SSE Team Project II - Netflix Content Discovery Web Application

Sachpreet Nehal, Zeeshan Ahmed, Jenarth Pathmakanthan, and Raymond Guo

Introduction

Our Netflix Content Discovery Web Application is designed to inform users of netflix content unique to various worldwide locations, such as the US and the UK. Netflix customers frequently discover that particular films or series are only available in certain locations due to licensing agreements. Customers may quickly access content libraries from multiple countries via a VPN finder, opening up a wider selection of entertainment possibilities.

We have adapted two methodologies, microservices and containerisation, to transform how our application has been planned, built and deployed, allowing scalability, reliability and flexibility. Rather than creating a single complex programme, we divided functionalities into smaller, independent services. Each microservice functions as a self-contained unit with its own distinct purpose, allowing for agile development, simple maintenance, and the capacity to grow components independently. This design encourages better fault isolation, shorter development cycles, and seamless integration of various technologies. Encapsulation of these individual applications and their dependencies ensure consistency across various stages of development, testing, and deployment.

Design Architecture

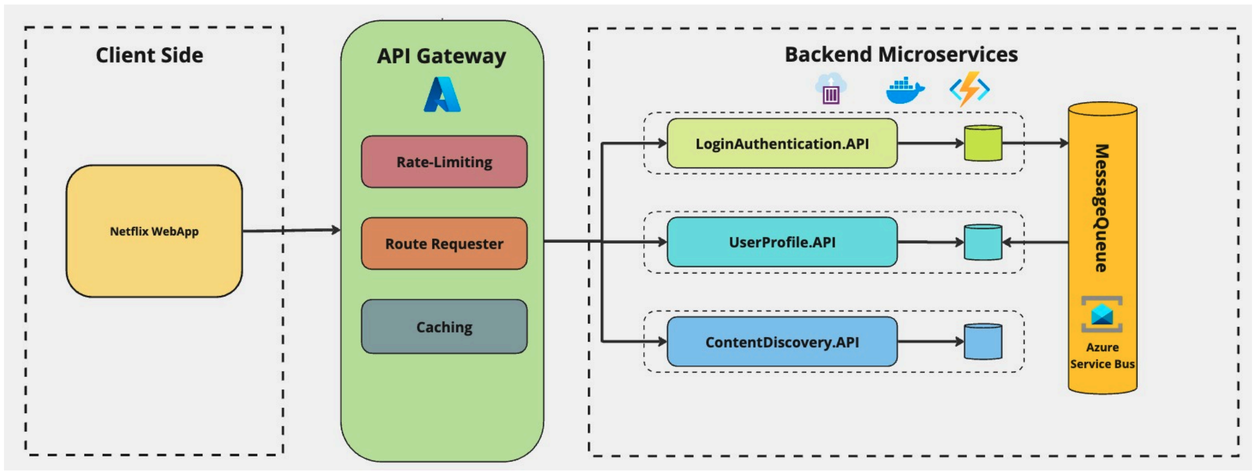
Considering the nature of our web application, we opted to use a microservices architecture to prioritise flexibility and scalability, identifying four distinct services to separate our web application.

1. Content Discovery (Backend API) - performs the functionality of querying a netflix database to return shows: http://netflix-content-server.c5chg3ggc2a2fddj.westeurope.azurecontainer.io:5001/
2. Login/Registration (Backend API) - registers, logs in and out users: https://finalprojectsse.azurewebsites.net/api
3. User Profile (Backend API) - allows users to favourite and unfavourite shows from a profile: http://user-profile-microservice-api.e4fvgqf6hucwg8ca.westeurope.azurecontainer.io:5000/
4. User Interface (Frontend): http://netflix-front-end-container.b4ehgthxb6evd0ga.westeurope.azurecontainer.io:8000/

