Table of Contents

Envelope Calculations Per Cluster

- Taxi
  - Number of Taxi Registered: 200,000
  - Number of Shifts: `3x16` Hours each
  - Active Taxis Average: 2/3 x N = `130,000`
  - Location Updates: 130,000/m = 130,000/60/second ~= `2100/sec`
- Users
  - Number of registered users: 10,00,000
  - Peak Query: N/10 = 1,00,000/1 hour = 27.77/s ~= `30/sec`
  - Average Load: N/10 = 1,00,000/day ~ `2/second`
  - SLA Average Load: Taxi lookup under 60 seconds
  - SLA Peak Load: Taxi Lookup under 120 seconds

Architecture Designs

There are three actors in whole project representing three real world entities.

1. *Server*: Cloud based application which handles all the business logic
2. *Taxi*: A client that represents a taxi for hire. A registered taxi can receive pick up request nearby
3. *User*: A client that represents a person who wants to hire a taxi for travel. A registered user can ask for an available taxi nearby

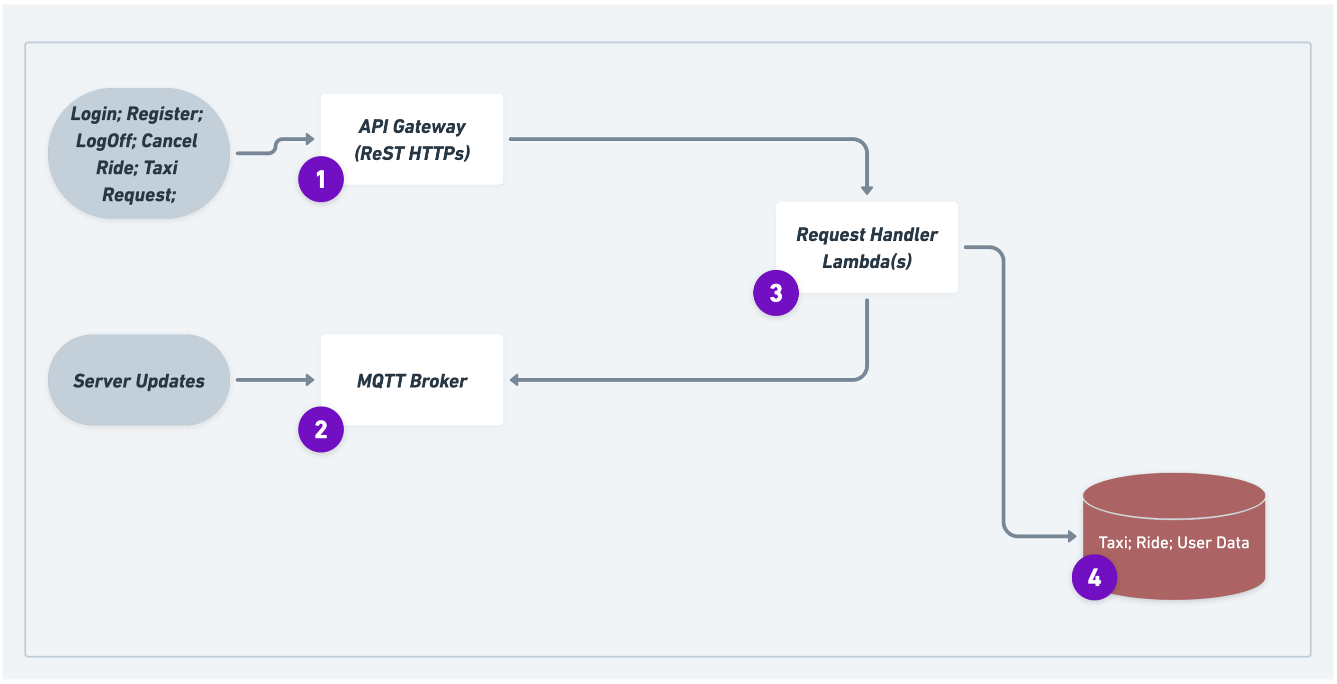
There are two distinct categories of communication between clients (user/taxi) and server.

1. ***Synchronous Requests***: These are requests by clients which requires immediate responses with blocking I/O. Example of such requests are *login*, *logout*, *cancel_taxi*, *register* and so on.

