Distributed Systems (H0N08A)

Report: Distributed Cloud Application

Fionnan O'Sullivan *(r0978907)*
Joe Mulvany *(r0978720)*

November 17, 2023

Question 1

When deployed to a real cloud environment our application and the various systems and components would be arranged as follows:

Front End

The front end would consist of two components:

1. Client UI:

   The user interface used by the client to interact with the application, this would be accessed via a web browser on the clients machine. In our current implementation this is the website displayed at localhost/8080

2. Front-end services:

   This could be described as the presentation tier and in our implementation this is handled by Spring. This component would be hosted on a server and would handle UI rendering, user interactions and communication with the back end API for data retrieval and updates. The front end service also handles user Authentication using Firebase services.

Back End

The back end would consist of the Server-side Application, the external services utilised by the server-side application, background worker(s) and the database

1. The Server-side application:

   The server side application is the central hub for the distributed system that processes the booking platforms business logic, managing operations and interactions with the other components of the booking platform and external services. The application is responsible for the following tasks:

   Handles requests and updates from the front end:

   In our current implementation this is handled by the Trains Controller class using the Spring framework and their request mapping functionality. when a http request is placed by the front end to a particular URI, for example /api/getTrains, the application handles the request and returns the appropriate data, in this case a list of Train objects that will be formatted by the front-end services.

   Interacts with external train companies:

   In our current implementation this is again handled by the Trains Controller which uses a Webclient to retrieve information on trains, times, seats etc from the two train companies

   Interfaces with the data base:

   When a client requests to see their bookings, trainController will call on functions from firebaseController, firebaseController will then interface with the external database service, Firebase and retrieve the requested data.

Interfaces with Indirect Communication Services

In our implementation the task of confirming quotes is handled asynchronously by a background worker. In the TrainController class, the server-side application interfaces with Google Cloud pub/sub, the indirect communication controller, publishing booking request messages to the corresponding topic.

2. Firebase Authentication

   Firebase Authentication is an authentication service used in our application to verify users authentication levels for sensitive requests and ensure that customers can only access their own bookings. The user's browser will contact Firebase to verify the user through a log in, and once verified, attach an OAuth Identity token containing the user's identity and attributes to every request sent to the /api/ endpoints. The front-end Web service will contact Firebase and retrieve signing certificates to verify the users identity and access rights before sending the request though to the main application.

3. Cloud FireStore Database

   Firestore is a cloud-based database used in our application to store and query customer booking data. In our implementation the server-side application retrieves bookings stored on firestore through the FirestoreController.

4. Cloud Pub/Sub

   Cloud Pub/Sub is an external service that manages the indirect communication between the server-side application and the background worker. When the client wishes to confirm a booking, the application will publish a booking confirmation request to a Cloud Pub Sub topic, a subscription subscribed to this topic will retrieve the message and then push it to the background worker

5. Background worker

   The Background worker, in our implementation called SubscriberController, will be pushed booking confirmation request messages from Cloud Pub Sub. When one of these messages is received the worker will decode the relevant data, interface with the external train companies to book seats and retrieve tickets, create a booking object from these tickets and finally interface with the Cloud Firestore database to store the created tickets.

Note in our implementation the majority of the server-side application functionality was put into the TrainController Java Class. However creating a separate "Booking Controller" to publish booking confirmation requests to Cloud Pub/Sub and to query booking from the database, as well as creating a separate "Manager Controller" to handle manager only services would result in a more organised implementation.

