Uber Rider Booking System Architecture

Service-Level Overview

An Uber-style system typically uses multiple microservices, each responsible for a distinct domain. For example, a **User (Customer/Rider) Service** handles rider profiles, signup, login and authentication. A **Driver Service** manages driver onboarding, profiles and availability. A **RideRequest Service** (or *Ride Service*) orchestrates ride matching and trip lifecycles: it receives ride requests, finds nearby drivers (using strategies), and updates ride status from requested→accepted→in_progress→completed. A **Payment Service** handles charging the rider and paying drivers – it may integrate with external gateways or internal wallets. A dedicated **Wallet Service** can manage user balances, top-ups and debits (e.g. debit funds on trip completion). A **Rating Service** stores ratings/feedback for drivers and riders after each trip. Notifications (push, SMS, email) are sent via a **Notification/Email Service** when events occur (e.g. new ride request, ride completion). An **Admin** component or dashboard provides global controls like driver approval, suspending users, and viewing all rides. Each service owns its data (database) and exposes APIs to other services or frontends.

Key responsibilities:

- **UserService**: handles user registration, login, logout. It issues auth tokens and maintains user credentials and profiles.
- **RiderService (Customer Service)**: manages rider-specific actions – viewing ride history, requesting or canceling rides, adding funds to a wallet.
- **DriverService**: manages driver registration and status. It allows drivers to go online/offline, accept/cancel rides, start and end trips, and even rate riders.
- **RideRequestService**: orchestrates ride flows. It records new ride requests, finds and notifies matching drivers (using strategies), and updates ride state (request→accept→start→end). It may compute estimated time/fare and interact with other services.
- **Payment & Wallet Services**: the Payment Service processes completed-ride payments (via credit card, Stripe, etc.) and handles payouts to drivers. The Wallet Service keeps

track of user credits – riders can *addFunds* and the system debits this wallet when rides are paid.

- **Rating Service**: after a ride, riders can *rateDriver* and drivers can *rateRider*. This service stores those reviews and averages, used for analytics.
- **Ride (or Trip) Service**: some designs include a dedicated Trip/Ride Service that maintains ride records (status, fare, route, timestamps) and generates receipts. In other architectures, ride management is part of RideRequestService.
- **Admin**: an Admin dashboard (or AdminService) is used by system operators to perform tasks like *onboardDriver* (approve new drivers) or retrieve data (e.g. *getAllRides* for reporting).

Most of the ride-booking flow ties these services together. For example, when a rider books a ride, the RideRequestService logs the request and triggers driver-matching logic; accepted rides move into the Ride service, then upon completion the Payment and Rating services are invoked. Using a microservices approach ensures each service can scale and be maintained independently.

Operation Mappings

The system's API exposes operations grouped by user roles and system functions (as suggested by the diagram). Typical mappings include:

- Rider (Customer) operations:
  - **signUp**, **login**, **logout** (handled by UserService).
  - **requestRide** (submit a new ride request with pickup/drop-off coordinates).
  - **cancelRide** (if ride not yet started).
  - **addFunds** (top-up their Wallet).
  - **rateDriver** (submit a rating for the driver after ride completion).
  - **getAllRides** (view past trip history or current rides).
- Driver operations:
  - **onboardDriver** (submit documents, done by Admin or DriverService).
  - **login/logout** and status update (online/offline via *updateLocation* calls).
  - **acceptRide** / **denyRide** (respond to incoming ride requests).
  - **startRide**, **endRide** (mark the ride as in-progress or complete).
  - **cancelRide** (driver-cancels, subject to rules).
  - **rateRider** (give feedback to the rider after trip).
  - **debitFunds** (if using wallet payments, debit any final fare from rider's wallet).
  - **getAllRides** (view all rides assigned to them or in system).

- **Admin operations** (often via a protected Admin API or UI):
  - **onboardDriver** (approve or reject new driver registrations).
  - **getAllRides** (list rides across the system, with filters).
  - **Manage users/drivers** (suspend/activate accounts).
- **System/Infrastructure operations**:
  - **updateLocation** (typically invoked by driver app periodically to update GPS location for matching).
  - **Notification**/event operations like *sendEmail* or *sendPush*: e.g. when a new ride request is created, the system might email/push to available drivers or administrators.
  - Other internal tasks (cron jobs, cache refreshes, etc.) are not part of user APIs but run in the background.

The diagram also shows system-level endpoints (e.g. location update) and background tasks. For instance, when a new ride is requested, the RideRequestService may use a **NotifyAllDrivers** operation to broadcast the request (via push or email) to relevant drivers.