The microservices are decoupled, each establishing links to their own database. Both the user profile and the content discovery APIs have been containerised using Docker and deployed on Azure Container Instances, while the login authentication API was developed using Azure Function Apps. None of these backend services hold any critical dependencies to each other, which is crucial to ensuring flexibility between developers working on different functions. The design decision with a database per service is crucial, since we wanted to avoid cases of teams writing unwanted changes to the same database. We implemented a mechanism to ensure separate databases, while also maintaining data consistency. In our architecture, this was engineered between the login authentication API and the user profile API, since it is often the case that certain login details about the user might be needed to help construct the user profile page. On the other hand, the content discovery database is its own database, and which is relatively safe compared to the other two, since the functionality of the API is only defined for read-operations.

To establish any links between backend microservices (which would inevitably be needed if future functionality was introduced), we used a message queuing system in Azure Service Bus. Whereas a system holding important dependencies between services might implement synchronous communication using endpoints, the introduction of the message queueing system is a form of asynchronous communication which is particularly useful for limiting the updates that one microservice has for another, and such that the microservice on the receiving end can integrate without establishing a direct link. In our case, we implemented a publish-subscribe model between the login authentication API and the user profile API. Every time a new user is registered, it will send an update to the queue with the data of the new user which the subscriber will listen out for, allowing it to pull and add the data to its own database to create a user profile. This model is beneficial because it reinforces the decoupled databases mentioned earlier while storing the same information and without affecting or writing unwanted changes.

Figure 1. Our Application Architecture



Having satisfied the backend architecture, we specify how the client-side application relates to the backend microservices. Also containerised using docker and hosted in production using Azure Container Instances, the Netflix web application sends requests and processes responses through an API gateway that serves as an extra layer of decoupling between the backend APIs and the frontend. It also introduces additional benefits such as rate-limiting, caching, and potential security measures that can be implemented if necessary, which makes the system particularly scalable if there was a desire to extend the web application beyond just a prototype. With or without an API gateway, the way that the frontend container communicates with the backend is still through the use of endpoints.

Client-side

We used flask for the client-side's back-end to handle routing, session management, and server-side logic. It interacts with the three APIs we deployed on Azure: User Login API, User Profile API, Content Discovery API - and also the public API Leaflet.js for rendering the map.

index.html (Homepage):

- Serves as the homepage and has different navigation bars based on the user's authentication status.
- Displays two widgets via iframes, and the interactive map also via an iframe.

Login / Logout / Register:

- Comprehensive user registration / login service was implemented via the User Login API.

- Intuitive redirection between the different steps of the user registration process.
- Favouriting feature only for logged-in users.

map.html (Interactive World Map):
- Utilises Leaflet.js to display the interactive map and defines geographical boundaries for various countries as interactive polygons. Upon clicking a country - unique netflix show information is displayed.
- Implements caching to minimise redundant API calls:
  - Before fetching country-specific show data, it checks if the data for that country is already cached in the local storage under the key 'cachedData'.
  - If cached data is available, it retrieves and utilises it, thus avoiding additional API requests.
  - After fetching new data from the Content Discovery API, the data is cached.
- /map redirects to /country inside of the iframe so that the user is still on the homepage throughout these interactions.

country.html (Shows for Selected Country):
- Selected country's data is fetched from local storage - localStorage.getItem('countryShows') and localStorage.getItem('country').
- Provides functionality to add/remove shows to favourites (javascript functions):
  - Utilises User Profile API (endpoints /checkfavourite with GET method, and /addfavourite, /deletefavourite with POST methods) to manage favourites.
- Implements session storage to trigger widget reload:
  - Sets a sessionStorage item 'reloadWidgets' upon adding or removing a show from favourites to signal a reload of the widgets on the page.

widget1.html (Top 5 Favourited Movies) (embedded on index.html):
- Fetches data from User Profile API (endpoint /topfavourite with GET method) to display the top 5 favourite movies in table format.

widget2.html (Your Saved Shows) (embedded on index.html and profile.html):

- Lists the user's favourited shows, fetched from the User Profile API.
- Allows users to unfavourite shows directly from the widget.
- Sets sessionStorage items 'reloadCountry'and 'reloadWidgets' upon removing a show from favourites to signal corresponding reloads of pages (otherwise could display "favourite" on widget but "unfavourite" on country, or incorrect favourite count on widget 1).

