Bicycle Rental System

Team 3
Hongyuan Li
Kang-Hua Wu

Introduction

Our bike rental system is designed to offer users with full bike rental experience. Users can book a bike a selected station through User Web UI. This booking process allows user to reserve bikes ahead of time, which helps prevent the “out of stock” scenario when customers arrive at the bike station.

Our application is a non-blocking focused system. There are two major components, User Service and Station Service. The communication between these two services are through message queue, in a non-blocking fashion. The User UI interacts with the User Service through RESTful API and the Station UI interacts with the Station Service through RESTful API as well.

Our non-blocking focused system take various system failures into consideration. Intermittent network may slow down a transaction, if the transaction is handled only through blocking system. For example, in an extreme scenario where a user submits a reservation request, but the station is disconnected for ten seconds from the network. During the disconnection, all bike might have been checked out by pedestrians on site. Our non-blocking system could accept user’s initial submission, and later cancel user’s transaction in a timely fashion.

Our system design echoes the the design principles of Event Sourcing (ES) and Command Query Responsibility Segregation (CQRS). The communications between major components are through message queue, where each message could be considered as an event. Services take actions based on the information of different messages (or events). Further, User Request and the actual operations are seperated. For example, the bike booking request doesn’t return the booking results immediately. Instead, reservation confirmation will come be updated later, but in a timely fashion.

How to Access?

User Service

The User Service components are currently shutdown. Designer will provide urls for User UI and User API during demo.

Station Service

You can access the frontend webpage through [redacted]

Functionalities

User UI

- Reserve a bike with given a station id
- Display user name and remaining credit
- Display previous or processing orders

User Service

- As RESTful Server
    - POST User Request
    - Update cash balance
    - GET previous or processing order list
    - Generate unique transaction id for each order
- As Subscriber
    - Receive bike reservation Confirmation
- As Publisher
    - Send Reservation Message

Station UI

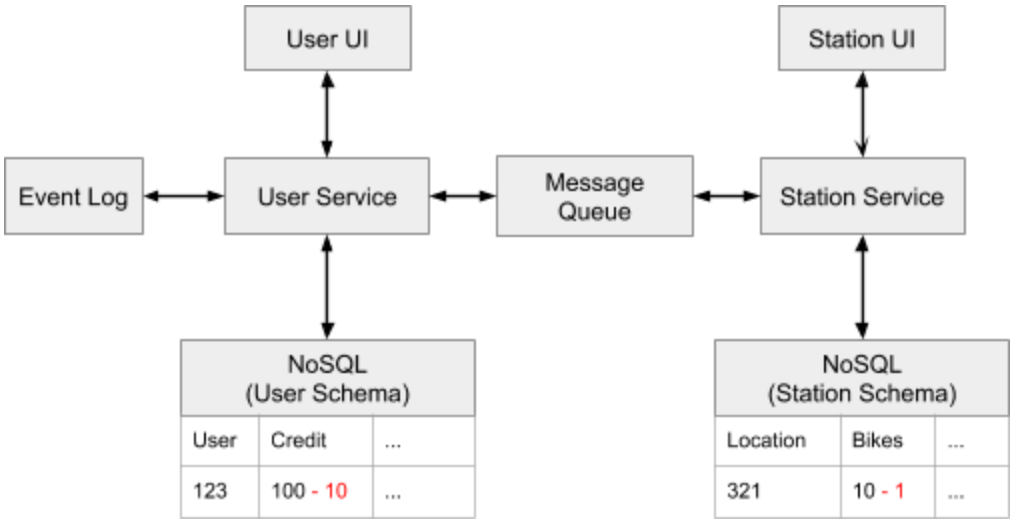
- Display all stations for selection
- Display details of selected station
- Input username to check out a bike after station selection
- Input bike Id to check in a bike after station selection

Station Service

- As RESTful Server
    - GET station list
    - POST check out bike
    - POST check in bike
    - Calculate rental fee
    - Keep track of available bikes
- As Subscriber
    - Receive reservation message
- As Publisher
    - Send message to confirm bike reservation
    - Send message to deny bike reservation
    - Send transaction complete message with rental fee information

The Timer Service mentioned in the proposal is not implemented in the final product, as we decided to put more efforts on User and Station UI development. The Event log functionality is incorporated to the User Service for simplicity

Architecture and Design



Design Patterns

The projected workload of our bike rental system is periodic. Heuristic knowledge tells us that bike rental activities go up during the day and fall down at night. Even though our system doesn't have the capability of auto-scaling, our design make this feature easy to implement.