Since these are less frequently used, it makes sense to use simple *Client-Server* Communication model here. A ReST Api over HTTPs is a good fit for these requests.

2. ***Long Term Continuous Communication***: These are data exchanges which are extended over long time and are frequent in nature. Message Queueing protocol MQTT based endpoint can serve data transfer between client and server for these requests. Example data transfer could be *location update* sent by taxi, *new user request* assigned to a taxi, *changes in ongoing ride*, *cancellation of a ride by user* and so on.

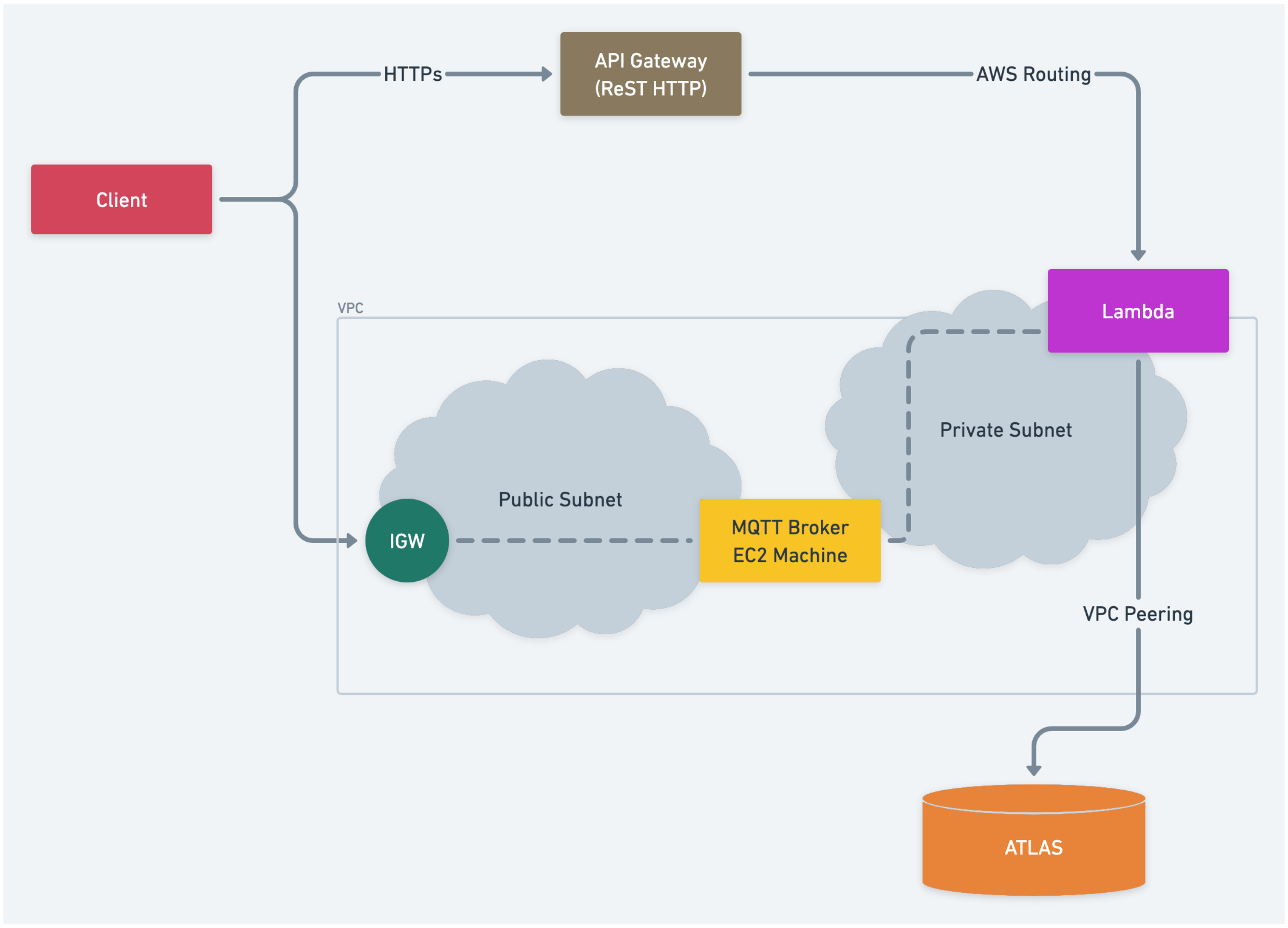
High Level Organization


Login; Register; LogOff; Cancel Ride; Taxi Request;
API Gateway (ReST HTTPs)
1
Request Handler Lambda(s)
3
Server Updates
MQTT Broker
2
Taxi; Ride; User Data
4

Below are the components that are used in the system design for serving different purposes. The components are numbered and are explained in natural order of the numbering.