Design Patterns Usage

The architecture can leverage standard design patterns for flexibility:

Strategy Pattern: This pattern defines a family of algorithms and makes them interchangeable at runtime. In our system, a DriverMatchingStrategy interface could be implemented by concrete strategies such as NearestDriverStrategy (select closest drivers) or RatingBasedStrategy (select drivers with highest rating). Similarly, a CalculateFareStrategy interface might have DefaultFareStrategy and SurgePricingFareStrategy. By choosing strategies at runtime (e.g. based on demand or configuration), the matching and fare calculations become pluggable without changing core logic. This follows the classic strategy definition: “define a family of algorithms… put each in separate class, and make them interchangeable at runtime”.

Builder Pattern: Used for constructing complex objects. For example, a RideRequest or PaymentRequest might have many fields (user, pickup, destination, options, etc.), so a fluent builder helps avoid telescoping constructors. A builder might look like new RideRequest.Builder().source(…).destination(…).fareOption(…).build(). Baeldung illustrates this: e.g.

```
Post post = new Post.Builder()

  .title("Java Builder Pattern")
```

```
    .text("...")
    .category("Programming")
    .build();
```

This simplifies object creation and makes code more readable. Similarly, DTOs (Data Transfer Objects) can use builders (or Lombok's @Builder) for ease of instantiation.

Factory Pattern: A factory provides a way to create objects without specifying the exact class. For instance, a FareStrategyFactory could return the right CalculateFareStrategy implementation based on current conditions (surge vs normal). Or a DriverMatchingStrategyFactory might produce a strategy instance based on config. The Factory Method pattern "defines an interface for creating objects but lets subclasses decide which object to instantiate". By delegating strategy instantiation to factories, the system is loosely coupled and easier to extend (new strategies can be added without altering caller code).

Singleton Pattern: Ensures a class has only one instance. In Spring Boot, most beans are singletons by default, but one might explicitly use a Singleton (e.g. a legacy ConfigManager or a shared cache manager). Singleton is useful to share common state or avoid reinitializing heavy resources. As Baeldung notes, "Singleton pattern involves a single class responsible for creating an object and ensuring that only a single instance ever gets created. We often use singletons to share state or to avoid the cost of setting up multiple objects". In practice, Spring's bean scope (singleton) typically covers most cases (e.g. a single instance of ObjectMapper or a currency conversion cache).

These patterns help keep the code modular and maintainable. For example, strategy pattern use in driver matching allows experimenting with different matching algorithms with minimal code change. A Builder or Factory can make constructing domain objects or configuration-driven behavior easier.

Swagger UI Integration

To document and expose our REST APIs, we can integrate Swagger (OpenAPI) into the Spring Boot app. A popular choice is **Springdoc OpenAPI**: by adding the springdoc-openapi-starter-webmvc-ui dependency, Spring Boot will automatically generate an interactive Swagger UI at runtime. For example, include in pom.xml:

```
<dependency>
 <groupId>org.springdoc</groupId>
 <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
 <version>2.2.0</version>
</dependency>
```

As BellSoft explains, this single dependency “helps to generate OpenAPI-compliant API documentation for Spring Boot projects” with no extra config needed. Then each controller method can be annotated with OpenAPI annotations (e.g. @Operation, @ApiResponse) to describe its parameters and responses. When the app runs, browsing to /swagger-ui.html (or /swagger-ui/index.html) will show a live, browsable interface of all endpoints, models, and example requests/responses. This makes it easy for frontend developers or external clients to discover and test the API.

Spring Boot Considerations

Profiles (DEV, PROD): Use Spring Boot’s profile support to separate configuration for development and production. For example, have application-dev.properties and application-prod.properties with different DB credentials, log levels, or API endpoints. Activate profiles via spring.profiles.active=dev (locally) or as an environment variable in production. This ensures you can run the app with embedded H2 (for dev) but use MySQL or PostgreSQL in prod, without changing code.

Testing Strategies: Adopt both unit and integration tests. For controllers and service layers, write unit tests using JUnit and Mockito (mocking dependent services). For example, annotate controllers with @WebMvcTest and use MockMvc to test HTTP requests. For service/database integration, use @SpringBootTest along with an in-memory database (H2) to test full application context. The Baeldung Spring Boot Testing guide shows using @SpringBootTest(webEnvironment=Mock) with @AutoConfigureMockMvc and a test-specific application-test.properties. These integration tests can verify that repositories and services work together. Also consider end-to-end tests (API contract tests) possibly with Testcontainers or Spring REST docs.

Email Sending: To send emails (e.g. registration confirmations, ride receipts), use Spring’s email support. Include spring-boot-starter-mail in your project. This starter configures a

JavaMailSender bean. You then set SMTP properties in application.properties (host, port, username/password for Gmail, SendGrid, etc.). In code, inject JavaMailSender and use it to compose and send emails. Typically one uses MimeMessageHelper to create an HTML or text email. Baeldung's "Guide to Spring Email" explains that JavaMailSender (from spring-boot-starter-mail) is the primary interface for sending emails. Ensure to disable email-sending or use a mock SMTP (like Mailtrap) in dev/test environments (e.g. via a spring.profiles=dev setting).