Most of the components in our system are stateless. Our backend service protocol is RESTful API. There are two data stores (Cassandra and MongoDB) running at the backend. Data stores are stateful. However, since both data stores are NoSQL databases with high scalability, with proper configuration, we could make our stateful components scalable.

In this project, we only deployed single node data stores. It is possible to deploy Cassandra and MongoDB clusters when there's a need. Our system primarily handles transactions for bike rentals, which requires strong consistency. With feature such as tunable consistency, we can configure Cassandra with strong consistency and balanced read/write

Our design incorporated the message oriented middle. This feature greatly simplifies how services talk to each other. Through the asynchronous buffered communication among services, components with different processing speed can coexist in harmony.

We also designed our system with the principle of microservices in mind. We break down the functionality of the components of a service and deploy them independently. This approach offers convenience for future scaling.

Cloud Services

Our application is built on Google Cloud Platform (GPC). We manage user access and monitor system performance at the GPC Console. Our message queue service is provided by Google Pubsub. Our containers are deployed on Google Kubernetes Engine (GKE).

Implementation

The implementation section first discusses how we define our messages in Google Pubsub and then elaborates on the details of two main services.

Message Definition

There are three types of messages: reservation, confirmation and completion. User Service sends out Reservation Message after a user requests to book a bike and have enough balance in the user's account. Station Service sends out Confirmation message to either confirm or deny a user's booking request. Station Service also sends out Completion message when a bike is successfully checked in.

Reservation Message

- Topic Name: TOPIC_RESERVATION
- Subscription Name: SUB_RESERVATION
- After successful balance check

```
{
    "transaction_id": "UUID",
    "user_id": "abc",
    "station_id" : "s001",
}
```

Test Example:

```
gcloud beta pubsub topics publish TOPIC_RESERVATION
'{"transaction_id": "txn1", "user_id":"user1", "station_id":"s001"}'
```

Reservation Confirmation Message

- Topic name: TOPIC_CONFIRMATION
- Subscription name: SUB_CONFIRMATION
- Recipient: User Service, Aggregator
- Message 1:
    - Successful Reservation

```
{
   "transaction_id":"UUID",
   "user_id":"abc",
   "is_reserved":"true",
   "bike_id":"b0001",
   "station_id":"s001",
}
```

- Message 2
    - Unsuccessful Reservation

```
{
   "transaction_id":"UUID",
   "user_id":"abc",
   "is_reserved":"false",
   "bike_id":0,
   "station_id":0,
}
```

Transaction Completion Message

- Completion Message
    - Topic: TOPIC_COMPLETION
    - Sub: SUB_COMPLETION
    - After a successful bike check-in

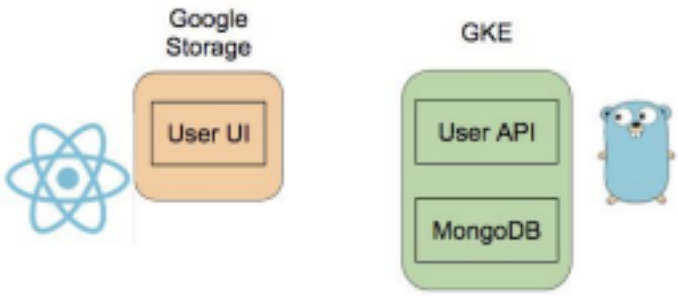
```
{
   "transaction_id":"UUID",
   "user_id":"abc",
   "grand_total": 123,
}
```

- Test Subscriber Example

```
gcloud beta pubsub subscriptions pull --auto-ack SUB_COMPLETION
```

User Service

From high level perspective, one of User Service's responsible is presenting user information including remaining credit, user's name and status of previous or processing order. The other responsibility of User Service is creating new reservation order through providing station id.



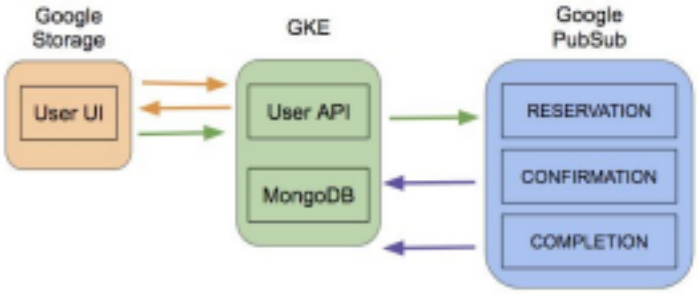
The User Service consists of two major components: User UI and User backend. User UI is written in Javascript using React and Redux framework. Since User UI is a static website, it is deployed to Google Storage. User UI has two sections as shown below. One for entering station id and the other for rendering User Info. On the other hand, User backend can be further divided into User API and MongoDB. Both User API and MongoDB are deployed to Google Kubernetes Engine as independent services.