1. **API Gateway (ReST):** A HTTP endpoint that serves the synchronous requests to register a user or taxi in the system. Requests are forwarded to a Lambda which internal does the required validations and creates a new entry in database.
2. **MQTT Broker(s):** These MQTT brokers are exposed over internet. Each topic allows pub/sub based communication from server to clients. Since MQTT are efficient in keeping long connections with minimal overhead, it provides an excellent mechanism to allow clients to read async communication from server. Since MQTT Brokers can be scaled, design has been keep simple to be able to use multiple clusters if needed.
3. **Lambda Functions:** Used to perform logic on the request incoming from taxi or user
4. **GeoSpatial Capable Database:** Houses data for whole system. It can be a DB or Persistent Cache that supports GeoSpatial queries. Possible candidates are MongoDb, ElastiCache or Redis Cache

Deployment Model in AWS



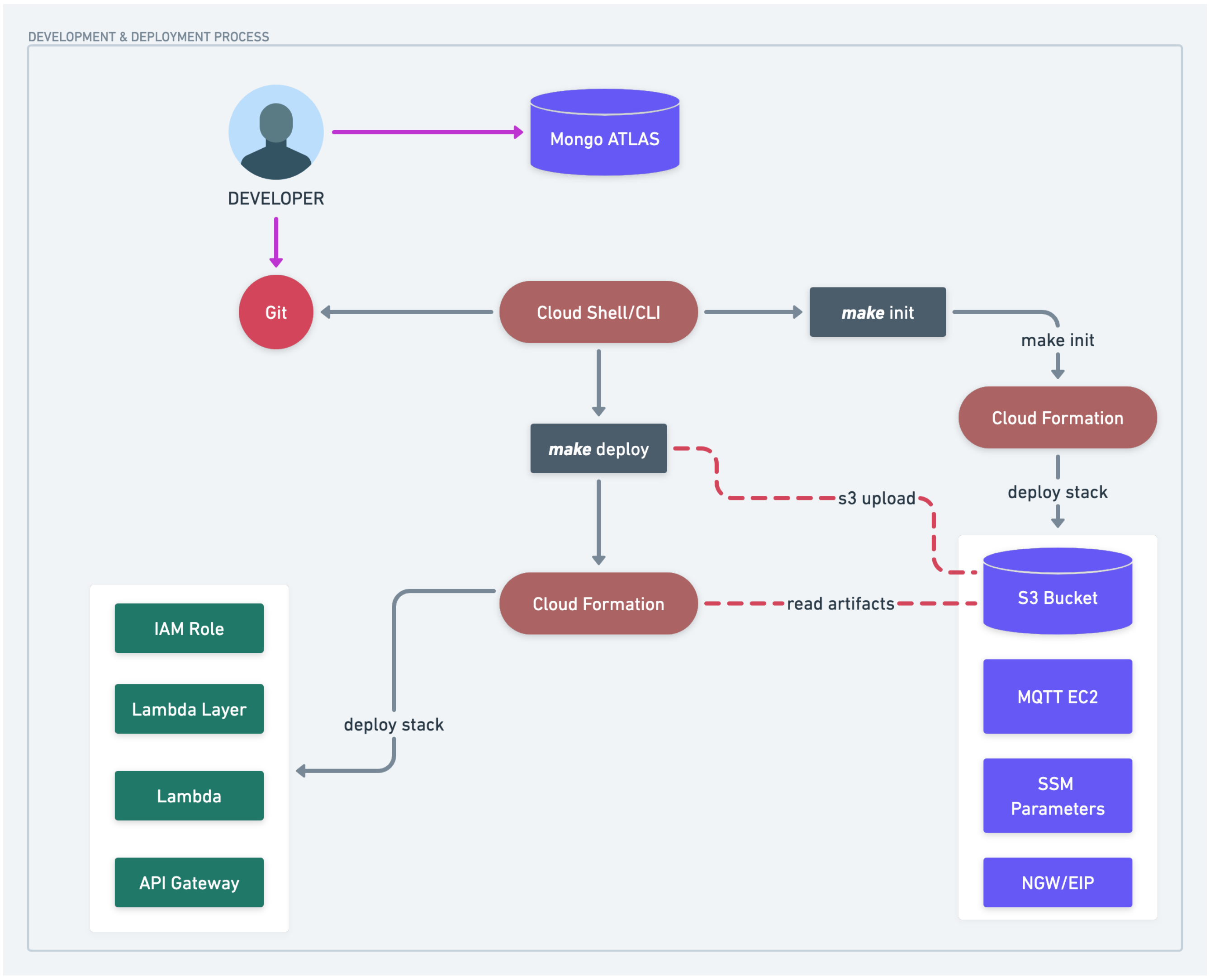
- **Client:** Python Code that runs anywhere in the world and has access to the internet
- **Lambda/Api Gateway:** Both of them are deployed using *CFN*. Lambda functions are deployed within the subnet
- **MQTT Broker (EC2):** Deployed in the same *public subnet* as Lambda Functions. Lambda functions can access this EC2 instance using private IP while *clients* can access the Machine over Public Ip
- **Mongo Atlas:** Deployed separately using Mongo Cloud Service. Lambda functions access Mongo using VPC Peering

