UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science

Faculty of Engineering, Built Environment & IT

University of Pretoria

COS301 - Software Engineering

CAPSTONE PROJECT

Architectural Requirements

1. Architectural Design Strategy

For our architectural design strategy we decided to go for design based on quality requirements. This was the best approach for us because:

- We had our quality requirements in place for demo 1. So it was a matter of using the quality requirements as reference.
- This also ensures that the system is built on quality and is meant to last.

2. Architectural Quality Requirements

2.1. Scalability:

The system should allow for companies of varying sizes to use the platform without any performance.

The Micro-services architecture allows us to address scalability, as we are able to replicate services that are in high demand.

2.2. Security:

Security is an important issue on our platform. We hold sensitive user data such as passwords and emails. Thus our system should be robust enough to deal with any security breaches.

The client-server approach allows us to separate security concerns between the client and the server.

- On the client side, we can use things like form validation to ensure security.
- On the server side we use tokens to ensure the valid user is on the system.
- On the storage side, we are using password salts and hashing to secure the stored data.

2.3. Reliability:

The system should be fault tolerant and be able to deal with loss of connection to the server. The system should communicate any downtimes to the server.

Currently the most critical point we've identified is the database. If it fails, there's no data or flow of data.

For the database, we have implemented 2 databases; One live and the other for redundancy.

In terms of the reliability of the system, a microservices approach allows services to function independently and have independent data stores. Hence that service will be active and functioning as usual.

2.4. Usability:

Usability in these applications is very important. Many admins will be looking at the application for long hours, hence the importance of something that looks and feels natural.

Angular provides angular materials. This library provides components like cards which are aesthetically pleasing and very ergonomic in terms of user finger interaction.

2.5. Maintainability:

For an application to have a very long shelf life, it must be easy to debug and allow addition of new features and updates.

The n-tier architectural pattern we've used divides our application into inter-dependable layers, each with their own functionality and unique contribution to the system.

This approach makes our application very modular and allows for fixing and adding features, suppose a new Database relation is less disruptive to the overall functionality of the system.

2.6. Performance:

Performance in any application is critical. With us, organisations of different sizes will be using our application, thus the need for consistent performance across all users.

Client-server archives this quality requirement because, most of the processing is removed from the client's terminal and into the server space. The client need only a browser to view and work with the application and does not need to have any pre-installed hardware or software.

Microservices improves performance through optimisation of services. A service is dedicated to a particular function and need not worry about other function calls. The service can be fully optimised to execute its tasks.

3. Architectural Styles

Our application makes use of two main architectural styles.

3.1. Structured/ layered architectural style:

This is observed through our n-tier pattern. The main reason is the power of separation of concerns. Different layers have different tasks. This makes maintaining and developing the system easier than having everything as a Monolith.

3.2. Remote Procedure call architecture:

This is demonstrated by our Microservices approach. The client is able to call various services remotely from their system.

4. Architectural Patterns

4.1. Quality Requirements and Patterns

4.1.1. Client Server

- Maintainability:
    - Server and client concerns can be dealt separately.

4.1.2. MVC

- Usability:
    - The MVC architecture is the best way to build
- Security:
    - Through form validation, MVC provides security at the front end.

4.1.3. N-tier

- Maintainability:
    - The separation of concerns makes the code base easier to debug and implement fixes, changes in the logic of the current layer will not affect other layers.
    - Adding features becomes easier, as the design is modular.
- Security:
    - With the implementation of a non-zero trust system, we can implement security at various interfaces in the pipeline.

4.1.4. Micro-Services

- Scalability:
    - With an increase in demand, the system is able to expand itself to deal with the incoming traffic.
- Availability:
    - If one service goes down, the overall system will not be affected. Other services will continue to run

5. Architectural Constraints

- The use of angular: Our client insisted on the use of Angular, thus we had to use a pattern that incorporates angular, such as MVC.
- The use of microservices: Our client insisted on the use of microservices, because the system should be scalable to accommodate clients of different sizes.
- Micro-services also allow us to address reliability which is a key requirement of our system.

6. Technology

6.1. Front End

6.1.1. Angular

Angular is a TypeScript-based free and open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations.
In this particular case, we are considering angular ionic, which uses typescript rather than javascript as a language.

Pros:

- Maintained by google. This means that the platform is constantly getting updated with new features and performance increases.
- Very easy to work with. Angular has a comprehensive development documentation that makes it easy to learn and pick-up.
- Angular is free.

Cons:

- New versions of angular are being churned out by the dozen. Without a proper dependency manager in place, this can be a pain for developers to upgrade their system to the newer updated version

  As a group, we experienced this first hand. Google released angular 14, while our project was running on angular 13. This completely broke our project when angular 14 was used to generate a component.