Create Order

1. When creating an order, User UI post a station id to User API.
2. User API create a transaction id for this order and persist in MongoDB
3. Publish new order to Google PubSub

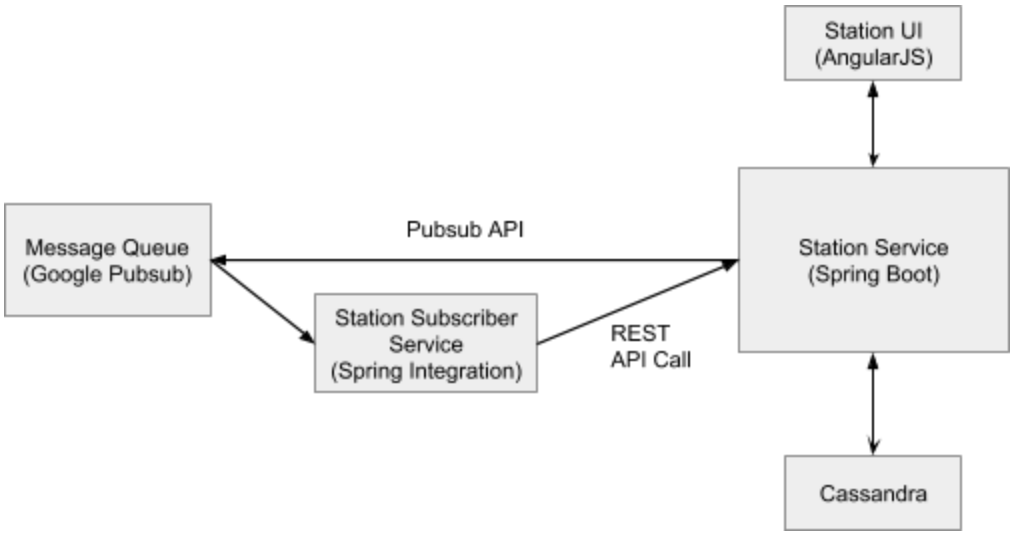
Update Order Status

1. User API consistently subscribe two Topics in Google PubSub
2. Update order as is Successful or Cancel
3. Update order as closed when user returns the bike to station



Station Service

Overview



Business logic is primarily implemented in with Spring Boot. Datastax cassandra driver and spring-data-cassandra libraries are used to connect with Cassandra. Station UI uses AngularJS to call the station RESTful API.

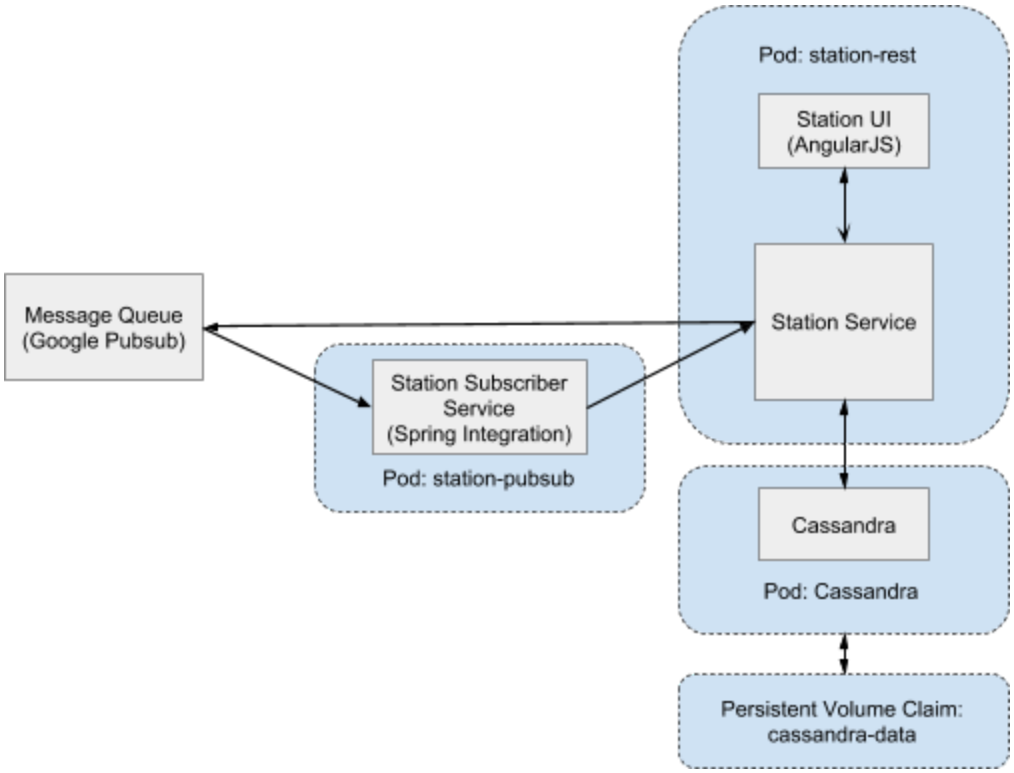
When implementing the interface between Spring Boot RESTful Server and Google PubSub, the initial design was using spring-integration-gcp and spring-cloud-gcp-starter-pubsub libraries, to bridge between Google Pubsub and Spring. Yet, there's a package reference conflict

