System Design for Dada

Table of Contents

[1 Architecture 2](#_Toc13694664)

[1.1 Micro Services 2](#_Toc13694665)

[1.1.1 User Management Service 2](#_Toc13694666)

[1.1.2 City Management Service 2](#_Toc13694667)

[1.1.3 Position Tracking Service 2](#_Toc13694668)

[1.1.4 Order Management Service 2](#_Toc13694669)

[1.1.5 Pricing Service 2](#_Toc13694670)

[1.1.6 Deal Making Service 3](#_Toc13694671)

[1.1.7 Payment Management Service 3](#_Toc13694672)

[1.1.8 Order Review Management 3](#_Toc13694673)

[1.1.9 User Review Management 3](#_Toc13694674)

[1.1.10 Distributed Lock Service 3](#_Toc13694675)

[1.2 Diagram 3](#_Toc13694676)

[1.3 API Definitions and Database Schemas 3](#_Toc13694677)

[1.3.1 User Management Service 4](#_Toc13694678)

[1.3.2 City Management 5](#_Toc13694679)

[1.3.3 Position Tracking 6](#_Toc13694680)

[1.3.4 Order Management 7](#_Toc13694681)

[1.3.5 Pricing Service 8](#_Toc13694682)

[1.3.6 Deal Making Service 9](#_Toc13694683)

[1.3.7 Payment Management 9](#_Toc13694684)

[1.3.8 Order Review Management 9](#_Toc13694685)

[1.3.9 User Review Management 9](#_Toc13694686)

[1.3.10 Distributed Lock Service 9](#_Toc13694687)

[2 Key Design Point 9](#_Toc13694688)

[2.1 How to find closest available driver. 10](#_Toc13694689)

[2.2 How to make a deal so that no conflict on driver between two new orders. 10](#_Toc13694690)

[2.3 How to horizontal scale Dada system to enable more cities. 10](#_Toc13694691)

# Architecture

First, I would break down the application into a set of loosely decoupled, collaborating services as follows:

* User Management
* City Management
* Position Tracking
* Order Management
* Pricing Service
* Deal Making Service
* Payment Management
* Order Review Management
* User Review Management
* Distributed Lock Service

## Micro Services

### User Management Service

User management service handles user registration, user login and logout, user role (driver or passenger) and other account related stuff.

It exposes following APIs:

* Register new account
* Update account role as driver or passenger or both
* User login
* User logout
* User authentication
* User authorization

### City Management Service

City management service manage cities on-boarded on Dada.

It exposes following APIs:

* Add new city
* Disable city
* Get all enabled cities
* Get city detail by city id

### Position Tracking Service

Position tracking service records the real-time location (latitude, longitude and height) of drivers and passengers whenever Data app is opened on mobile phones.

It exposes following APIs:

* Update real-time location (latitude, longitude and height)
* Find drivers within specified radius

### Order Management Service

Order management service manage orders for users. It tracks passengers and drivers order history and create new order.

It exposes following APIs:

* Create new order for specified passenger and driver
* Get order history (support pagination)
* Get current order (active order)
* Update order status

### Pricing Service

Pricing Service calculate estimated price for new orders and calculate real-time price for ongoing orders.

It exposes following APIs:

* Calculate estimated price for new order
* Calculate real-time price for ongoing orders

### Deal Making Service

Deal making service’s responsibility is to match passenger with the closest available driver and create a new order for them.

It exposes following APIs:

* Create ride request for passenger (Triggered by passenger from mobile APP), this API also enqueuer a message to queue, detail is [here](#_API_Definitions).
* Cancel ride request (Triggered by passenger from mobile APP)
* Match closest available driver and make an new order (Triggered by backend server)

### Payment Management Service

Payment management service handles the deal payment.

It exposes following APIs:

* Add new payment method
* Update/remove payment method
* Update order payment status