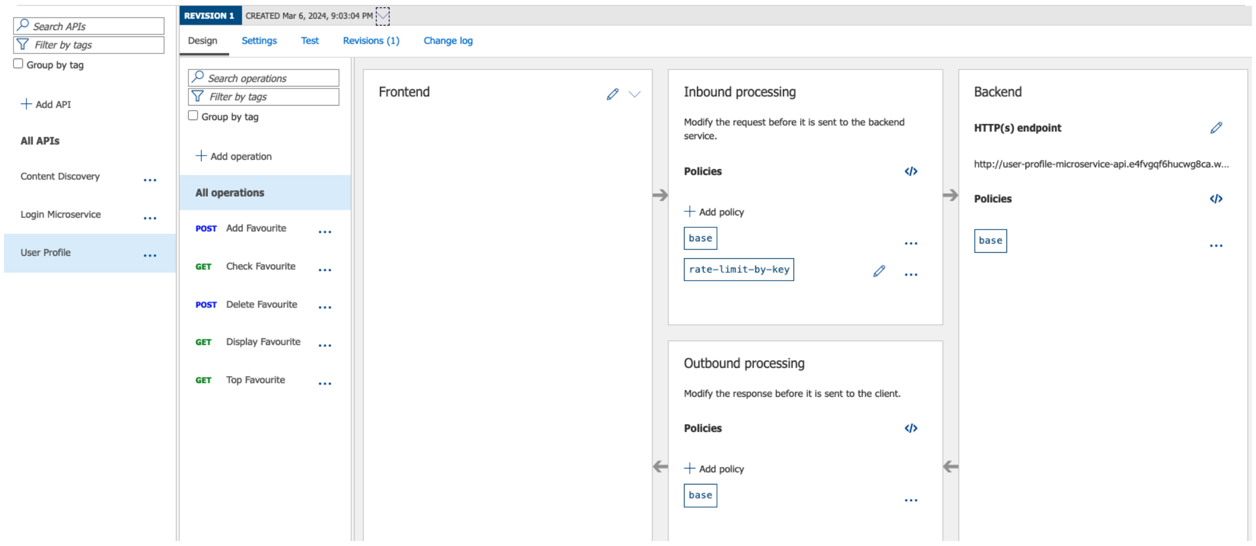
Search (navbar form and searchresult.html):

- Communicates with Content Discovery API, users can search specific show strings or country strings.

API Gateway

The API Gateway was established as a means of promoting greater scope for developer flexibility and scalability. To integrate the gateway, we used the Azure API Management service to generate a unique Gateway URL (https://sse-project-2.azure-api.net), used to replace all of the HTTP base URLs that would previously represent the client-side call to the backend APIs. Unifying all of these URLs under a single gateway URL is beneficial, as it allows for greater simplicity in the development process. Consequently, the set up of the calls to the APIs was completed in the APIM service, by specifying three different APIs, and the accompanying operations within those APIs, including its method, endpoint, query parameters, and any templates (depending on the method). This is shown in the screenshot below.

Figure 2. API Management Tool in Azure



The benefits of using the APIM service are multitudinal, specifically in granting the developer the ability to easily specify extra functionality that may be required by using policies. For example, we adopt rate limiting for each microservice, which can help to manage requests sent to these APIs in high traffic. While our web application is still small, this demonstrates a viable path for rising up to scalability issues when presented to a larger audience. The service is also able to handle CORS policies, as well as caching, which we use in displaying the content of countries, thereby bypassing the need to re-query the database after a country has already been queried. Although not implemented in our example, an API gateway can provide enhanced security features which would certainly be useful in the context of further scaling.