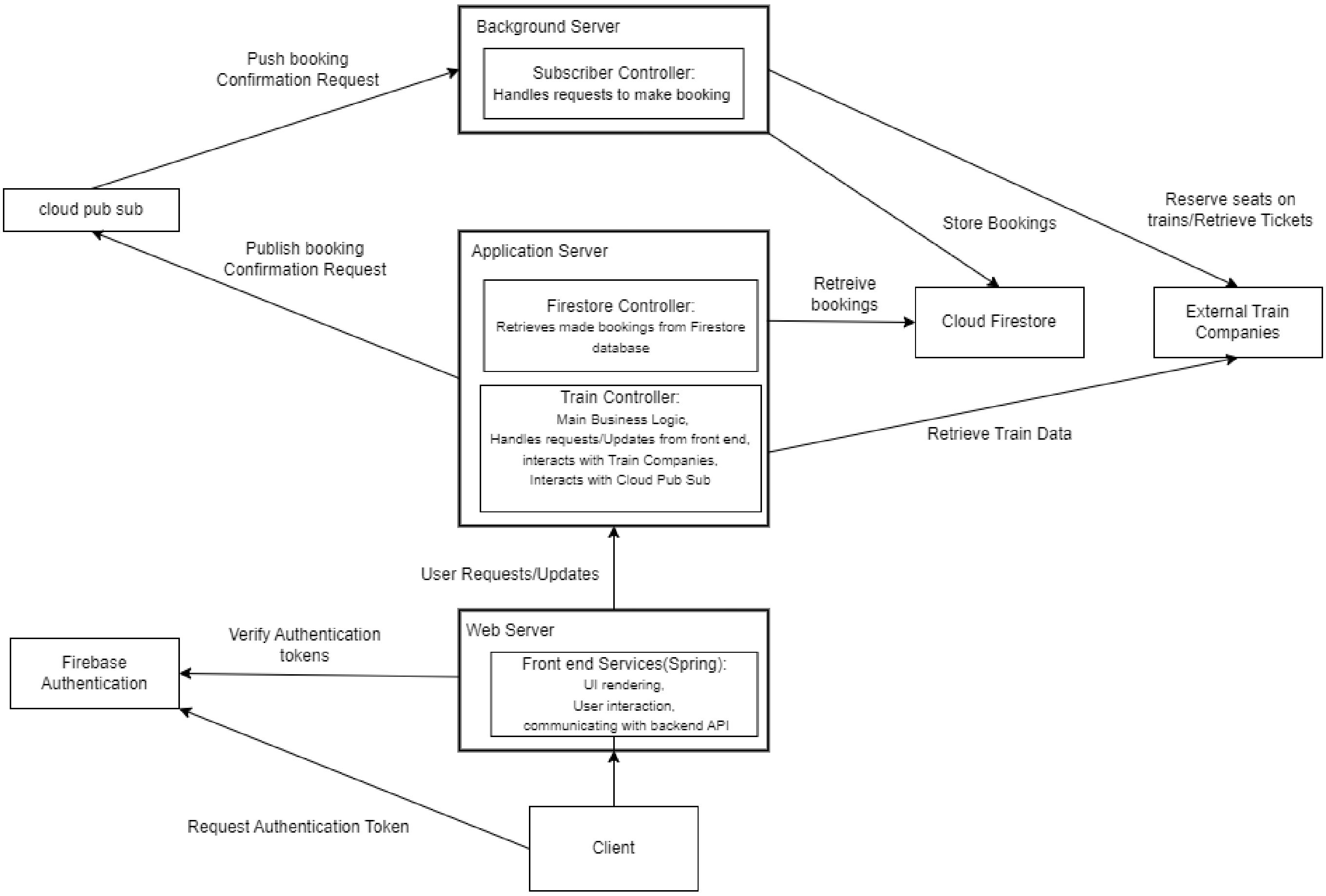
Background Server
Subscriber Controller:
Handles requests to make booking
Push booking Confirmation Request
cloud pub sub
Publish booking Confirmation Request
Store Bookings
Reserve seats on trains/Retrieve Tickets
Application Server
Firestore Controller:
Retrieves made bookings from Firestore database
Retreive bookings
Cloud Firestore
External Train Companies
Train Controller:
Main Business Logic,
Handles requests/Updates from front end,
interacts with Train Companies,
Interacts with Cloud Pub Sub
Retrieve Train Data
User Requests/Updates
Firebase Authentication
Verify Authentication tokens
Web Server
Front end Services(Spring):
UI rendering,
User interaction,
communicating with backend API
Request Authentication Token
Client

Figure 1: Deployment Diagram

Question 2

There were multiple places where we were able to leverage middleware to hide complexity and speed up development such as Spring WebClient, Firebase Authentication and Cloud Pub/Sub.

Spring WebClient: We used a Spring WebClient bean to allow us make REST requests to the API endpoints. The use of the bean is helpful in development as it allowed the automatic conversion of JSON responses into objects of classes that we specified, removing the need for the manual parsing of JSON responses. The use of the bean also hid the complexity of manually creating the HTTP requests, allowing the use of simple calls such as put(), get(), delete().

