Architecture choices -

Microservices

The services are separated from one another so that they can be deployed as microservices, allowing a failure of one microservice to not affect the entire system, which also provides good scalability. This choice also reduces the overall build complexity of each individual service but the complexity of the overall system is increased. Each microservice is uploaded on its own virtual machine, this allows us to configure the machines based on the load each service is receiving and be more flexible. It is slightly more expensive than hosting on 1 virtual machine, but the ability to effectively scale horizontally and vertically the different machines makes up for it. Fault tolerance is also increased since each microservice is deployed and running separately. It does make the complexity of integrating for the other team slightly harder as they need to access different IP's to send their requests to. Our services are independent of each other, with only minimal communication needed and that is why we choose this architecture.

GraphQL

As part of the solution, we have implemented the products path through GraphQL, which was not free of choice as it was a mandatory requirement, but we can summarize several pros and cons after implementing it. GraphQL with its flexibility allows developers to get exactly the data that is needed, which improves the speed because with GraphQL you don't underfetch or overfetch the data. This mechanism also promotes loose coupling, since once the data is available, then we can modify the queries to our needs. One of the cons of using GraphQL was the complexity, we had to spend time understanding how it works, as we did not use it in our previous projects. Another disadvantage is error reporting and monitoring as there needs to be extra actions involved to get the error response from the query(HTTP response is always 200). Since the schema is public the developer can use tools that can help him get the exact data he wants and configure queries correctly(tools like GraphiQL, Apollo Explorer).

Rest Api

The majority of the services were created as Rest APIi's as they allow easy integration, easy to build and construct. Drawback is that they are not really that asynchronous so some events or actions can become blocked. Sometimes data between services needs to be shared. This need was met by implementing a service bus by our team. Another disadvantage of Rest Api is fetching little or more than the required data which could lead to over-fetching or under-fetching. For the other team, maintaining the state of their frontend information may create a situation where they over-send requests, since REST doesn't send updates itself, thus leading to server load.

Database choices

For the project we use several databases, which were chosen for a specific reason which best suits the problem of each individual service. The con of using multiple databases is that some information even small may need to be shared between databases. So a need to keep data integrity between them has to be implemented. Another is hosting multiple databases could increase overall costs, but will provide good scalability and availability of data.

Neo4j for email and websocket (friends list)

The database of choice is Neo4j,which is a graph database that represents relationships of nodes, making it easy and quick to search for users, their friends, and whom they have invited. Since Neo4j is a new evolving database, a really good OGM is not yet available for Javascript, and those that exist have some limitations, so utility functions were added to allow good functionality. Since these services(email and websocket) are dependent on each other to some extent, which involves the handling of adding a friend by another user, for this reason we assigned the database to both services. With email sending the invitation links/adding them, and websocket notifying the frontend of the acceptance and when friends login.

MongoDB for storing user credentials and updating user information

Due to the data being saved as a document, MongoDB makes a good choice for nodejs applications because of the ease of integration with the mongoose package for nodejs. Furthermore, due to its document-oriented nature MongoDB offers higher speed and performance. MongoDB has a free tier which is great and allows for easy hosting. It is important to note that MongoDB does not support collection joining. While this does not mean it is impossible, it would be time-consuming and would affect the performance, and in our case this wasn't a concern since services use different collections to save data.

Sqlite for Products

Sqlite software library was used for saving the product details according to the schema provided as part of the requirements. Working with Sqlite was an interesting change from working with other RDBMS. The pros that we could feel while working with the Sqlite are the lightweight format as it is simple and easy to start using, Sqlite is self-contained and serverless without the need for the operating system or external libraries to run which helps to boost its performance efficiency. Despite SQLite's limited size, for now we do not need to worry about space on disk, but there might be a problem in the future.