Aside from even just functionality of the APIM service, the concept of the API gateway can be useful for extended platforms later on. For example, while we do not incorporate load balancing features (due to cost concerns and also not needed for this specific application), the API gateway is able to handle this sufficiently for now. Furthermore, if more client-side applications were to be established, the API gateway helps to make connections with the backend less messy.

Backend Microservices

API	**Content Discovery**	**Login/Registration**	**User Profile**
Endpoints	/shows?country={countryCode}	/register?username={username}&password={password}	/displayfavourite?username={username}
	/search?query={queryString}	/login?username={username}&password={password}	/checkfavourite?username={username}&show={show}

	-	/logout?token={token}	/deletefavourite (**POST** method with body: username, show)
	-	/protected?token={token}	/addfavourite (**POST** method with body: username, show, country)

Content Discovery

The purpose of this microservice is to return results about the shows from a Netflix database. Using Supabase as our database platform, we imported a publicly available dataset with information about all of the shows for each country where Netflix is hosted. We then established a Postgres connection using environment variables to enable easy querying functionality. In our first function, we use an endpoint '/shows' requiring a query parameter for a country's ISO code (i.e. DE for Germany), which triggers the SQL query to identify the unique shows to that country (i.e. shows that cannot be found in any other country). An important point to note is that the dataset provided online is likely not fully accurate, and was used because the Netflix API is no longer available to developers. After the query, the show titles for that country are returned in the form of a JSON string. Unit tests were written for this microservice, specifically focusing on mocking the database connection. These tests helped to verify that the return value of a valid country query is correct, that a country with no unique shows will return an appropriate message, and that an invalid country code will return an invalid country message.

The content discovery microservice also introduces a search function using the search bar in the frontend, with an endpoint '/search' requiring a query parameter for a string. This loads into an SQL query that is able to return shows for the query regardless of whether it is an actual show title, a country, or a country code. This was also followed by a suite of unit tests, specifically focusing on testing a valid query, an empty query and special characters. Final tests that were conducted included checking for database connection errors and handling a potential SQL injection attack.

User Registration/Login

For our app, we decided to include user profiles. Therefore it was necessary to have a user authentication system. The registration/login microservice was created as a HTTP Trigger in Azure Functions App allowing for user registration, login, protected resource access, and logout features. Its endpoints are '/register', '/login', '/protected', and '/logout'.

The microservice stores information on the username, password, account creation date and session token in a Supabase database. Before being inserted into the database, if new users are registered successfully, their passwords are encrypted using 'bycrypt'. A successful user registration sends a message to our Azure Service Bus queue containing information on eventType ('UserRegistered') and the new user's username. This allows our User Profile microservice access to usernames to create user profiles. The login feature verifies the login details against the Supabase database using 'bycrypt'. If successful, a Json Web Token is created and added to the database for that specific user's session token. A valid session token is required to access the '/protected' endpoint, which is called in our front end to restrict pages that require a user to be logged in. The logout feature again checks for a valid JWT token and deletes it from the database on a successful logout. The microservice handles errors, password encryption/decryption and JWT token expiration.

As a HTTP Trigger, this registration/logic microservice only runs when requests are made to it, reducing upkeep costs. Additionally, as a serverless model, Azure Functions automatically allocates/deallocates resources to the function app. This allows for scalability without hassle if the app were to need to handle a greater amount of traffic. If there were millions of requests per day, the microservice could use an upgraded function app plan or be moved to a container instance to lower costs. Because this microservice is not an app, no app tests were written. However, tests were done on Azure as well as by using 'httpie' http requests in a terminal. Deployments and updates were made through VSCode's Azure extension easily through the click of a button.

