School of Computer Science
University College Dublin

COMP41720

Distributed Systems

Final Assignment

PeerRate: Transactional Anonymized Feedback from Coworkers

Assignment Report

Aness, Hongpeng, Anshu, Anju

What is PeerRate?

A shocking 65% of professionals are thought to be suffering from impostor syndrome in the workplace [1], a syndrome where the person constantly suffers from self-doubt and feels like they are far underqualified when compared to their peers [2]. It is thought that a significant contributor to this syndrome is lack of sufficient positive reinforcement from peers, making the individual feel as if they are looked down upon.

Additionally, the lack of knowing a person's weak points means that the individual may struggle to understand where they need to improve, further leading the individual to feeling like an impostor.

Both of these issues would be resolved if a system existed whereby peers are able to give honest feedback to one another – both positive to uplift them and constructively negative to point out where they need to improve. But how can I be honest with someone when my feedback of them can sever our relationship?

PeerRate is the answer. It is based on the concept of anonymised feedback, where any user can find another individual in their company or outside whom they know and help them on their journey towards self-perfection. As opposed to corporate feedback, this feedback is not a performance review, and will never be read by any manager or in any professional setting. It is solely meant to help peers identify their strong points that they should embellish, as well as their flaws which may be glaring to everyone around else but not to them.

But who would go through the effort of reviewing others? This is where the transactional element of PeerRate comes in. You are notified of when someone reviews you, but you are unable to read the review until you review someone else yourself. This helps ensure the natural exchange of reviews and ensures a system of giving is in place to allow for receiving.

PeerRate understands that in certain cases, people who review you may not have had the best impression, and that their feedback may be an outlier which should not really be relied on for self improvement. That's why it includes an aggregation system whereby the most common themes from all the feedback obtained is shown to the user. This helps reduce the outliers and gives users reliable pointers to work off.

PeerRate is based on a culture of improving oneself whilst uplifting others, being truthful and fair with others, along with telling them why you value them, so that we can all feel more confident in our abilities, work on ourselves, and feel less like impostors if possible.

PeerRate Architecture

Overview

Having understood the premise of the system being designed, the following architecture was concocted:

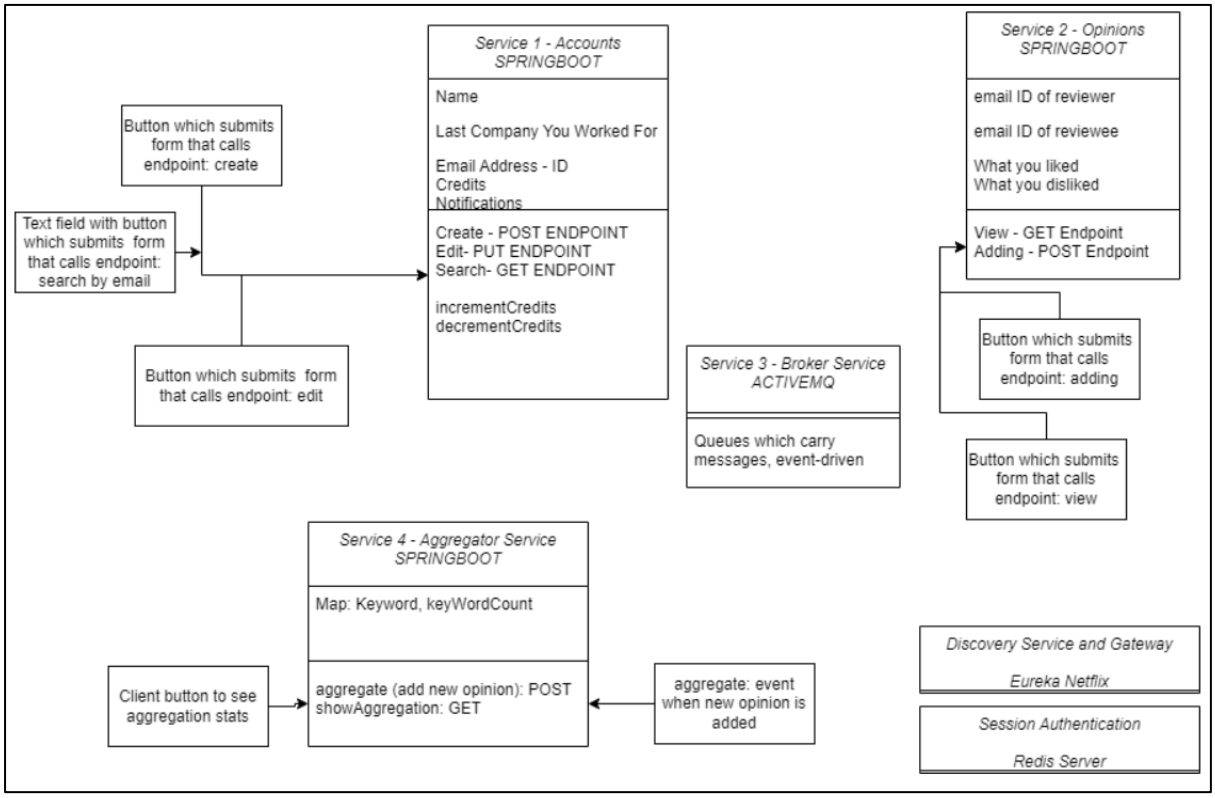


Figure 1: Overview of PeerRate Architecture

As can be seen in figure 1, a microservices based approach was opted for, using three main RESTful microservices along with their data storages; accounts, opinions and aggregator; implemented using SpringBoot. These three services discover each other and can redirect to each other through the use of a Eureka service and a Eureka Client Gateway. An event-driven architecture is used with the broker service that relies on the Java Messaging Service (JMS), allowing for communication between the microservices to fetch and update data. A Redis server is used to maintain session data across all microservices, which is mainly used to unify authentication across all services instead of requiring users to log in to each service separately. Traditional SQL based databases are created separately for each of the three microservices to store the data pertaining to each one. The services are then all containerised using docker, which allows for easy deployment that is scalable and fault tolerant.

SpringBoot was chosen for this task as it supports Redis sessions, ActiveMQ JMS, RESTful controllers, and has a great Netflix Eureka library.

Over the remainder of this report, each of these components is explained in more detail, as well as the pros and cons of the choice made.

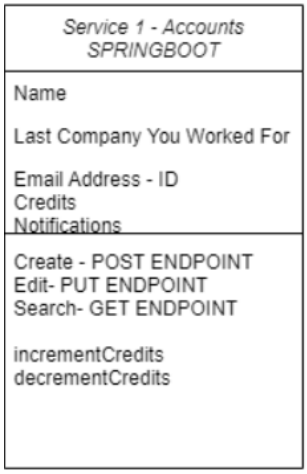
Team Contributions

As per the assignment instructions, each person on the team was responsible for one of the services:

Team Member	Responsibility
Aness	Event driven architecture, interactions between services, front end development, selecting idea and coming up with architecture, session and authentication, containerisation
Anju	Aggregator service and endpoints
Anshu	Opinions service and endpoints
Hongpeng	Accounts service and endpoints

1. Accounts Microservice and Redis Session

At its core, the accounts microservice is a simple RESTful SpringBoot service which stores all information relating to a user on PeerRate, as well as serving the front-end pages relating to those attributes.



In this service, the email (primary key), password, name, company, notifications, and credits of each user are stored.

It also is responsible for showing search results for individuals in a certain company so that they can be reviewed by their peers.

The "credit" attribute is in place to validate whether or not a user can view an opinion written about them. The credits are incremented when the user adds a review about someone else in the opinions service. They are decremented when the user reads a review about themself.

The accounts service is responsible for user authentication. It is here that users log in, sign up, and log out. On signup, the username of the user is placed in the session, and other services use the presence of the username in the session as the marker that proves the user is logged in. If not present, the services redirect to the login page in the accounts service and deny the user access to their contents. On logout, the session is invalidated, allowing for another user to log in/sign up.

A possible improvement over this implementation is the use of the in-built Spring Security, which was not used in this case as it was beyond the scope of the assignment.

Since Redis is fast access and uses key-value storage, it is perfectly suited to storing the session data and sending it to any of the three microservices as needed. Its decentralised nature and the great ease by which it is deployed, maintained and containerised means it was the correct choice for this application. This architecture can be seen in figure 2:

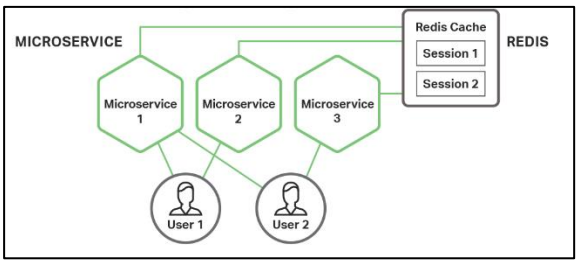
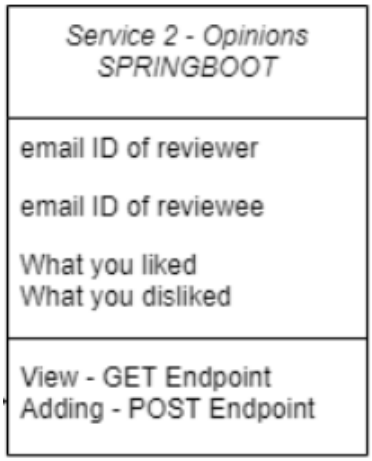