Redis for Products

Redis was used as a cache layer for storing the products and their information. Since Redis is an in-memory database, we opted for it (instead of working with the products.db file directly) due to its high performance and scalability. We were considering that if there would be a need of scaling up the application, Redis would be a good option since it also offers zero downtime or performance impact when scaling. Setting the Redis database was straightforward as we did it in RedisLabs. Integrating with Redis wasn't such a big of a hustle as we’ve used the official “redis-om” package for NodeJS which provided functionality to create Classes (Product, Product_Image etc.) through which we would interact with the Redis itself and use the RedisSearch feature that was required for this part. And lastly, Redis is open source, which is a benefit considering the cost and community support available. As Redis stores data in-memory, therefore all the data must fit in RAM . In comparison RDBMS usually stores the data on disk and cache part of the data in memory. That means that with a RDBMS, you can manage more data than you have memory. With Redis, you cannot. Additionally, by default Redis doesn’t persist the data to storage, however RedisLabs has replication enabled which is definitely an advantage.

Service bus

We chose to create a service bus thorough topics to facilitate internal communication between our services. This is especially needed as the different microservices and their respective databases need to know of the existence of new users that sign up. This does complicate the overall structure of our system but allows us to achieve a level of asynchronous communication. Since we are pushing events to the service bus, even if the service bus fails it won’t really hinder all of the other services like a POST call would. With topics you can implement retention policies which can save events and be replayed by the subscribers if needed, in case of failure of the event. Costs do slightly increase as we needed to host this in Azure Service Bus. This messaging service can also become a single point of failure, but setting up multiple clusters does solve this issue.

Websocket

Websockets allow 2 way communication between client and server. It allows easy state management by the integrating team as the server itself sends updates of new information/events and they don’t need to actively query it. Server load is mostly dependent on the server. Socket.io for NodeJS is a good library that is widely used and simplifies the communication between server and client. Configuring to listen to the websocket does take more effort from the integrating team, as who exactly receives these updates has to be controlled in some way.

Serverless functions through Azure functions

Serverless functions are a fast, convenient, and cost-effective way of developing cloud-native applications. Within minutes, it’s possible to deploy functions to respond to events. The reason we pick subscribe and rss services is that they are not an integral part of our architecture and we can save a decent amount of money by creating these as serverless functions. They may not be used often and thus money can be saved by implementing them through functions. Scaling is also very efficient with them, as the functions do scaling and load balancing by themselves without needing extra configuration.

Subscribe service

We added a webhook for the other group to connect to and put their url for their application within.This allows us not to hardcode their connection points, and if there's a need to change(rehosting) we don't need to change our code, but the onus is on them to update this information. Initially we thought of this server as the one to update most of the information for the friends list(updating about sent friend requests and who is offline and online), but we reconsidered and decided that websocket would be easier to develop and for the other team to integrate with too. We felt that doing it as a webhook would create a server with a lot of load on it because most of the data transfer would be handled by it.

RSS service

We use the RSS so the integrating team can send the 'wished for' items to it. This feature saves a lot of time as it is quick to get updates and send them to clients. The information that is sent is formatted based on time so the client receives the latest "news". The cons are that it is an old technology and is heavily dependent on files being properly maintained.

Azure Blob Storage

We decided to use Azure Blob Storage as we needed a place to store the necessary files – products.db file, rss file, etc. We opted for this option instead of saving the files locally because by using the public cloud storage, any service that needs a certain file just needs to acquire the connection string and will be able to get it. Moreover with cloud storage, we have "a single source of truth" for those files, and we won't end up in a situation where two services use files with the same file name but have different or outdated content. Using a public cloud storage adds extra costs, however these are very minimal since storage is quite cheap nowadays. Moveover we used it to also store images because if we would store it in a database, the data would get too big and it would be hard to scale or replicate the database.

Documentation

We use swagger files to generate documentation as having 1 swagger file and then quickly editing it is super easy for developers. Azure also has an api management service where we can host the documentation for all of our servers from an OpenAPI 2 specification file (swagger file). This is great for the integrating team as they don't have to navigate to our separately split servers and try to access documentation there (even though we also support documentation for each service on its virtual machine). Hosting this service is very cheap as well. A cons is that the GraphQL file doesn't work with Swagger so that documentation had to be hosted elsewhere.