User Profile

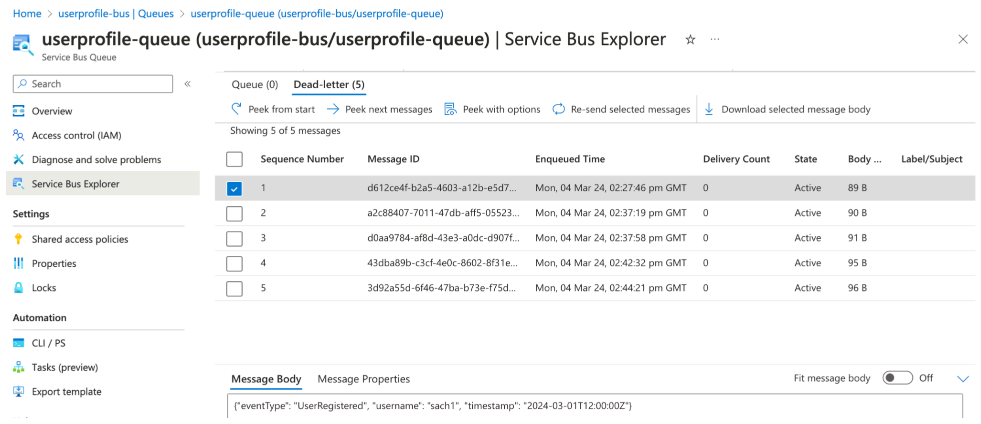
An essential part of our web application was the user profile page which integrates user information and favourite movie selections. The user profile page serves as a hub, offering users a comprehensive overview of their account. This service allows users to mark their preferred content by clicking a 'Favourite' button on a movies page, authorising our system to capture the selected movie's name and store it in a Supabase database, associating it with the respective user. When a new user registers on our platform, a corresponding event is generated and added to the Azure Service Bus. This event provides information concerning the user, such as the event type and username. Our user profile microservice actively listens to the addition of these events and updates the database, ensuring a real time synchronisation. This creates a centralised and organised repository of user data for easy access and retrieval. The Supabase database serves as the foundation for our user profile page, allowing it to display accurate user information.

The implementation code is a Flask application that serves as a backend API for managing users' favourite selections using Supabase. Flask-RESTful is used to establish three functions: 'AddFavourite', 'DeleteFavourite' and 'Favourite', each performing a distinct operation. The 'AddFavourite' resource handles a POST request to add a user's favourite movie, first verifying for existing favourites. The Supabase Python client is used to connect to the database, get user IDs, and manage items in the 'Favourites' table. The programme starts with environment variables, and the API is set up with three endpoints. Overall, it offers a simple interface for users to administer.

Azure Service Bus

As previously mentioned, our service bus allows data communication between our login/register microservice and our user profile microservice using a message queue and event listener system. When active messages are read by the user profile microservice, data is handled appropriately and the message is turned into a dead-letter-message to prevent multiple listens/reads.

Figure 3. Azure Service Bus



Home > userprofile-bus | Queues > userprofile-queue (userprofile-bus/userprofile-queue)