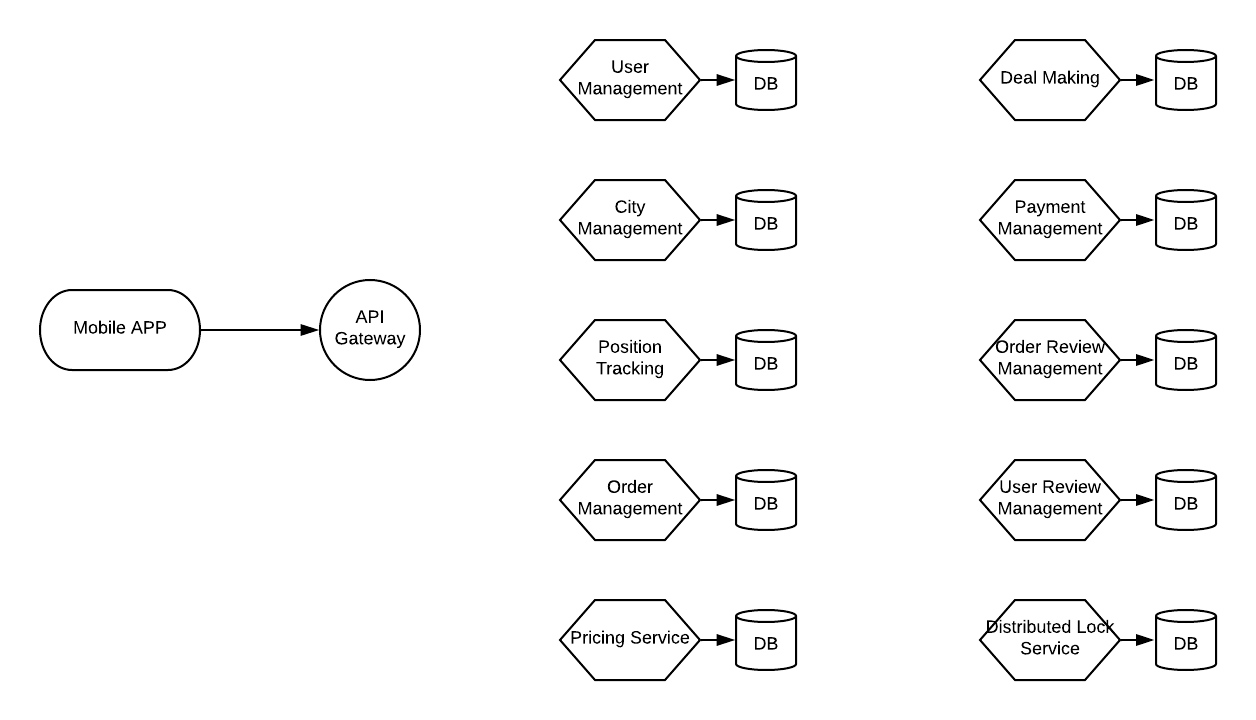
### Order Review Management

### User Review Management

### Distributed Lock Service

This service provides a distributed lock service for other service.

## Diagram



## API Definitions and Database Schemas

All micro services expose RESTful APIs.

All request and response are in json format.

The code field returned from APIs is global unique.

### User Management Service

#### Database Schemas

Use mongoDB to save user and role data.

* User model:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| user\_id | String | Yes | Yes | Yes |
| user\_name | String | No | Yes | No |
| password\_hash | String | No | No | No |
| nick\_name | String | No | No | No |
| roles | List<String> | No | No | No |

* Role model:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| role\_id | String | Yes | Yes | No |
| role\_name | String | No | Yes | No |

#### API Definitions

* Register new account

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user |  |
| Method | Post |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Update account role as driver or passenger or both

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user/{user\_id}/role |  |
| Method | Put |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* User login

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user/login |  |
| Method | Post |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* User logout

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user/logout |  |
| Method | Post |  |
| Header | session\_id | Header |
| Body |  |  |
| Response | | |
| Body |  |  |

* User authentication

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user/authentication |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* User authorization

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/user/authorization |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

### City Management

#### Database Schemas

Use mongoDB to save city data.

* City model:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| city\_id | String | Yes | Yes | No |
| city\_name | String | No | Yes | No |
| enabled | Boolean | No | No | No |

#### API Definitions

* Add new city (For administration)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/city |  |
| Method | Post |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Disable city (For administration)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/city/{city\_id} |  |
| Method | Put |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Get all enabled cities

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/cities |  |
| Method | Get |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Get city detail by city id

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/city/{city\_id} |  |
| Method | Get |  |
| Body |  |  |
| Response | | |
| Body |  |  |

### Position Tracking

#### Database Schemas

Use mongoDB to save position data. Use Redis to cache position data in memory.

* Position model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| user\_id | String | Yes | Yes | No |
| city\_id | String | No | No | Yes |
| location\_time | Timestamp | No | No | No |
| location | EmbededDocument | No | No | No |

#### API Definitions

* Update real-time location (latitude, longitude and height)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/position |  |
| Method | Post |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Find drivers within specified radius

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/closestdrivers |  |
| Method | Get |  |
| Header | session\_id |  |
| Body | passenger\_position |  |
| Response | | |
| Body |  |  |

### Order Management

#### Database Schemas

Use mongoDB to save order data.

