Team FinTech Service Aggregator

Distributed Systems - Group - 12

James Clackett	Cormac Egan	David Mallon	Harry Ó Cléirigh
15732749	18309231	08597596	12468998

Synopsis:

Describe the system you intend to create:

The system is designed to offer a distributed loan generation application. The primary purpose of this application is to allow for the comparison of various types of loans (student, car, personal, and home) between numerous providers. The system is designed with scalability in mind, opting for a microservices based system which can support varying numbers of users.

What is the application domain?

The application is focused on loan generation and comparison. It involves the generation of credit scores and the creation and comparison of loan quotations between several providers for various loan types.

What will the application do?

The application presents the user with a form to fill out, requesting their personal information, details on the loan they wish to obtain, and questions relating to their credit worthiness. From there, this information is sent as a JSON object via a gateway implemented in Express, to a Credit Score Calculator. The credit score is either retrieved from a Redis cache or stored in both the Redis cache and a MongoDB database for future use. The JSON object is returned to the gateway with the addition of the credit score. From there, the gateway routes the JSON object to the appropriate service providers using RabbitMQ. These offers are consolidated and sent back to the user, showing the various loan options available to them.

The application is implemented using a number of design patterns appropriate for its purpose. The MVC pattern has been used in both the gateway server and the loan provider services. At a higher level, the overall system architecture loosely adheres to the microservices structure (strict adherence was not suitable for our goal). This allows for greater levels of fault tolerance and scalability. The use of docker further enhances fault tolerance and also improves the ease of distribution for our system. As we will discuss later, kubernetes was considered to introduce an elasticity to our application. While we ultimately decided against this due to time constraints, it was nevertheless a factor in opting for a microservices pattern, which would be well suited for automated scaling.

Technology Stack

List of the main distribution technologies you will use

- **React.** React was used to provide a clean and intuitive user interface for branches/clients. As a framework, it makes use of stateful objects and the built-in fetch library to manage its on-screen content. React was chosen for its ease of maintenance, debugging, and ability to create clean and performant UIs in a timely manner.
- **Express.** Express was used to build the Gateway server of our system. It is responsible for handling client requests from the frontend in the appropriate manner. In summary, client requests are received, the relevant credit score is found and both are sent to the various loan services. Express was chosen for its ease of use and its lightweight build. It is an extremely appropriate framework for a simple application that has a potential for future growth. Its compatibility with ReactJS was another important consideration.

- **Docker.** Each of the services of our application was containerised via Docker. This was done to ensure consistency across different environments and to simplify deployment and scaling, as containers are device agnostic.
- **Flask.** Flask was chosen as it is lightweight and flexible, ideal for the tasks we were carrying out. Each of the services using Flask only deal with a few POST and GET requests (for example the credit score calculator has two endpoints, it handles POST requests on /calculate_credit_score and GET requests on /get_all_data). As we adopted a microservices approach we wanted each service to be as lightweight as possible and so there was no need to opt for a more heavyweight web application framework.
- **Redis.** Redis was used as a cache for our credit scores, with a hash key based upon the name and date of birth (DOB) of the user. The cache helps avoid redundant calculations and improves the performance of the Credit Score Calculator. We opted for Redis given that it accommodates various persistence options (how long items stay in the cache) as well as the ability to support master-slave replication, providing the opportunity for enhanced scalability if required.
- **MongoDB.** MongoDB was selected to provide permanent storage for the JSON objects sent to the Credit Score Generator. The entire contents of the database is available on the /get_all_data endpoint, which provides us with the option to perform analysis on the users of the application. We opted for MongoDB, which is a NoSQL database, because we believed its document-oriented data model was well-suited for our goal of storing JSON objects. A conventional relational database with a rigid schema was not needed and would not provide the scalability and flexibility offered by MongoDB.
- **Eureka.** Eureka simplifies the process of service registration and discovery in microservice architectures. Rather than implementing our own HTTP service advertising, we opted for Eureka as it offers built-in configuration, load balancing, and many other features. It is a robust service advertisement solution.
- **RabbitMQ.** Messaging queues were implemented to facilitate communication between the gateway app and loan service registries. Messaging queues are suitable for a task like this as there is an unknown amount of demand for loan services. Traditional HTTP requests can become cumbersome when there are multiple replicas of a service in use. It becomes necessary to keep track of multiple addresses and ports. With messaging queues, all replicas of a service can be made to use the same queues, simplifying development. RabbitMQ was chosen for its compatibility with ExpressJS.