userprofile-queue (userprofile-bus/userprofile-queue) | Service Bus Explorer

Service Bus Queue

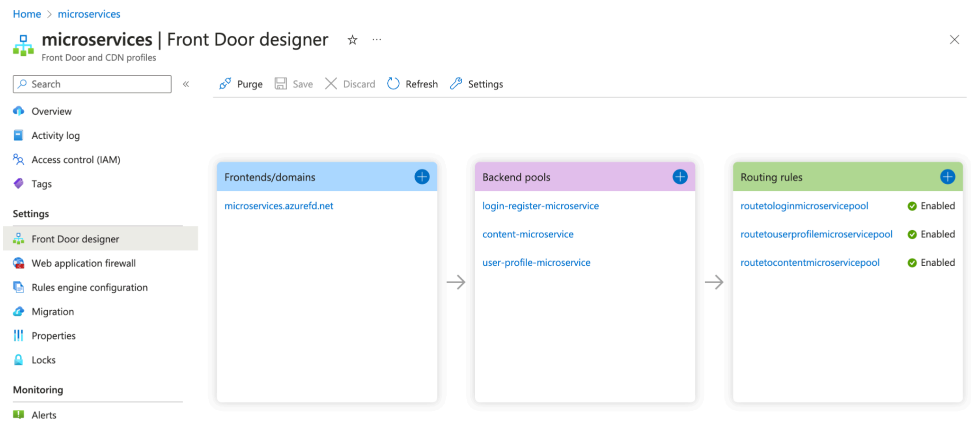
Showing 5 of 5 messages

	Sequence Number	Message ID	Enqueued Time	Delivery Count	State	Body ...	Label/Subject
✓	1	d612ce4f-b2a5-4603-a12b-e5d7...	Mon, 04 Mar 24, 02:27:46 pm GMT	0	Active	89 B	
	2	a2c88407-7011-47db-aff5-05523...	Mon, 04 Mar 24, 02:37:19 pm GMT	0	Active	90 B	
	3	d0aa9784-af8d-43e3-a0dc-d907f...	Mon, 04 Mar 24, 02:37:58 pm GMT	0	Active	91 B	
	4	43dba89b-c3cf-4e0c-8602-8f31e...	Mon, 04 Mar 24, 02:42:32 pm GMT	0	Active	95 B	
	5	3d92a55d-6f46-47ba-b73e-f75d...	Mon, 04 Mar 24, 02:44:21 pm GMT	0	Active	96 B	

```
{"eventType": "UserRegistered", "username": "sach1", "timestamp": "2024-03-01T12:00:00Z"}
```

Load Balancing

We implemented the architecture for load balancing using Azure Front Door to protect our app against failures due to increased usage. However, as we only have one instance of our front end and one instance of each microservice, it currently does not balance between any instances. We decided to not fully commit to Azure Front Door due to costs, our current app usage and critically, front door does not automatically horizontally scale instances of our app. Below are screenshots of two potential load balancers, one balancing the microservice instances, and the second balancing the front end container instances.



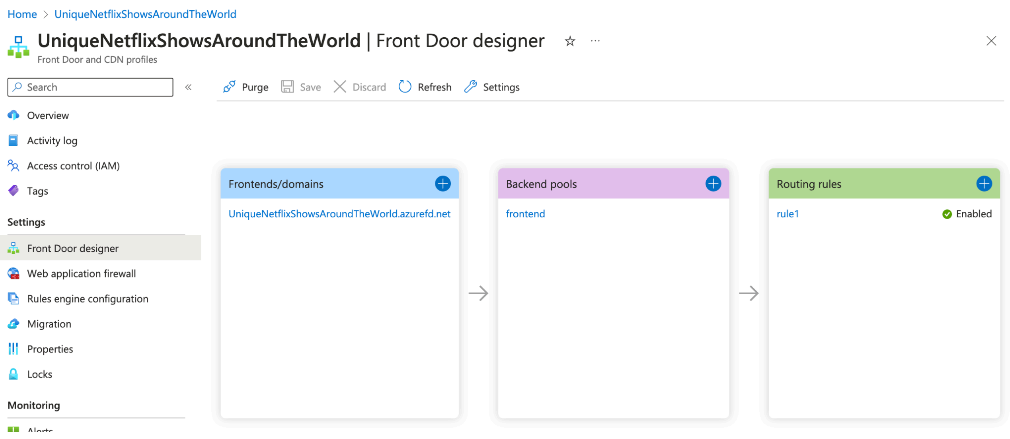
Home > microservices

microservices | Front Door designer

Front Door and CDN profiles

Frontends/domains	Backend pools	Routing rules	
microservices.azurefd.net	login-register-microservice	routetologinmicroservicepool	Enabled
	content-microservice	routetouserprofilemicroservicepool	Enabled
	user-profile-microservice	routetocontentmicroservicepool	Enabled