between spring-cloud-gcp-starter-pubsub and the Datastax cassandra driver. Both of them have to work with com.google.guava package, but with different version. This conflict complicates the communication between Spring RESTful Server and Google Pubsub.

Eventually, we single out the subscriber functionality and build it on a seperate server, called Stationi Subscriber Service. This service receive message from Google Pubsub and called the RESTful API at Station Service.

When it comes to publisher functionality, we managed to use Google Pubsub API to send out message from Spring Boot RESTful Server to Google Pubsub. The Conflict addressed by excluding old com.google.guava package from the Maven dependency tree, as Google PubSub API has broader compatibility. Since message publishing operation is an on-demand process, it is viable to just include Google Pubsub API call as regular java method call. On the other hand, message receiving role requires a dedicated process. Therefore, a separate server is built.

Containers

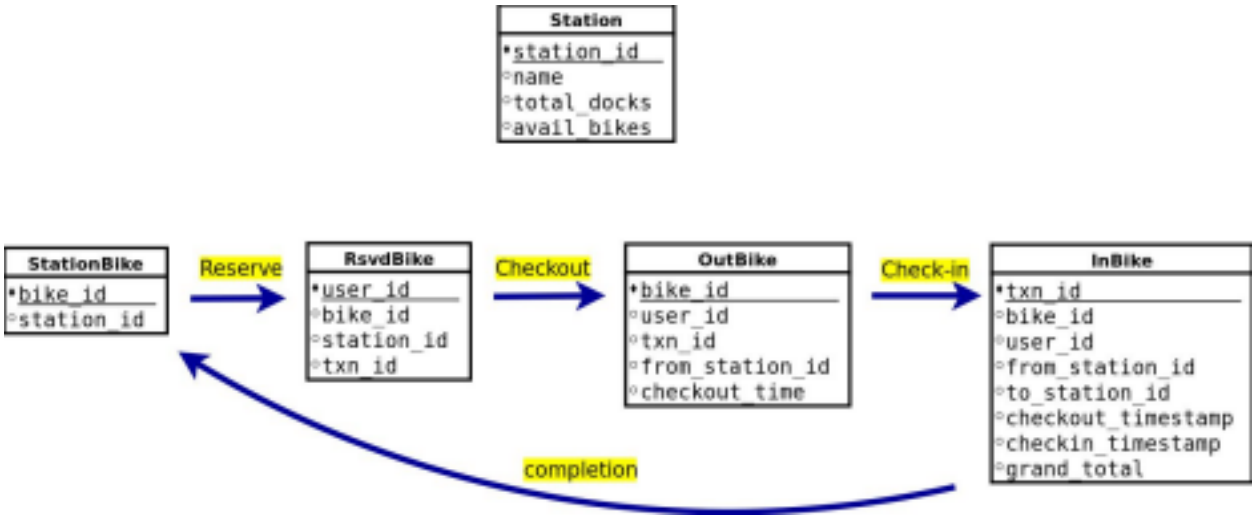


On Google Kubernetes Engine(GKE), three pods are created to break down Station Service into microservices. Currently, each pod only contains one container. Pod station-pubsub and station-rest are stateless, which offers high scalability. We could simply have multiple replicas of

the same container in each pod. Pod cassandra and its underlying persistent volume are stateful. Our current configuration allows only one Cassandra instance.

To scale Cassandra database, we could leverage its own high scalability. Though, complex configuration steps are involved, it is achievable to a Cassandra cluster within a pod or across multiple pods.

