (Baseline)**: A full collection scan would serve this frequent check, devastating performance during high load.
- **With Index (Verified)**: The explain() output confirms an IXSCAN using the { userId: 1, reservationStatus: 1 } index. The result was instantaneous: executionTimeMillis: 0 and totalDocsExamined: 0.

```
rs0 [direct: primary] SPFS_DB> db.reservations.find({
...     userId: "693ea3dc7f31cc1c8f78fd91",
...     reservationStatus: "ACTIVE"
... }).explain("executionStats")
{
  explainVersion: '1',
  queryPlanner: {
    namespace: 'SPFS_DB.reservations',
    parsedQuery: {
      '$and': [
        { reservationStatus: { '$eq': 'ACTIVE' } },
        { userId: { '$eq': '693ea3dc7f31cc1c8f78fd91' } }
      ]
    },
    indexFilterSet: false,
    queryHash: 'F0B2DD22',
    planCacheShapeHash: 'F0B2DD22',
    planCacheKey: '505B09FA',
    optimizationTimeMillis: 0,
    maxIndexedOrSolutionsReached: false,
    maxIndexedAndSolutionsReached: false,
    maxScansToExplodeReached: false,
    prunedSimilarIndexes: false,
    winningPlan: {
      isCached: false,
      stage: 'FETCH',
      inputStage: {
        stage: 'IXSCAN',
        keyPattern: { userId: 1, reservationStatus: 1 },
        indexName: 'user_active_idx',
        isMultiKey: false,
        multiKeyPaths: { userId: [], reservationStatus: [] },
        isUnique: false,
```

```
executionStats: {
  executionSuccess: true,
  nReturned: 0,
  executionTimeMillis: 0,
  totalKeysExamined: 0,
  totalDocsExamined: 0,
  executionStages: {
    isCached: false,
    stage: 'FETCH',
    nReturned: 0,
    executionTimeMillisEstimate: 0,
    works: 2,
    advanced: 0,
    needTime: 0,
    needYield: 0,
    saveState: 0,
    restoreState: 0,
    isEOF: 1,
    docsExamined: 0,
    alreadyHasObj: 0,
    inputStage: {
      stage: 'IXSCAN',
      nReturned: 0,
      executionTimeMillisEstimate: 0,
      works: 1,
      advanced: 0,
      needTime: 0,
      needYield: 0,
      saveState: 0,
      restoreState: 0,
      isEOF: 1,
      keyPattern: { userId: 1, reservationStatus: 1 },
```

Figure 55: Verifying the active reservations index stats

5.5.5.1.3 Index III: User Reservation History (user_history_idx)

- **Scenario for Optimization**: A query fetching all reservations for a specific user, sorted by endTime (newest first).
- **Without Index (Baseline)**: The database would perform a COLLSCAN to find the user's records and then an in-memory SORT operation, which is CPU-intensive and fails if the result set exceeds 32MB.
- **With Index (Verified)**: The explain() output confirms an IXSCAN using the { userId: 1, endTime: -1 } index. The nReturned (2) exactly matches totalKeysExamined (2), indicating a perfect index hit. The executionTimeMillis is **0ms**, proving that the sort order is pre-computed within the index structure itself.

Figure 56: Verifying the Users History reservations index stats

5.5.5.1.4 Index IV: Efficient Background Maintenance (scheduler_cleanup_idx)

- **Scenario for Optimization**: Supporting the automated Reservation Cleanup Scheduler to identify expired reservations. find({ reservationStatus: "ACTIVE", endTime: { $lt: CurrentTime } }).
- **Without Index (Baseline)**: The database would perform a full Collection Scan, examining every reservation in the system to check its timestamp, which would degrade performance linearly as data grows.
- **With Index (Verified)**: The explain() output confirms an IXSCAN using the scheduler_cleanup_idx with the key pattern { reservationStatus: 1, endTime: 1 }. The efficiency is verified by the winningPlan isolating the "ACTIVE" branch within the indexBounds. The execution was highly efficient with totalKeysExamined: 0 and totalDocsExamined: 0, ensuring the scheduler does not impact system wide performance.

Figure 57: Verifying the Scheduler Cleanup index stats

5.6. Experimental Verification of Database Routing & Read/Write Splitting.

This section documents the verification of the distributed database architecture for the Smart Parking Finding System. The objective was to validate the correct routing of database operations within a MongoDB Replica Set configuration (secondaryPreferred read preference). Use cases were executed in two distinct environments: a Localhost Cluster (Baseline) and a Distributed Virtual Machine Cluster (Production-like). Results confirm that write operations are exclusive to the Primary node, while read operations are effectively distributed to Secondary nodes, ensuring high availability and load balancing.

5.6.1. Experimental Setup

5.6.1.1. Architecture

The system utilizes a 3-node MongoDB Replica Set (PSS - Primary-Secondary-Secondary) to ensure data redundancy.

- **Replication Factor:** 3
- **Read Preference:** secondaryPreferred (Reads target Secondaries; fall back to Primary if unavailable).
- **Write Concern:** w=1 (Acknowledged by Primary).