Front Door Load Balancer: http://microservices.azurefd.net/topfavourite
User-Profile-api:
http://user-profile-microservice-api.e4fvgqf6hucwg8ca.westeurope.azurecontainer.io:5000/topfavourite
Api Gateway: http://sse-project-2.azure-api.net/profile/topfavourite



Frontends/domains	Backend pools	Routing rules
UniqueNetflixShowsAroundTheWorld.azurefd.net	frontend	rule1 Enabled

Load Balancer: http://uniquenetflixshowsaroundtheworld.azurefd.net/
Front End (1 instance):
http://netflix-front-end-container.b4ehgthxb6evd0ga.westeurope.azurecontainer.io:8000/

The Azure Front Door architecture is partially set up for load balancing. To go down this path we would need to horizontally scale our microservice's containers, and our frontend container. Furthermore, we would need to correctly implement the endpoints for all microservices adding the microservices load balancer url (http://microservices.azurefd.net/) to our api gateway, adding health checks, priorities and weights within each microservice pool. In the end, I believe a different service that automatically horizontally scales our containers should be used for ease, perhaps Azure Kubernetes Service (AKS).

Evaluation

Advantages of Our Design

Our architecture for our web app provides significant benefits as a system. These include the increased reliability of our microservices as they're decoupled, and the ability to horizontally scale containerised services individually using the Azure CLI interface. The decoupling of microservices has allowed flexibility in development and updates without disrupting the whole system. It has also allowed us to pinpoint errors more easily within cleaner codebases that are linked when necessary through an event queue. This additionally ensures data consistency across databases. The HTTP trigger (login/register microservice) is automatically scaled by Azure based on request demands. These factors facilitate efficient resource allocation and better performance.

Our API Gateway provides an additional layer of abstraction between the frontend and backend services. It allows for better management of requests, rate-limiting, caching, and potential security measures were we to enter production using https, allowing for an improved performance in the overall system.

In terms of cost efficiency, the benefits include ease of maintenance, scalability, and adaptability. The ability to scale specific components independently can lead to a more cost-effective resource allocation. It is possible to switch certain services between Azure HTTP triggers, container instances and web apps to optimise the costs as demand grows. As demand grows, it could be important to move our azure container instances into azure web apps for better scaling, particularly if we were to use load balancers.

Disadvantages of Our Design

Despite the benefits obtained through using a microservices architecture, plenty of drawbacks still exist that can affect the performance of a system. First of all, the microservices architecture can be overly complex depending on the type of application. In the case of our web application, it could be argued that three microservices is small enough to warrant the use of a monolithic architecture instead, which might simplify the development process. To justify our choice of microservices, we argue that this is done particularly with future scalability in mind, such that it becomes easier to continue developing if new products and services want to be added, thus removing the need to re-build any infrastructure that could be the case with a monolithic application. The complexity resulting from this type of architecture can lead to more difficult testing of applications, specifically given its distributed nature.

Potential performance bottlenecks might also arise as a result of a microservices architecture that is not well-designed. Firstly, since we have adopted a database-per-microservice framework, there might be issues in data consistency. This is an especially pertinent issue between two of our microservices, the login authentication API and the userprofile API. Currently, since they communicate over an event queuing mechanism in Azure Service Bus, it may be the case that there are delays in data transfer and updating, even despite the benefits of decoupling and using asynchronous communication methods. This is therefore a potential performance bottleneck. In a similar vein, it might also be seen that the independently containerised microservices can introduce network latency. Since microservices have to spend time communicating over the network, this might be slower than in a more tightly coupled architecture.

Thirdly, while the API gateway can be a useful tool to ensure reliability and consistency between the client-side and the backend, it can also be the case that API gateways become a bottleneck in themselves. Since all requests must go through the API gateway, the entire system could be affected if the gateway itself does not perform well. Furthermore, while it was fairly simple to maintain in our application given the small number of microservices, it can become a lot more tricky to configure when the number of microservices increases. It can also be particularly difficult if more clients are introduced and who may require different versions of the APIs. The careful design of the API gateway is therefore crucial.