Figure 2: Redis Session Architecture, Courtesy of StackOverFlow

An alternative approach could have been taken with JWT tokens, which can be arguably safer but is more complicated to implement and scale as more microservices are added. The beauty of this implementation resides within the fact that the authentication of a user to every new service becomes as simple as registering the new service with the Redis Server.

2. Opinions Service

Similar to the accounts service, the opinions service is implemented as a RESTful SpringBoot service for the aforementioned reasons. It stores information about each of the reviews or "opinions" written by the users.



The information stored about each opinion is the timestamp of each opinion (primary key), the email of reviewer and reviewee, the actual feedback text (positive and constructive) along with a flag that marks whether or not the feedback was read by the reviewee.

There are endpoints in the service to CRUD the attributes as required.

The opinions service also hosts and manages the main dashboard page of PeerRate, which is the hub for the user to read and write reviews as well as access the aggregates and notifications.

The flows for adding and reading opinions is heavily dependant on the messaging service and will be discussed in more detail there. However, in essence, an event is fired once the user creates or attempts to read an opinion, leading to messages to be exchanged between the opinions and accounts service as required to ensure that all microservices are on the same page.

3. Aggregator Service

As with the previous two microservices, the aggregator service is also a RESTful SpringBoot application. Its main goal is to efficiently extract keywords in order to identify patterns in the feedback of the user. The design philosophy of this service was such that it is as efficient as possible, meaning that:

1. The service only stores the results of the aggregation and not the full text of all opinions written about the user
2. The aggregation of the opinions is only done when the user reads a new opinion that they hadn't read before. Otherwise, the aggregator simply returns the results of the previous aggregation.
3. Not all users will read their aggregation, and so it does not make sense to automatically aggregate each time a new opinion is read. Rather, the aggregation is done on demand when the user decides to view the aggregation. This is possible due to the event driven nature of this architecture and leads to great performance savings as PeerRate scales.

An open source key-phrase extraction tool called RAKE is used to perform the extraction as needed, and the results of this are stored as per above. The aggregator service stores these results, the user email along with a flag which determines whether reaggregation should be done or not (when a new opinion is viewed).

4. Inter-Service Messaging

In what was the most challenging part of the project, a scalable, robust and intelligent system for communication between the microservices needed to be devised. Regardless of the technical system used, a consensus was formed amongst the team that the architecture needed to be event-driven. This is because of the event driven nature of PeerRate; an example being the flow involved when a user wishes to view a review:

User wishes to view an opinion of them → User must be validated → Credits of user must be greater than zero → User has now viewed an opinion, which means the aggregator should dynamically aggregate on the next request for aggregation

As can be seen in the example above, all three microservices have a role to play in the event that an opinion is viewed. As such, it makes sense to have queues connecting consumers and producers in each of the services that need to communicate with one another.

The philosophy of using this messaging was to only use it when strictly necessary – if a redirect to another service made more sense than porting over the information required over the queues, then a redirect should be made. This is to for the sake of system performance as traffic grows. The architecture opted for was as follows:

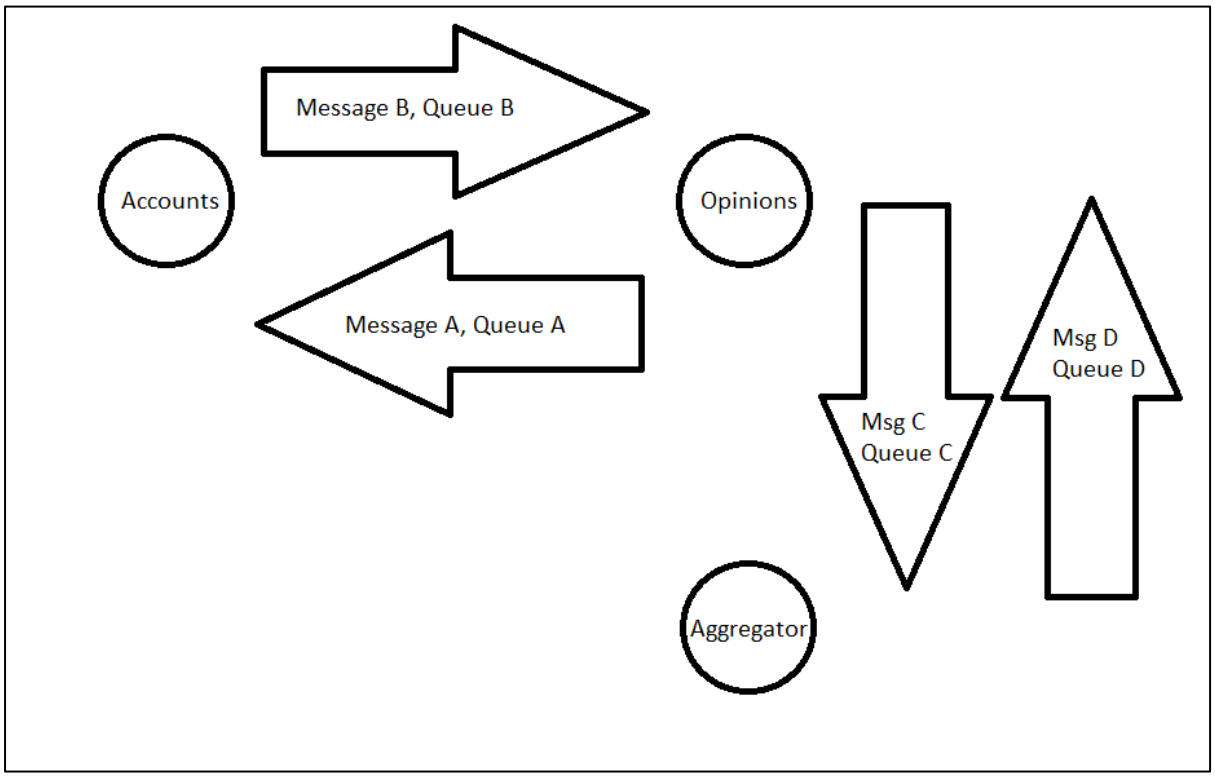


Figure 3: Java Messaging Service Architecture

As can be seen in figure 3, a queue connects each of the services that need to interact with one another, with the arrow direction denoting whether the service is a consumer or producer in the queue in question. It is important to note that queues were chosen in lieu of topics as that is the most efficient in terms of resources for the current system, but as the system scales and more microservices are needed, it would be prudent to set up topics where the event being triggered can be propagated to all consumers as required. It just happens that, in this case, each "topic" only has one consumer, so it makes more sense to use a queue.

The queues are labelled A, B, C, D to coincide with the types of messages sent along them, as each queue has a different message class which corresponds to the type of information sent in that respective direction.

When an event is triggered, it is sent to the respective recipient and the data is updated as required. Notifications are a great example of this, as these are the types of notifications that may occur:

(Reminder) Notifications are stored in the Accounts microservice

Notification	Triggered By	Propagation Flow
Notify all users in a company when a new user from their company joins.	A new signup.	No JMS required: signup and notifications both occur within accounts service
Notify user that they earned a new credit by reviewing their peer.	Writing a peer review	Peer review registered on Opinions service; Accounts is triggered to add the notification. Aggregator is triggered to set reaggregate flag to true.
Notify user that they used a credit by reading the review of themself	Reading a review of yourself	Peer review reading triggered on opinions service, needs to be validated by accounts for correct email and enough credits, and returned to opinions service to either display opinion or insufficient credits message

It is important to note that the use of the messaging service is a direct consequence of the scaling of PeerRate; a monolith would simply have all data in a single database and so would not need this level of complexity. But on the other hand, the distribution of the services in this manner allows for greater performance and control of the application, as the more demanding microservices (like opinions) would likely be run on more powerful servers that can handle the demand, as opposed to a monolith which would not be as efficient as traffic grows. Additionally, this approach allows for easy addition of more microservices in the future without much significant effort if topics are used – the same events will fire and the microservices will simply be coded to react to them as required.

Due to the unfortunate fact that JPA, the module used for data storage and retrieval, is not thread safe, threading was not implemented in the solution. But ideally, threading would be used to decouple the sending from receiving in these situations, which further increases performance compared a purely sequential monolith.

An alternative approach could have been an event-driven RESTful architecture as opposed to messaging, which in hindsight seems to be a powerful contender for the inter-communication technology of choice. This is because it involves far fewer moving parts than messaging, fewer dependencies and so more robust and maintainable code as the application scales.