System Overview

Describe the main components of your system.

Application

- **Frontend (React)**
  - The system's frontend is built using ReactJS. The frontend consists of two main React components: ‘Summary’ and ‘Form’. *Form* provides a HTML form built using the built in JSX library and allows the user to input various information about themselves and the type of loan they desire. A button is provided with the form to trigger the submission (via HTTP POST) of the input information to the gateway server. This button is also responsible for updating the useState values *fetchedQuotes* and *hasGottenQuotes.* When these values are set, the UI is refreshed. If hasGottenQuotes is true (i.e. the gateway has provided valid and non-empty quote offers), then the Summary component is displayed instead of Form. *Summary* is responsible for simply displaying the list of offers that have been received from the gateway.
- **Gateway (Express)**
  - The Gateway server is essentially the middle-man of the system. It is responsible for handling the requests of the front-end, requesting credit scores from the credit score calculator, and sending loan requests to the loan provider services. A typical use cycle would be as follows:
    - User submits data from their browser

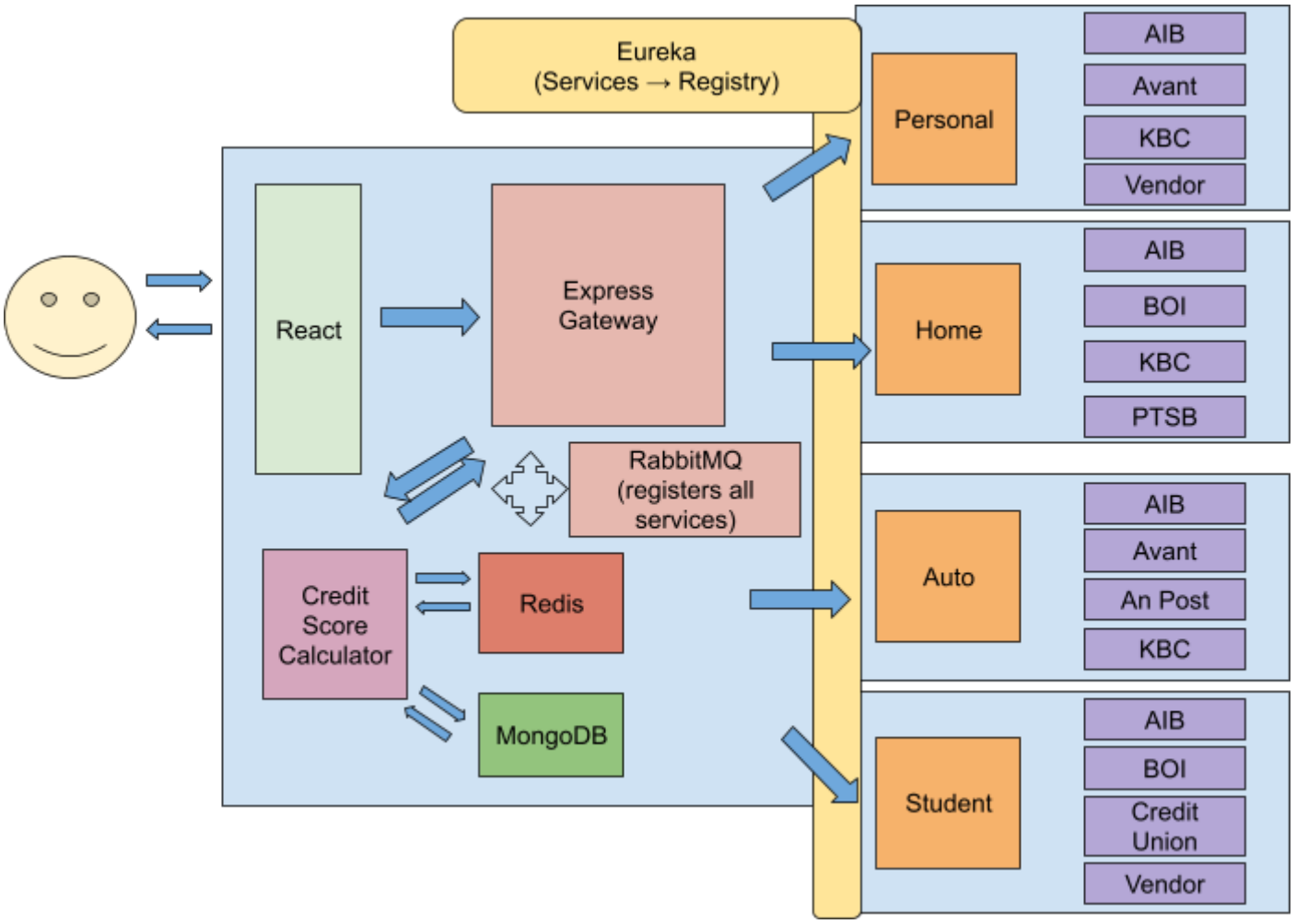
        - The data is sent to the credit score calculator who returns a score to the gateway
        - The gateway then places the client info and credit score into a messaging queue where it will be distributed to the relevant loan providers.
        - Another messaging queue for offers will be listened to. When all offers from loan providers have been received they will be sent to the user.
    - It follows the MVC pattern with a controller that handles requests, and a means to communicate with the Credit Score Calculator (which can be seen as the gateways data layer). In production, the server would also be responsible for providing the view too (i.e. the 'V' in MVC), however we have opted to run our React 'view' on its own development server.
- **Service Discovery (Eureka)**
    - Eureka is a service discovery tool used in microservices architecture. It allows our services to register themselves and discover other services. This is crucial in our microservices project as we have many small, loosely coupled services that need to communicate with each other.
        - Registration: Each of our services, when they start up, registers itself with Eureka. This is done in the server.js file of each service. For example, in the car-loan-aib/server.js file, the service registers itself with Eureka using the eurekaClient.start() method. If the registration is successful, a message is logged to the console (for ease of debugging and visibility that everything is working as it should be).