Finally, the most significant drawback is the cost of developing a microservices architecture. Given the need to decouple the components in this design philosophy, it becomes all the more expensive to configure them separately and to ensure that they have the tools necessary to communicate in a constrained and effective way. This is clear from the costs of introducing aspects such as event queues and API gateways, both of which might be avoided in a monolithic architecture. Despite that concern, it might be countered that the long term cost of our architecture would actually be less than that of a monolithic tightly coupled architecture if more services were to be introduced in the future, following the “Ball of Mud” principle. In effect, the complex entanglement of the monolithic architecture might lead to extra time and resources spent on debugging, and dis-entangling software, introducing higher technical debt than that of a microservices architecture which starts with a higher upfront cost.

Concluding Remarks

In this report, we have presented a prototype of our Netflix Content Recommendation application. Following important principles of a microservices architecture, we have implemented design choices such that our web application can be scaled reliably and flexibly if extended to a larger audience. Specifically, our application was partitioned into four elements: a frontend, a content discovery API, a login authentication API, and a user profile API. With the objective in mind of ensuring single responsibility and autonomous decisions, we focused on ensuring maximum decoupling where possible of our services, such that our software system is more robust to potential failures of parts of the application, and more flexible when developing, re-deploying, and versioning the software. To facilitate these principles, we containerised each service, and introduced asynchronous communication between backend services, decentralised data management with each service owning its own database, and an API gateway to manage and route requests from the frontend to the backend. It is hoped that such principles can be applied at a larger level when introducing more microservices or more clients.

While we support the adoption of these principles, there are also areas where such an architecture experiences drawbacks. The complexity of a microservices design can have performance bottlenecks if not carefully considered, and the cost of developing such a system can be more expensive. We recognise that given the current size of our system with a small number of services, it may be the case that a monolithic architecture makes more sense due to its simplicity of development. However, we have designed this architecture particularly focusing on future ease of development over its current status. Despite this, there are clearly areas for improvement that we consider below.

For further improvements to our system, we would enhance the load balancing capabilities of our web application. Currently, it is the case that there are four distinct services each mapping to its own container (or function app in the case of the login authentication API). Owing to the cost of development, we did not have multiple instances of the same service, which would be preferable when wanting to extract the maximum benefits that a load balancer can offer. By implementing such a strategy, our system would be more robust to certain services going offline for development, and allow for greater geographical distribution of services if needed. Along with the distributive nature of load balancing, our system could also introduce closer monitoring and health checks of services to fully understand the demand for microservices and to ensure that traffic is more consistent and fair, as well as introduce fault tolerance mechanisms such as circuit breakers. Further improvements to our system would also include better security measures, which we have currently not considered but which would be important if deployed to the public, and to ensure greater data consistency measures if failures in our event queueing mechanism were to be experienced. Finally, we would also want to further explore potential edge cases in our testing, in particular with regards to integration and system tests. If these were to be implemented, the foundations and infrastructure of our system would be more robust for scaling.

GitHub Repositories

* Zeeshan's Front-end GitHub:
  https://github.com/za-zeeshan-33/sseproject2
* Raymond's ContentDiscovery API GitHub:
  https://github.com/RaymondGuo2/ContentDiscovery
* Sach's User-Login API GitHub:
  https://github.com/sn2023imperial/usermicroservice/
* Jenarth's User-Profile API GitHub:
  https://github.com/JenarthP06/SSE-Project-II

GHCR Links

https://github.com/users/RaymondGuo2/packages/container/package/netflix-content
https://github.com/users/RaymondGuo2/packages/container/package/netflixfrontend
https://github.com/users/sn2023imperial/packages/container/package/userprofile