5. Service Discovery

Since each of these microservices is its own application, it becomes very difficult to try to route from one service to another without knowing the ports of the other services. This is critical especially in cases where load balancing is used, where multiple instances of the same service are run on different ports. To counteract this, Netflix Eureka along with an API Gateway service are used:

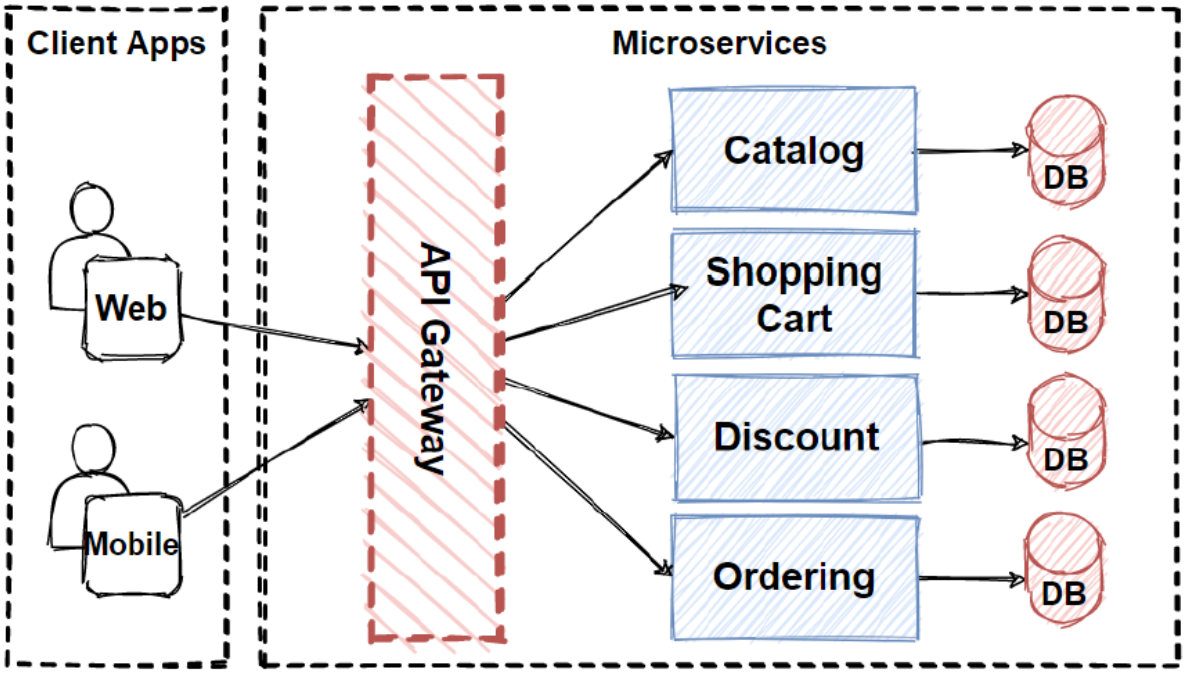


Figure 4: Gateway Routing Architecture (Credit: Mehmet Ozkaya)

With this approach, only the ports of the gateway and Eureka server need to be known to each microservice. The port number of the Eureka server is needed so that all the microservices can register with it, and the port of the gateway needs to be known so that the microservices can route between each other by simply calling the relevant path after the gateway port. This implementation is critical to the operation of the distributed system to be as cohesive as that of a monolith, where users on any platform can access all microservices on one gateway base URL.

6. Containerization

Now that the entire application in its many microservices was built, it becomes time to place each of the microservices in its own container. Aside from the security benefits that this offers, the containerization of the application is critical to its deployment and scaling. Using docker compose, the entire application in its incredible complexity can be deployed with a single command.

More importantly, the settings in the docker compose file establish a health check ping on each of the services every 30 seconds to make sure they are still up, and automatically restart the services that go down for any reason. This is essential to ensure that the service is fault tolerant and highly available, immediately relaunching in cases of failure.

Additionally, the number of container instances can also be set when using the “docker compose up” command, meaning that multiple instances of the same image can be deployed, which allows for backup images in cases that one is overloaded. This all means that PeerRate is a highly efficient and scalable distributed system that can very easily be scaled up to accommodate millions of users.

References

[1] https://www.prnewswire.com/news-releases/imposter-syndrome-affects-65-of-professionals-new-study-finds-301295516.html

[2] https://www.verywellmind.com/imposter-syndrome-and-social-anxiety-disorder-4156469