        - Discovery: Once any of our services are registered, it can be discovered by other services. This is useful when one service needs to communicate with another. For example, this might occur when the gateway service needs to route a request to the car loan service. It can ask Eureka for the location of the car loan service and Eureka should provide the current IP and port of the service.
        - Health Check: Eureka also periodically sends health check requests to each service. If a service does not respond, Eureka will remove it from the registry. This helps to ensure that services are always able to find and communicate with each other, even if some services go down or are scaled up/down.
- **Credit Score Calculator (Flask)**
    - The Credit Score Calculator receives a POST request on /calculate_credit_score endpoint. This contains the JSON object sent from the form on the frontend via our gateway. From there, the JSON is parsed and a hash key is generated based on the person's name and DOB. We check whether the key is in the Redis cache and if so whether the value associated with the key (the JSON) is the same (i.e. has the user provided the exact same details previously). If so, we can use the item stored in the cache and send out the credit score. Otherwise, we calculate the credit score (based on age, missed payments frequency, credit utilisation, credit history, debt to income ratio and use of new credit), send this score out and save this in cache for future use. MongoDB acts as our permanent storage. We implement a similar logic of checking whether the key is there and whether we need to update or add an item. The entire contents of the database are provided to a GET request on the /get_all_data endpoint.
- **Cache (Redis)**
    - As described above, the cache is used to store the credit scores for each user. This reduces the use of redundant calculations (as the credit scores are not reliant on the loan type / value etc and thus will be reused for multiple applications) and improves the responsiveness of the application. Items are stored in the cache for 24 hours.
- **Permanent Storage (MongoDB)**
    - MongoDB is used to store user data. There is one record per user. MongoDB's document based setup is ideal for our purposes of storing JSON objects. MongoDB allows for horizontal scalability via sharding, ensuring that our application is capable of scaling to meet changes in demand. The entire contents of this database are provided on the /get_all_data endpoint to allow for analysis of user profiles.
- **Loan Registries**
    - For each loan type (Student, Home, Personal, Auto) there is a registry service built (also using ExpressJS). The function of these services is to handle messages received from the gateway and send

them to the appropriate loan providers. The registries listen to registration messages received from providers and add each to a list of listening services.