* Order model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| order\_id | String | Yes | Yes | No |
| status | Int | No | No | No |
| passenger\_id | String | No | No | No |
| driver\_id | Timestamp | No | No | No |
| from\_location | EmbededDocument | No | No | No |
| to\_location | EmbededDocument | No | No | No |
| start\_datetime | Timestamp | No | No | No |
| Finish\_datetime | Timestamp | No | No | No |
| payment\_info | EmbededDocument | No | No | No |

#### API Definitions

* Create new order for specified passenger and driver

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/order |  |
| Method | Post |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Get order history (support pagination)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/orders |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Get current order (active order)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/order/active |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Update order status

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/order/{order\_id} |  |
| Method | Put |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

### Pricing Service

#### Database Schemas

Use mongoDB to save price card data.

* Order model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| price\_card\_id | String | Yes | Yes | No |
| city\_id | String | No | No | No |
| unit\_price | Decimal | No | No | No |

#### API Definitions

* Calculate estimated price for new order

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/price/estimate |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Calculate real-time price for ongoing orders

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/order/{order\_id}/price/ongoing |  |
| Method | Get |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

### Deal Making Service

#### Database Schemas

Use mongoDB to save ride request data. Use Redis to cache ride request in memory.

* Ride Request model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| request\_id | String | Yes | Yes | No |
| user\_id | String | No | Yes | No |
| city\_id | String | No | No | Yes |
| status | Int | No | No | No |
| from\_location | EmbededDocument | No | No | No |
| to\_location | EmbededDocument | No | No | No |

#### API Definitions

* Create ride request for passenger (Triggered by passenger from mobile APP)

This API just save the ride request do Redis and Mongodb. And then enqueuer a message to queue to ask backend server to match closest driver and make a deal.

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/ride-request |  |
| Method | Post |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Cancel ride request (Triggered by passenger from mobile APP)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/ride-request/{request\_id}/cancel |  |
| Method | Put |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

* Match closest available driver and make an new order (Triggered by backend server)

|  |  |  |
| --- | --- | --- |
| Request | | |
| URL | https://{domain}/api/v1/ride-request/{request\_id}/make-deal |  |
| Method | Post |  |
| Header | session\_id |  |
| Body |  |  |
| Response | | |
| Body |  |  |

### Payment Management

#### Database Schemas

Use PostgreSQL to save payment info.

* Payment Record model

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Field | Type | Primary Key | Unique | Shard Key |
| payment\_record\_id | String | Yes | Yes | No |
| order\_id | String | No | No | No |
| amount | Decimal | No | No | No |
| discount | Decimal | No | No | No |
| actual\_amount | Decimal | No | No | No |
| payment\_method | String | No | No | No |

#### API Definitions

* Add new payment method
* Update/remove payment method
* Update order payment status

### Order Review Management

### User Review Management

### Distributed Lock Service

# Key Design Point

I think following key point should be taken into consideration in designing Dada system.

## How to find closest available driver.

As above design, two services are involved in finding closest available driver: Position tracking service and order management service. Position tracking service knows the drivers close to passenger who want a ride, but it doesn’t know which drivers are free. Order management service knows which drivers are free, but it doesn’t have the position data. I thought out two ways to solve this problem.

One way is to update driver’s available property in position tracking service too. Whenever order status is updated, order management service publish a message. Position tracking service subscribe this type of message and use it to update driver’s free status.

Another way is combine the two service to find the closest driver. Position tracking service expose API to find drivers in a specified radius, order management system expose API to tell who are free. If not drivers are free in specified radius, we grow the radius and retry until the radius reach an upper threshold.

I prefer the second solution because I think the first one make the boundary between these two services unclear.

## How to make a deal so that no conflict on driver between two new orders.

We need to take race condition seriously when make a deal for a ride request. There would be race condition if two passengers match the same available driver at the same time for the availability of driver would be changed during the time between ‘get free closest driver’ and ‘make a deal’. (We can do this easily in RDBMS by setting the isolation level as Repeatable Read) To avoid this, we need to lock the driver before assign the ride request to him.

## How to horizontal scale Dada system to on-board more cities.

User horizontal scaling to support more cities.

First, each service should have load balance and auto-scaling mechanism.

For database, we can shard mongoDB collections. For User collection, we can shard it by user\_id. For position collection and ride request collection, we can shard it by city\_id.

## Handle race condition.

I have a distributed lock service to help get rid of race condition.