CI-CD Model

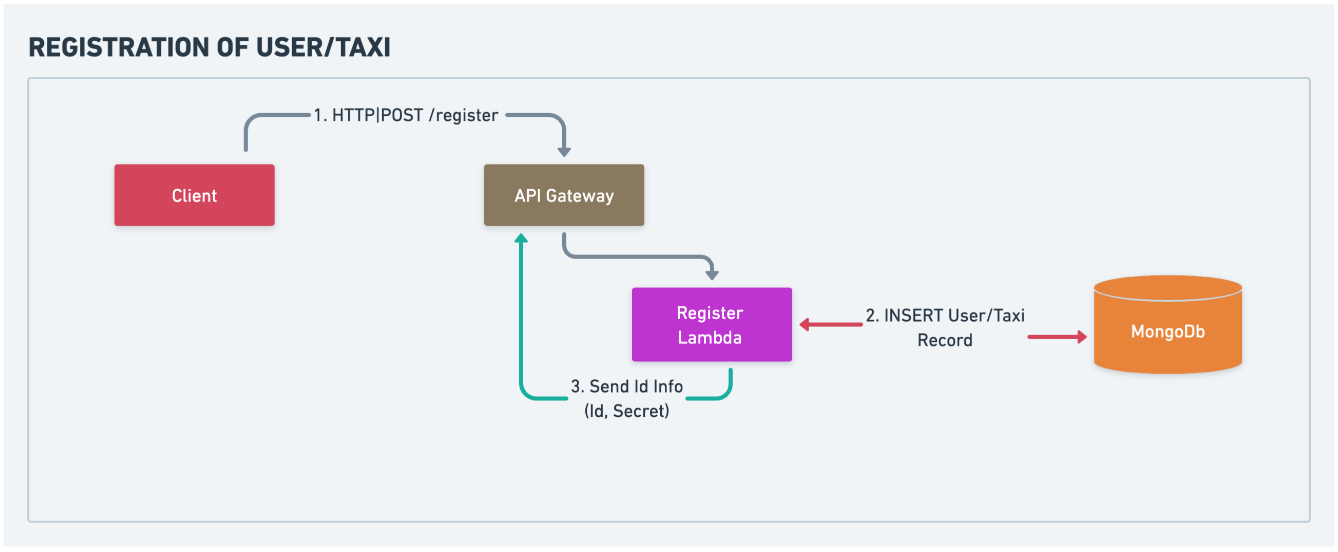
For automated code build and deployment, we have used `make` system to automate code build and code deploy

- Each developer uses a *namespace* to have their personal stack
- `make init` creates a *S3 Bucket* and *MQTT EC2* instance that can be shared among all
- `make deploy` packages the code and uploads to the S3 Bucket on appropriate path(s). Same build artifacts are then used for deploying stack using *Cloud Formation*

To assist local development, there was a small flask app was written that can convert a HTTP request into a *AWS Api Gateway Proxy* event and invoke the lambda function. This has helped a lot to debug issues with quick turnaround time.