- The registries listen to a messaging queue for loan requests and for each relevant message, requests a loan offer from all of their listening provider services. A list of quotes is appended to for each provider's response, and when all have been submitted, a message is added to the offers queue and will be returned to the gateway.

- **Loan Providers**
  - The loan providers are responsible for calculating the actual loan offer a client will receive. The provider receives a HTTP request from its registry and uses this to calculate an offer. Client information and loan type is considered, and the calculated response is returned to the parent registry.

Include your system architecture diagram in this section.



Explain how your system works based on the diagram.

In this diagram each of the blue boxes represents a docker container. The process begins with the user filling out the form which is provided via a React webpage, supplying their personal information, credit history, and information about the loan they wish to provide. From there, this information is sent as a JSON object to the Express Gateway and forwarded to the Credit Score Calculator as a POST request to the /calculate_credit_score endpoint. A hash key is generated based on the user's name and DOB and the Redis cache is checked to see whether there is a record for the user. If there is, the credit score will be retrieved from the cache, if not the credit score will be calculated and this information will be added to the cache / the record will be updated (if there was a record for this key but with different details relating to credit worthiness). Simultaneously, the MongoDB database will be checked to see whether there is a record for the user and whether a record needs to be added or updated. MongoDB provides persistent storage for all user applications (there is one record per user in this database).

Having calculated the credit score, this updated JSON is sent back to the Express Gateway. From there the new credit score is appended to the JSON client object. A custom Express middleware function *determineServiceProvide* uses the client request to determine which of the loan services should be requested. Once this has been established a message is sent to the chosen provider using the RabbitMQ messaging protocol and queue. The gateway listens to an 'offers' queue, while the loan services each have their own queue to consume (car-loan-queue, home-loan-queue etc).

The loan services are created by and run in a separate set of docker containers. For each type of loan (auto, student etc.) there is a registry (the orange box) responsible for keeping track of a list of loan providers. These loan providers register with the registry on startup using Eureka and represent the various companies or institutions offering loans. The provider services wait for the registry to send them client requests and on receipt, calculate a loan offer that is returned to the registry. When the registry has waited for a designated timeout period, it will send all its offers back to the gateway via another messaging queue. Once the gateway has received offers in its queue, it creates a list of all and responses to the initial React request with this list.

Explain how your system is designed to support scalability and fault tolerance.

Scalability and fault tolerance are at the heart of this application. The decision to divide our application into several docker containers allows us to scale up different parts of the application as required.

With regards to the Credit Score Calculator, the Redis cache improves the scalability of the system by guarding against redundant credit score calculations, thus reducing the load placed on the system. Redis offers master-slave replication, offering us the ability to further enhance the fault tolerance of the system. Similarly, the permanent database which is implemented in MongoDB offers horizontal scaling through sharding and so it can adapt to meet changing demands. MongoDB also supports replication (where data is replicated across multiple nodes), which adds to the fault tolerance of this application.

The use of messaging queues further enhances both scalability and fault tolerance. Their asynchronous nature facilitates the arrival of offers at various times without impacting the final result. More importantly, they are resistant to system failures as messages will remain in the queue if other systems fail. This greatly reduces the risk of data loss. In terms of scalability, while the benefits cannot be seen clearly in a small-scale application, queues are very easy to scale horizontally. If a higher level of communication is anticipated, it is a simple matter to add more nodes and tell them to listen/send to the existing queues.

Contributions

Provide a sub section for each team member that describes their contribution to the project. Descriptions should be short and to the point.