Data Model



The data model design for Cassandra database is quite different from relational databases. First, Cassandra is a key-value store that doesn't support complex sql query. When tracking the status of a bike in a relational database, the common approach would be creating a status table with bike id and status. When an event happens, the status is updated accordingly. However, it difficult to join tables in Cassandra and it's also inefficient to find a row with non-key columns. Secondly, data duplication is the essence of Cassandra. Effective Cassandra tables is designed based on queries, instead of saving storage.

A rental cycle of bike goes with docked, reserved, checked out, and checked in. For each action, query on database is different. When reserve a bike, we need to know the bike_id to find the station where the bike docked, thus bike_id is the key. To check out a reserved bike, user_id information is sufficient, thus user_id is the key of RsvdBike table. To check in a bike, again, we only need to know bike_id to gather other information. Once bike is checked in, it will be added to correct StationBike table. The number of available bikes at the destination station is updated. There a various duplicated data between tables, but the read and write are fast without any join.

Examples

Station Subscriber Service received a message from User Service with the following information. In this message, "user1" wants to book a bike at station "s0001" with transaction id "d9bdbb6a-94f8-45c6-a5cf-993d4ef2b350".

```
{
  "station_id": "s001",
  "transaction_id": "d9bdbb6a-94f8-45c6-a5cf-993d4ef2b350",
  "user_id": "user1"
}
```

Station Subscriber Service calls the `http://<host>:8080/brs/api/station/reserve` to reserve a bike. It turns out the station service has the available bikes. In the return message, we can see that bike id "b0005" is reserved for "user1"

At the Station UI, user select the station Id where the bike is reserve and type in the userId to checkout a bike. As show in the screenshot below, when select station s001, we can see this station is at King Library with 4 available bikes. When click at the "Check Out" button, we received a message says bike "b0005" is successfully checked out.

Back to the Station UI, when a needs to check in a bike. User can first select a destination station. As show in the screenshot below, station "s003" is selected. This station is at San Salvador & 9th. Then the user inputs bike ID "b0005" and click "Check In". The return message indicates that the bike is successfully checked in and this user is charged with $6.

Station Service then send the following message to Google Pubsub. It tells the User Service that transaction "d9bdbb6a-94f8-45c6-a5cf-993d4ef2b350" is completed and "user1" needs to be charged with $6.

```
{
      "transaction_id":"d9bdbb6a-94f8-45c6-a5cf-993d4ef2b350",
      "user_id":"user1",
      "grand_total":6.0
}
```

Source Code Folder Structure

- "message" folder
    - This sub-project defines messages in Java class for Java applications.
- "station-gcp" folder
    - This sub-project contains files about creating docker image, deploy GKE pods and common kubectl commands for Station Service

    - "image" folder
        - This folder includes build.sh script to create and push docker images to google cloud
        - "station-pubsub" folder
            - This folder includes a Dockerfile and a empty keyfile for a GCP service account
        - "station-rest" folder
            - This folder includes a Dockerfile and a empty keyfile for a GCP service account
    - ".yaml" files are used to create pods with kubectl
    - ".sh" scripts are kubectl commands.
- "station-pubsub" folder
    - This sub-project contains source code for the Station Subscriber Service
    - Run `mvn install` to generate executable jar
    - Run `mvn spring-boot:run` to build the project
    - This sub-project depends on project "message"
- "station-rest" folder
    - This sub-project contains source code for the Station RESTful Service
    - Run `mvn install` to generate executable jar
    - Run `mvn spring-boot:run` to build the project
    - This sub-project depends on project "message"
- "userapi" folder
    - In the subfolder src/userapi, this is where User API actions take place, including creating Restful service and connect with MongoDB
    - The other two subfolders, src/pubsub.pub and src/pubsub.sub, are used for testing purpose. By consuming or publishing messages into PubSub to see what's in PubSub helps to verify User API correctness.
    - Install related dependencies by executing go get ./...
    - All of go files can be run by executing go run file_name.go
- "userui" folder
    - Contains only User UI related files.
    - Executing npm run serve can load the project into debugging mode

Contribution

Task	Contributer	Start Date	End Date
User Service	Kang-Hua Wu	10/22/2017	11/16/2017
User Service on GKE	Kang-Hua Wu	12/05/2017	12/10/2017

Google Pubsub Messages	Kang-Hua Wu, Hongyuan Li	10/15/2017	11/16/2017
GKE Cluster	Hongyuan Li, Kang-Hua Wu	11/16/2017	12/04/2017
Station Service	Hongyuan Li	10/15/2017	11/28/2017
Station Service on GKE	Hongyuan Li	11/16/2017	12/04/2017
Station Data Model	Hongyuan Li	10/15/2017	11/16/2017