API: User/Taxi Registration

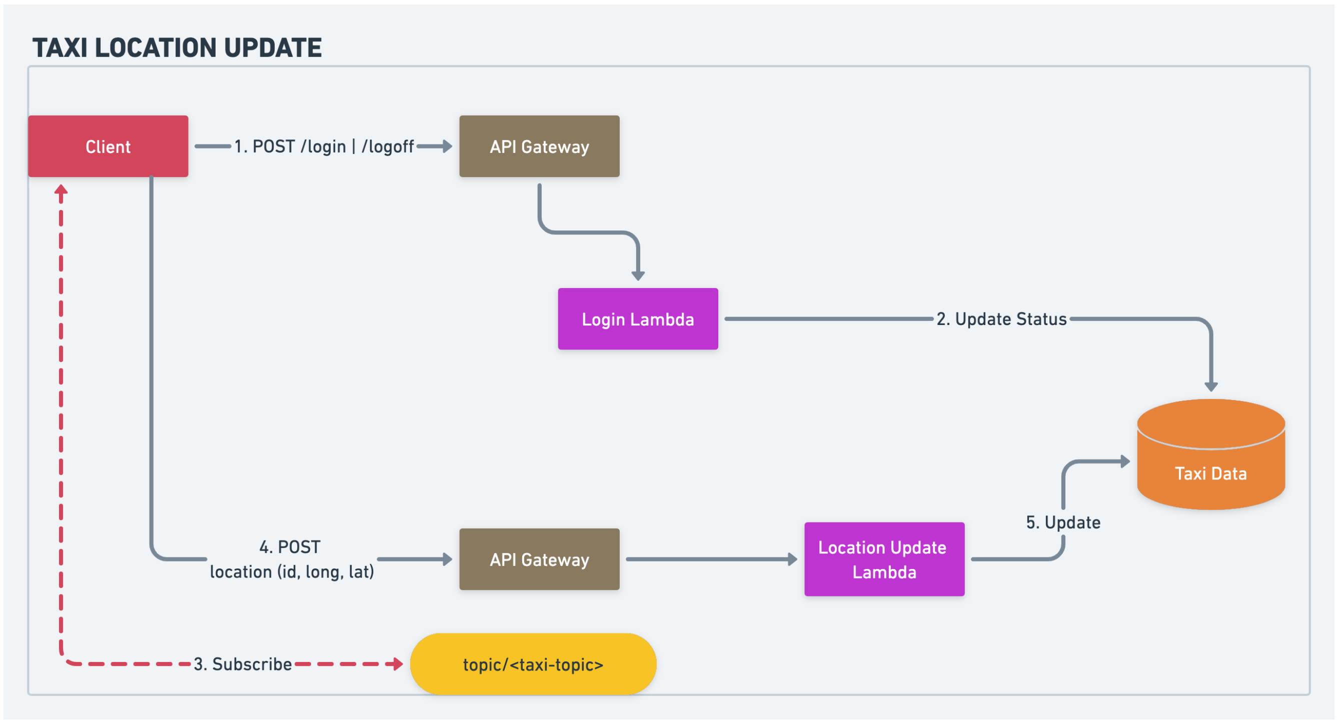

REGISTRATION OF USER/TAXI
1. HTTP|POST /register
Client
API Gateway
Register Lambda
2. INSERT User/Taxi Record
MongoDb
3. Send Id Info (Id, Secret)

A user and taxi needs to be registered within the system to be able to use the service. In effect, *registration* Api becomes the first interaction of user with the system. The flow goes like below

1. User/Taxi makes a `HTTP POST` call to `/register` Api with JSON payload. Payload contains specific information about a new user or taxi. Request is recieved by API Gateway and is routed to a *register lambda* function
2. Register Lambda function does the validation and creates a record in database. It also generates a random *secret* for each profile which provides security via `JWT`
3. A success response is sent back with assigned `id` and a `secret`

On a *success* response client is supposed to persist the data locally along with the assigned `Id`.