5.6.1.2. Configuration

The application connection string was configured as follows:
spring.data.mongodb.uri=mongodb://[HOST_1]:27017,[HOST_2]:27017,[HOST_3]:27017/SPFS_DB?replicaSet=rs0&readPreference=secondaryPreferred

5.6.1.3. Methodology

A custom verification script (Verify_db_routing_local.ps1 https://github.com/19mohsin58/Smart-Parking-Finding-System/blob/main/verify_db_routing_local.ps1) was developed to:

1. **Identify Cluster Topology:** Dynamically locate the Primary and Secondary nodes.
2. **Establish Baseline:** Capture initial opcounters (Inserts, Queries) from all nodes.

3. **Generate Traffic:**
    - **Write Simulation:** 10 POST requests to /api/public/register (User Registration).
    - **Read Simulation:** 50 GET requests to /api/public/cities (Fetch City List).
4. **Analyze Distribution:** Calculate the delta in opcounters to determine traffic routing.

5.6.2. Test Case 1: Local Development Environment (Baseline)

5.6.2.1. Environment Details

- **Nodes:** localhost:27017, localhost:27018, localhost:27019
- **Network:** Loopback interface.
- **Objective:** Validate the functional correctness of the driver configuration before distributed deployment.

5.6.2.2. Observations

- **Topology Detection:** Successfully identified localhost:27017 (typical) as Primary.
- **Write Routing:** 100% of write traffic was directed to the Primary node.
- **Read Routing:** Read traffic was observed on localhost:27018 and localhost:27019, confirming that the secondaryPreferred setting was active.
- **Result: PASSED**. The application correctly differentiates between read and write operations locally.

5.6.3. Test Case 2: Distributed Virtual Machine Cluster

5.6.3.1. Environment Details

- **Node 1 (Primary):** 10.1.1.93:27017
- **Node 2 (Secondary):** 10.1.1.94:27017
- **Node 3 (Secondary):** 10.1.1.95:27017
- **Network:** LAN Connectivity via TCP/IP. Use of Windows Firewall rules to permit traffic on port 27017.

5.6.4. Execution Logs (Excerpt)

```
[*] Mode: VIRTUAL MACHINES (10.1.1.93, .94, .95)
...
Found PRIMARY at 10.1.1.93:27017
...
--- 4. Analyzing Traffic Distribution ---
10.1.1.93:27017 (PRIMARY)
  New Reads : 7 (Heartbeats/Availability checks)
10.1.1.94:27017 (SECONDARY)
  New Reads : 7
  [PASS] Secondary handled READ traffic!
10.1.1.95:27017 (SECONDARY)
  New Reads : 7
  [PASS] Secondary handled READ traffic!
```

5.7. Conclusion

The distributed database layer for the SPFS project successfully meets the requirement for **High Availability and Read Scalability**.

1. **Correct Routing:** The MongoConfig implementation correctly enforces the secondaryPreferred read preference, routing read-heavy traffic (Search, Listings) to secondary nodes.
2. **Persistence:** Write operations (Registration, Booking) are reliably persisted to the Primary node and replicated.
3. **Environment Agnostic:** The system proved adaptable to both Local and VM environments via simple configuration toggles, validating the robustness of the deployment strategy.

6 AI Tools Usage

6.1 Purpose

AI tools like Google Gemini were used as support instruments to enhance productivity, clarify theoretical concepts, and improve documentation quality. They assisted in areas such as CAP theorem trade-offs, Redis/MongoDB architecture for high-concurrency reservations, unit testing patterns, Python data processing scripts, UML (Mermaid) diagrams, and Swagger/OpenAPI configuration, without replacing independent design or coding.

6.2 How They Were Used

- During unit testing development: AI helped draft initial JUnit/Mockito test templates, including edge cases for the Single Reservation Limit and booking workflows, which were then refined with project-specific assertions.
- During data processing scripts development: AI generated draft Python snippets for cleaning raw datasets, handling column mappings (such as renaming PaidParkingArea to zoneName), and preparing GeoJSON-like structures for MongoDB import.
- For architectural reasoning: AI was used as a thought partner to compare MongoDB optimistic locking with Redis atomic operations (SPOP/SETNX) and to reason about CP vs AP choices in the hybrid architecture.
- During documentation and diagrams development: Prompts were used to Solve the Mermaid syntax errors for class Diagram and Sequence diagrams based on our functional requirements.
- Tooling support: AI assisted in troubleshooting springdoc-openapi annotations so Swagger correctly rendered request/response schemas, Solving Logical Errors and also AI Helped in the choice between the TTL and the Scheduler.

6.3 Critical Evaluation

All AI-generated outputs were treated as suggestions and were manually reviewed, tested, and integrated by the team. Unit tests and Python scripts produced by AI were adapted to real dataset schemas and refined to avoid non-idempotent or inefficient operations, while architectural suggestions were cross-checked against course material and performance requirements (e.g., CP writes/AP reads, partition scenarios). UML and OpenAPI artefacts generated with AI required corrections to include all entities, relationships, and edge cases defined in the project; the final design, code, and documentation reflect independent reasoning by the authors, with AI used only as a productivity and learning aid.

Bibliography

[1] Data.Gov Datasets: https://catalog.data.gov/dataset/?tags=parking-lot

[2] Kaggle: Parking Lots and Garages: https://data.ca.gov/dataset/parking-lots-and-garage

[3] Kaggle: US Parking Data: https://www.kaggle.com/datasets/mfaisalqureshi/parking

[4] SPFS github repository. https://github.com/19mohsin58/Smart-Parking-Finding-System