Firebase Authentication Firebase Authentication was used to authenticate users, giving them roles and specifying which endpoints were accessible to users with certain roles. The use of firebase authentication gave each user a token, encoded in which was their role. This token was attached to each request made to the api endpoints and authenticated by using Firebase authentication. As developers, all we had to do was to define the Security Configuration which specified which users were allowed access which endpoints based on their role and the Security Filter to decode the tokens sent with each request.

Cloud Pub/Sub In order to fulfil the requirement of having an indirect communication model, we utilised Cloud Pub/Sub. Cloud Pub/Sub is a service offered by Google which allows applications to register a topic, post messages to that topic to which subscribers of the topic get notified on a push or pull basis. In our case we wanted to transfer the booking procedure to a background worker and used Cloud Pub/Sub to do this. By doing this we were able to simply register a topic, assign the background worker as a subscriber to that topic and publish messages to the topic when we wanted to make a booking. This sped up the development and hid complexity as we did not need to create our own server with infrastructure to create topics, publish messages on the topics, deal with subscriptions and the associated notifying of subscribers.

Question 3

The indirect Communication between the main application, in our case TrainController, and the background worker, in our case SubscriberController, occurs at the create booking stage of the booking workflow. When a post request is made to api/confirmquotes the list of quotes sent to TrainsController in the request body is processed and the quote components e.g trainId, seatID, etc are extracted and used to make the URL used to reserve seats and retreive tickets. A list of these URLs are collected and converted to a utf-8 encoded byte-string, a pub-sub message is then created with this byte-string as the data component.

The publisher in TrainsController then publishes this message to the putTicket Topic that is listed on the Pub-Sub emulator. The pushTicketsub Subscription will then retreive this message and push it to the push endpoint; http://localhost:8080/subscription , using a http POST request.

SubscriptionController has a Springframework @PostMapping request mapper attached to http://localhost:8080/subscription and will thus receive the pub-sub message pushed to this URL. The message is decoded and the list of URLs is extracted. A Webclient builder will then place PUT requests on these URLs to reserve the desired seats and retrieve the resulting tickets. These tickets are then used to create a booking and this booking is passed to the Firestore database for storage.

Question 4

The data that we send in our Pub/Sub message to the background worker is a utf-8 encoded byte-string containing a list of URLs. These URLs reference web addresses on the train companies' websites where train seats can be reserved and tickets can be retrieved. Here we are passing by reference, the URL is a reference to the ticket objects, and these ticket objects are persisted on the train company website.

This approach to message passing definitely makes sense and has numerous benefits when compared to the alternative, publishing the ticket objects themselves in the Pub/Sub message, i.e pass by value. For example passing the URLs that reference these objects and not the objects themselves, means that the pub sub message size will be significantly smaller. This difference in size will add up over time and save money for the booking company as Google Cloud Pub/Sub charges users by the total byte size of their published and delivered messages.

Passing a URL reference to the ticket object also means that the background worker can handle the put request which reserves the seats and retrieves the object, reducing the load placed upon the our main application.

Question 5

Our solution to indirect communication improves scalability and responsiveness in a distributed /cloud context by decoupling the main application and some of the services it relies upon, in this instance confirming a booking. With this implementation the only task that the main application must complete is generating a list of URLs and publishing this list as a pub sub message.

The background worker will then handle the reserving of seats and the retrieval of tickets, the creation of bookings and the storing of these quotes in the database, all relatively time consuming processes as they involve contacting external services, leaving the main application to handle the next request from the user with a reduced waiting time, thus improving responsiveness.

This decoupling of the application and it's confirm booking service increases the flexibility of how the application can be scaled, allowing different components of the application to be scaled at different rates. For example since the tasks carried out by the background worker in our case are time and resource intensive, we can scale this service horizontally independently of the main application, by adding multiple background worker nodes that subscribe to the booking confirmation topic.

Question 6

Many of the user's actions would be unsuitable for indirect communication because they require an immediate return of objects or data retrieved from the Train Company websites, with a requirement that this data is received in real-time. If the data request to the Train Company must go through multiple layers of a Pub/Sub system, e.g being published to the Broker, being retrieved by the subscriber, handling the request and then resending to the main application, there will be a large increase in latency between requests and the return of their required data, decreasing responsiveness and thus reducing usability.