Code Design Suggestions

Layered Architecture: Organize code into layers. Define **Controller** classes (annotated @RestController) for each microservice to handle HTTP requests and route them to services. Each controller should delegate business logic to a **Service** layer (annotated @Service) which implements an interface (e.g. UserService, RideService, etc.). The service layer contains the core logic and calls **Repository**/DAO classes (annotated @Repository) for database access (using Spring Data JPA or similar). Keep controllers thin (just mapping endpoints and request/response conversion) and put validations and transactions in the service layer.

DTOs and Entities: Use **Data Transfer Objects (DTOs)** to represent API request/response payloads, separate from JPA entities. For example, a RideRequestDto can capture JSON fields from the client, which a controller converts into a RideRequest entity or command object internally. This prevents exposing internal models (e.g. database fields) and allows validation. You might use libraries like MapStruct or simple constructors to map between DTOs and entities. As an example from Baeldung, one can use a @Builder on DTO records to fluently create them.

Interface-Based Services: Define interfaces for each service component (UserService, DriverService, etc.) and then implement them (UserServiceImpl). This makes testing easier (you can mock the interface) and allows swapping implementations if needed.

Dependency Injection: Leverage Spring's DI. Use constructor injection (@Autowired on constructors or rely on a single constructor) for services and repositories. For example:

```
@RestController
public class RideController {
    private final RideService rideService;
```

```
    public RideController(RideService rideService) { this.rideService = rideService; }

    // endpoints...

}
```

This ensures loose coupling and easy unit testing.

Configuration and Beans: Centralize configuration using @ConfigurationProperties or @Value (for things like API keys, SMTP settings, strategy selection, etc.), and consider making utility classes singletons. For example, you might have a @Bean for a GeoService or EventPublisher that is reused across services.

Error Handling: Include global exception handlers (@ControllerAdvice) to convert exceptions into meaningful HTTP error responses. Use appropriate HTTP status codes (e.g. 404 for ride not found, 400 for validation errors).

By separating interfaces, DTOs, controllers, and service implementations, and by using Spring's built-in DI, the codebase stays modular and testable. Each microservice can be packaged independently (e.g. as its own Spring Boot application), or some can be grouped behind an API Gateway if desired.

Sources: The above breakdown follows common microservices design practices and Uber-like system design guides. Patterns references are from design pattern literature, and Spring Boot-specific guidance is from official and community tutorials.

Citations

System Design of Uber App | Uber System Architecture - GeeksforGeeks

https://www./system-design/system-design-of-uber-app-uber-system-architecture/

System Design: Uber - DEV Community

https://dev.to/karanpratapsingh/system-design-uber-56b1

GitHub - richxcame/ride-hailing: A scalable, microservice-ready backend powering a modern ride-hailing ecosystem — including passenger, driver, and admin,... services. Built with Go, PostgreSQL(Postgis*), and Docker, and designed for Cloud Run + Cloud SQL deployment on Google Cloud.

https:///richxcame/ride-hailing

System Design of Uber App | Uber System Architecture - GeeksforGeeks

https://www./system-design/system-design-of-uber-app-uber-system-architecture/

GitHub - anasabbal/mini-uber-microservice: Nuber (Mini uber microservice - Spring Cloud) there is another version advanced but sorry it's private (stable bransh B-STABLE-01 and refactor-03)

https:///anasabbal/mini-uber-microservice

GitHub - richxcame/ride-hailing: A scalable, microservice-ready backend powering a modern ride-hailing ecosystem — including passenger, driver, and admin,... services. Built with Go, PostgreSQL(Postgis*), and Docker, and designed for Cloud Run + Cloud SQL deployment on Google Cloud.

https:///richxcame/ride-hailing

System Design: Uber - DEV Community

https://dev.to/karanpratapsingh/system-design-uber-56b1

Strategy Design Pattern - GeeksforGeeks

https://www./system-design/strategy-pattern-set-1/

Implement the Builder Pattern in Java | Baeldung

https://www./java-builder-pattern

Factory method Design Pattern - GeeksforGeeks

https://www./system-design/factory-method-for-designing-pattern/

Singleton Design Pattern vs Singleton Beans in Spring Boot | Baeldung

https://www./spring-boot-singleton-vs-beans

How to use Swagger with Spring Boot

https:///blog/documenting-rest-api-with-swagger-in-spring-boot-3/

Testing in Spring Boot | Baeldung

https://www./spring-boot-testing

Guide to Spring Email | Baeldung

https://www./spring-email

How to use Swagger with Spring Boot

https:///blog/documenting-rest-api-with-swagger-in-spring-boot-3/