James Clackett

- Small amount of middleware development and cleanup in the gateway server and loan services. My goal was to enhance readability and modularity.
- Implemented a RabbitMQ messaging queue system to facilitate distributed communication between the gateway and loan service providers. This is used in our application to decouple the gateway service from the loan services, allowing them to communicate and process tasks asynchronously. Incorporating this feature provides our application with more scalability and resilience, as services can be added, removed or updated independently without affecting the rest of the application.
- Duplicated and implemented a number of the finalised loan services.
- Troubleshooted various issues relating to docker networking.
- In-code documentation of gateway and the services mentioned above.

Cormac Egan

- **Credit Score Calculator:** I was responsible for the implementation of the Credit Score calculator, which takes in the JSON object, parses it, and generates credit scores for users based upon age, missed payment

frequency, credit utilisation rate, credit history, debt to income ratio, and use of new credit. I reward / penalise these factors as appropriate and then provide these credit scores back to the Express Gateway. The credit score calculator was implemented via Flask and handles POST requests on the /calculate_credit_score endpoint and GET requests on the /get_all_data endpoint.

- **Redis cache:** I implemented the Redis cache used in the credit score calculator. I wrote a function to generate a hash key based on the person's name and DOB and this is used as the key for both the Redis cache and the MongoDB database. The cache is used to prevent redundant credit score calculations and to improve the performance of the application.
- **MongoDB database:** The MongoDB database is used to provide persistent storage, allowing us to run analysis on the users of our application. The logic relating to adding / retrieving / updating both the MongoDB database and Redis cache has been explained previously, the essential point is that we check whether or not the item is present and whether it requires updating and act accordingly. I also implemented error handling in case of database inaccessibility. As previously mentioned a noSQL database was ideal for our purposes of handling JSON objects.
- **Containerisation:** I was also responsible for containerising the Credit Score Calculator as well as the Redis cache and MongoDB database which it relies upon.

David Mallon

- I was responsible for preparing the skeleton for our loan provider services which provided information and calculations related to the various types of loans (mortgage, personal etc).
- This was originally prepared using python but we decided to implement them as Node.js applications for consistency (Harry took ownership of this change). The services are containerised using Docker which allow them to be easily deployed, scaled and managed.
- The services expose an API which the gateway service can call. The API then accepts loan details as input and returns loan calculations as their outputs. The services interact with the rest of the application through the gateway service. When a user submits loan information through the front-end service (localhost:3000), the front-end service sends a request to the gateway service. The gateway service then routes the request to the appropriate loan service based on the request details. The service performs its calculations and returns the result to the gateway service, which then forwards the result back to the front-end service to be displayed to the user. The service also registers itself with the Eureka server when it starts which allows the service to be discovered by other services and the gateway service to route requests to it.
- I was responsible for implementing service docker containers.
- I also assisted with troubleshooting issues with docker networking, Eureka and the way these worked with our loan services.

Harry Ó Cléirigh

- **Front-End Service Development:** I was responsible for creating the user-friendly front-end interface using React in a Node.js environment. My work focused on designing an intuitive interface which creates the initial loan request object used in our backend application to determine loan values for each partnered vendor.
- **API Gateway with Calculator Proxy:** I developed the initial API gateway, implementing a middleware proxy specifically for the calculator service. This role was crucial in ensuring efficient routing and handling of requests between the front-end and calculator service before a quote is requested from a service cluster.
- **Service Clusters Creation with Express:** My role involved constructing service clusters using Express, a Node.js web application framework. This contribution was instrumental for our project in organising and managing various loan services, contributing to the robustness of our microservices architecture.
- **Integration of Eureka for Service Discovery for Service Clusters:** To improve the system's scalability and resilience, I updated our service clusters to include Eureka for enhanced service discovery and utilisation. This integration was a significant step in evolving our microservices architecture, ensuring better service orchestration and reliability. This also required some light scripting as well, to ensure that the eureka server is up and running before any services are registered.

Reflections

What were the key challenges you have faced in completing the project? How did you overcome them?

One of the key issues we faced in completing the project was coordinating our efforts to ensure that we adhered to the same architectural approach. We restructured our application during the project, which resulted in merge conflicts and required refactoring our code. We overcame this issue by having several meetings in order to consolidate our contributions and resolve our merge conflicts. We used screen sharing on Discord in order to help with this process.