This increased latency could also result in stale data being presented to the user, causing them to operate on this stale data, potentially leading to errors that would further reduce usability. For example if the getAvailableSeats method was to use indirect communication and there was delays caused by the pub sub system, the list of available seats returned to the user may have gone stale and some of these seats may have been reserved, and thus made unavailable, in the mean time. if the user then attempts to make a booking on these unavailable seats the confirm booking operation will fail due to the inbuilt ACID properties, meaning the user will have to start the seat booking process again.

Question 7

There is no scenario in which the double booking of one seat is possible. We were given the assurance that the the train companies handle concurrent reservations, meaning if a ticket has been created successfully, it will not create another ticket for the same seat. This means that when we make a PUT request to a train company and receive a ticket, we know we are the only ones to have a ticket for that seat. Therefore, if we ensure that each PUT request made to the endpoint to book a ticket successfully returns a ticket before creating the booking, we ensure that the ticket cannot be booked twice.

We do this by detecting if there is an exception thrown during the ticket receiving process, maybe due to another user booking the seat at the same time. If one of these exceptions is thrown, our application will catch the exception and discontinue making the booking. At this stage it will also use a DELETE to free any other tickets that have already been received as part of this booking. This ensures either all or none of the tickets are booked.

Question 8

Role based access control simplifies the access of manager methods by giving each user a role and defining each endpoint access based on these roles. We define these endpoints and which roles users must have to access them in our Security Configuration file. With each of the endpoints linked to the roles of users that may access them, all we must do is identify these roles when users are querying the endpoint. This is done using Firebase Authentication.

In Firebase Authentication when the user signs into the application with their username/password, it is verified against the Firebase Authentication server. The server will send back an JWT identity token, encoded in which will be the role of the user. When the user is then using the application, each request to an endpoint will have their identity token attached. In this way all we must do is decode the identity token of the user at each request and check if they are allowed to access the endpoint and its associated methods / functions.

This decoding of the token and assigning the correct authority is done by the Security Filter upon each of the requests. This is done by decoding the JWT token, accessing the *roles* field, and specifying the users authority based on this role. This authority is then set in the security context and that's it. All we did was configure the Security Configuration and Security Filter with role-based access control, really showing how using role based control simplified the use of manager methods to authorized users.

Question 9

The Security Configuration and Security Filter would need to be changed as the new authentication provider will have different specifications that we would need to adhere to. Also the endpoint logic around manager access would likely have to be changed if the new provider did not support role-based access. We currently use JWT identity tokens that have the roles of the user encoded in them. If the new provider used a different form of token, we would need to change the token parsing logic in Security Filter. If the new authentication provider did not have role-based authentication, a different form of authentication for the managers would need to be employed. For example, this may be Auth0's access tokens which allow certain users access to protected resources. This could be applied to our endpoints which only managers should be able to access.

Question 10

The main way that our application copes with these failures is the use of the .retry() method on the Webclient class whenever a http request is sent to an endpoint of the Unreliable Train Company. This method will attempt to retry the http request if an error is encountered. For example, in our scenario the Unreliable Train Company will occasionally give 5xx Errors when requests are made, when this happens the Webclient will resubmit the request until the correct response is received or until the max number of retries is carried out, this number is specified by us in the brackets of the retry method.

Our get Trains method is also implemented in away that if all of the 9 retry attempts fail the user will not be shown an error but will instead only be shown the trains of the reliable train companies, meaning that unreliability in one company will not disrupt the services for others. This is achieved by using the webclient method shown below that returns an empty value when the webclient encounters an error. This empty value is checked for in the calling function, preventing the error from propagating downstream :
onErrorResume(throwable ->Mono.empty())

Currently the Unreliable train company will return an error approximately on 1 in 5 requests , with our protocol of attempting nine retrys this means that on average, approximately 1 in every 10 million requests will result in an error and with the average booking (presuming a booking of 2 seats with some comparing of different trains) consisting of 15 requests, 1 in every 666,666 user experiences involving the unreliable train company will result in an error, a number we judge as acceptable for our use case.

Since the number of retry attempts can be easily changed using the retry() method, we can adjust our protocol to attempt more retries if the Train Company becomes more unreliable. However although we can prevent errors occurring with more retry attempts, as the average number of retry attempts increases due to more prevalent errors, the responsivity of the application will decrease. A significant impact on services would start to occur when the success rate of a request becomes lower then the error rate, meaning on average every request will require at least one retry.