API: Taxi Login

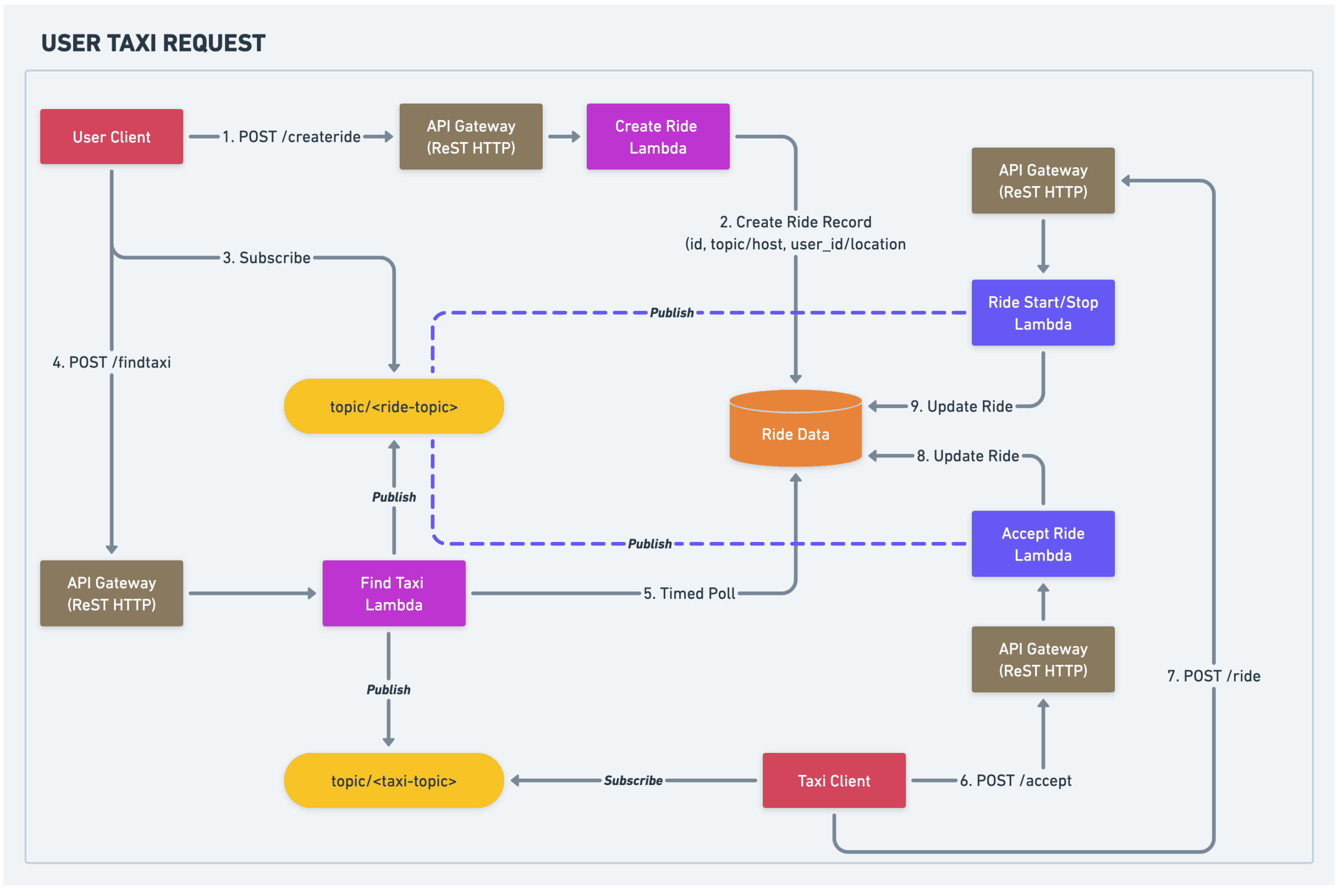


Once a taxi successfully registers with system, it gets an `Id` and a `Token`. Using these two taxi can start or stop getting pickup requests from system. The flow goes like below;

1. Taxi makes a `POST /login` call received by API Gateway endpoint. Then it is routed to *Login Lambda* with below structure;
   - `taxi_id` as a Header `X-Taxi-Id`
   - A `HS256` based JWT token as header `X-Token`
2. Login Lambda validates the *token* using *secret* stored in database. A *random* topic name is generated for this taxi until a *logoff* call is made by the taxi. This *topic name* will be used to send messages to taxi from server. Along with the *topic name*, a *mqtt server host* is also sent which represents the address for broker. This allows us to distribute load on multiple brokers if needed to scale
3. Taxi client reads the *mqtt host* and *topic name* and subscribe to the topic
4. Taxi then starts making *location update* call to the Api Gateway using HTTPs REST calls
5. For each update call, the current Geo Location is stored in Database. This location will be used for taxi lookups

Once taxi wants to stop serving request, a `/logoff` Api call is made which is routed to a *Logoff Lambda*. Once the request is validated, the taxi is *marked* as `OFFLINE`. Such *offline* taxi are not used for taxi allocation.