Another challenge was the time of year in which the project was due. It was sometimes difficult to coordinate our work efforts as each member of the team had wildly varying schedules and availability. Git version control proved to be a vital tool for isolating our activities into multiple branches, enabling each member to work on their specific tasks without worrying about others. In addition to this, dummy systems and data were widely used to mimic any functionality that had not yet been implemented due to schedule differences. This approach has worked well for us as it prevented certain functions from being at a standstill while waiting on others to be built. It also acts as a useful means for spotting potential architectural oversights before genuine code is written.

It could also be said that choosing the system's architecture itself was a challenge. Many of the approaches and architectures presented in this module were unknown to us at the beginning of the semester. Of course we were familiar with REST and SOAP but many of the other approaches shown were alien. When initially designing our application and how it would work, it was difficult to determine which approaches were appropriate for our goals. This challenge was overcome in a number of ways. Firstly, by reviewing our practicals and discussing the various pros and cons of the solutions shown in each. We discussed what we liked and what we did not, and what each could offer us. Secondly we brainstormed various architectures in order to fully visualise the shortcomings or benefits of each. We also discussed the various solutions with our TAs. With a well rounded idea of our needs as a result of these actions, we were able to come to a decision on the architecture seen in our application and detailed in this report.

What would you have done differently if you could start again?

If we were to redo this project, we may have spent more time discussing container orchestration and the possibility of using Kubernetes to handle the scaling, load balancing, and resource management of our application. It would also have been interesting to fully explore the relationship between Kubernetes and Docker. There are many ways to build a scalable and fault tolerant system with these two technologies and perhaps in a longer timeframe it would be possible to implement various versions of our system and compare their performance. In a similar vein, it would also have been very interesting to create another version of the system using RPC, RMI, or SOAP to assess the pros and cons of these against our more modern approach. As with all software engineering projects, there are an infinite number of things that could have been done differently, but these are the ones that stand out most to us.

What have you learnt about the technologies you have used? Limitations? Benefits?

It was our first time working with a Document-based NoSQL database, MongoDB. We felt that MongoDB was ideal for our purposes, given that we were working with JSON objects as well as its extensive support for scaling and fault tolerance. As we were performing rudimentary operations on the database (simply retrieving all records) MongoDB worked well, however we are aware of the CAP theorem and that MongoDB opts to support partition tolerance and availability at the expense of consistency. It was not of the utmost importance that our database was always consistent so this was acceptable, however this is a limitation which is worth noting.

Another limitation may be the use of a messaging queue. In our application, it was introduced primarily because it provides a scalable solution for communication when the number of future services are unknown. In theory, the number of loan types/providers in our system could grow significantly. In such a case, a messaging queue would add structure to, and greatly simplify communication with the gateway. So as a matter of good practice, it was a useful addition. However the reality is that our application has a definite number of services that will not change. For a small-scale application a messaging queue could be viewed as overkill in comparison to using simple HTTP requests which can be achieved in very few lines of code.

One important benefit of the technologies we have used (as a whole) is their high level of agnosticism. In the beginning of this module we used technologies like Java RMI and SOAP. These were highly dependent on the underlying software or Interface Description Language (IDL). In contrast, in our application, we have used multiple languages and frameworks, as well as HTTP requests and messaging queues. These various tools are working together with no issue which would have been impossible with the older approaches we have learned about.

Readme / How to run the application

- If using a Mac laptop, turn off Airplay Receiver on *System Settings > General > AirDrop & Handoff > Airplay Receiver*.
- Open two tabs in your terminal and cd your terminal 1 window into the root of the project and cd your terminal 2 into the *loan-services* directory.
- In the terminal 1 window, run "*docker network create shared_network*". Once the *shared_network* network has been created, cd into the *application* directory.
- Run "*docker build*" in the terminal 1 window and then in the terminal 2 window.
- Run "*docker compose up*" in the terminal 1 window and then in the terminal 2 window.
- Then open a browser and submit a fully populated form using *localhost:3000*
- You should see the response with quotations in the browser.