API: Taxi Booking



Once a user successfully registers with system, it gets an `Id` and a `Secret`. Using these two, user can start booking taxi from system. It is a 2-Step process and the flow goes like below;

1. User makes a `POST /createride` call with below structure;
   - `user_id` as a Header `X-User-Id`
   - `jwt` as a Header `X-Token`
   - JSON Payload containing *user location*
2. Request is handled by *create ride lambda*. It generates a random, *topic name* for the ride request, creates `ride` record in database containing, user id, location, topic name and status. All of these are sent back to client along with *mqtt broker* address
3. Client reads the broker address and topic name from the response and subscribes to it for updates
4. Client makes an authenticated ReST `POST /findtaxi` call to make server look for taxi
5. Request is served a *find taxi lambda* which does the heavy lifting in below manner
   - It reads the ride data and finds the available taxis
   - It sends a ride request via a message onto *mqtt broker topic* as saved in taxi record

    - It then does a poll on the *ride record* to be updated by an available taxi
    - If a taxi accepts the request, a confirmation is sent to user by publishing a message onto the *ride topic*
    - If no taxi accepts the request, a confirmation is sent after a timeout of 2 minutes

6. An available *taxi client* will be listening on a taxi topic as assigned by the server. For a taxi request, if a taxi is nearby, a message will be sent with ride details onto this *taxi topic*. The taxi client will read this message from *mqtt topic* and may decide to *accept* or *reject* the request. In any of the case it makes a *accept* call. The *accept* call is served by a Lambda which updates the client about a rejection or acceptance by a taxi. If the taxi is the one to accept a ride, *ride* record is update to mark the taxi assigned and status of the record is updated. In the next poll, the *find taxi lambda* will return a success in taxi allocation for the request. A confirmation is sent to *taxi client* indicating that it was the first one to respond and hence it is assigned to the ride
7. Taxi Client then can start and then end the ride. It makes an authenticated `POST /ride` call that is served by a *ride lambda*. When a ride starts, ride record is updated and user client is notified via *mqtt user topic*. Similarly, when the taxi ends the ride, it sends a *ride update* onto the same ReST call. The *ride lambda* then reads the ride completion and then it updates the taxi status and ride status. A ride completion message is sent to the *mqtt user topic* so that user client can consider a ride to be completed

Envelope Size Estimation: Taxi

```
Taxi Mongo Record {
 "id"  : "<string:128>",
 "type": "<string:32>",
 "registered_on": <epoch:8>,
 "license_number": "<string:64>",
 "manufacturer": "<string:64>",
 "model": "<string:64>",
 "driven_by": {
   "name": "<string:256>",
   "license": "<string:256>",
   "expiry": <epoch:8>
 },
 "token": "<string:128>"
}

Taxi Location Records {
 "last_update": <epoch:8>
 "longitude": <float:8>
 "latitude": <float:8>
}

Taxi Status Record {
  "last_update": <epoch:8>
  "logout_at": <epoch:8>,
  "trip_id": <long:8>,
}
```

- Taxi
  - total=200,000; active=130,000
  - Max Record Size: 1024 Bytes (1KB)
  - Max Active Database Size: 200,000KB/ 200MB
  - Max Status Size: 32B
  - Max Cache Size: 130,000 x 32B / 4MB
  - Peak Cache Memory Size: 200,000 x 32B / 6MB

Envelope Size Estimation: User

```
User Mongo Record {
 "id"  : "<string:128>",
 "registered_on": <epoch:8>,
 "gender": "<string:32>",
 "contact": "<string:64>",
 "address": "<string:64>",
 "name": "<string:256>",
 "token": "<string:128>"
}

User Trip Status Record {
  "last_update": <epoch:8>
  "trip_id": <long:8>,
  "taxi_id": "<string:128>"
}
```

- Users
    - Number of registered users: 1000,000
    - Cache Size Max: 200,000 * 144B/ 28MB
    - Database Size: 680B * 1M / 680MB