Angular is built on MVC-Model view controller architectural design pattern. With the model being classes, view being the html files and the controllers being all typescript files.

6.1.2. Flutter

Flutter is an open-source UI software development kit created by Google. It is used to develop cross platform applications for Android, iOS, Linux, macOS, Windows, Google Fuchsia, and the web from a single codebase.

Pros:

- Allows you to write platform independent code.
- Built on C++, meaning fast execution
- Open Source

Cons:

- Flutter is built on Dart. This language might be a steep learning curve for most of us, as it's very unfamiliar. We must be considerate of the time constraints.

Flutter is built on C++, a very low level programming language. The client-server architecture boosts the performance of a platform because most of the heavy lifting is done on the server side.

Flutter being built on C++ might increase those gains even further, considering that our system depends on the client server design pattern.

6.1.3. React Native

React Native is an open-source UI software framework created by Meta Platforms, Inc. It is used to develop applications for Android, Android TV, iOS, macOS, tvOS, Web, Windows and UWP by enabling developers to use the React framework along with native platform capabilities..

Pros

- Open Source
- Cross platform

- Ready stable solutions are available through the many Libraries that react native supports.

Cons:

- React is hard to debug. You need external services like flipper to properly be able to run the application

Conclusion

We decided to go with Angular. It's what the client wanted us to use in the first place. Angular has a well documented documentation and typescript is very easy to pick up as it's relation to javascript.

MVC is another reason we chose Angular. This fits well with our architectural pattern.

6.2. Back End

6.2.1. Nestjs

Nest (NestJS) is a framework for building efficient, scalable Node.js server-side applications. It uses progressive JavaScript, is built with and fully supports Typescript (yet still enables developers to code in pure JavaScript) and combines elements of OOP (Object Oriented Programming), FP (Functional Programming), and FRP (Functional Reactive Programming)

Pros:

- Allows the creation of scalable applications.
- Allows the creation of loosely coupled architectures. This helps with Maintenance.

Cons:

- If you are building microservice architecture, the abstraction is too generic and doesn't allow for a more granular approach to building microservice architectures.
- Lacks in features compared to other frameworks .NET or spring(for Java).

6.2.2. Nodejs

Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.

Pros:

- Asynchronous event driven IO helps concurrent request handling.
- Node is built on javascript, which is easy to pick-up and learn

Cons:

- Node.js doesn't provide scalability. One CPU is not going to be enough; the platform provides no ability to scale out to take advantage of the multiple cores commonly present in today's server-class hardware
- Dealing with relational databases is a pain if you are using Node.

6.2.3. Nextjs

Next.js is an open-source web development framework built on top of Node.js enabling React-based web applications functionalities such as server-side rendering and generating static websites

Pros:

- Very well suited to react. In fact, it's one of the most highly recommended react tool-chains.
- Allows the website to be rendered on the server-side before being sent to the client.

Cons:

- Built for React i.e not multiplatform.
- NextJs is very difficult to pick up and start developing using it. And given our time constraints, it's not the best option

Conclusion

For this project we decided to go with nestJs. It's mainly inspired by angular, and our client insisted on using angular, so it fits perfectly with angular.

Nestjs is also very easy and intuitive to pick-up.

Nestjs allows the building of scalable applications. So this fits well with our microservices architectural pattern.

6.3. Database

6.3.1. Postgres

Postgres is an object-relational database system that uses and extends the SQL language. Supports both SQL and non relational queries.

Pros:

- Postgres accepts both SQL and JSON queries
- Can safely store and scale most complicated data
- Supports a large number of data types
- Support writing database functions in many languages such as Python, Java, Javascript, SQL, R and many more
- Gives the ability to define complex data types
- Conforms to the SQL standards

Cons:

- Many applications support MySQL but not Postgres
- Slower than MySQL

6.3.2. MongoDB

MongoDB is a cross-platform document oriented distributed database. The data is stored using JSON-like structures. Each Document can have its own structure with varying complexities. MongoDB works with binary JSON which provides some level of flexibility to what one can add to the database.

Pros:

- Has its own query language that is simplified and easy to use
- Does not have schemas so it has data flexibility

Cons:

- Offers less flexibility when querying

- The data size is higher

6.3.3. MySQL

MySQL is an open-source relational database management system. Its name is a combination of "My", the name of co-founder Michael Widenius's daughter, and "SQL", the abbreviation for Structured Query Language

Pros:

- Very easy to use and start learning
- Compatible with all manner of languages such as C++, Java, PHP, etc.

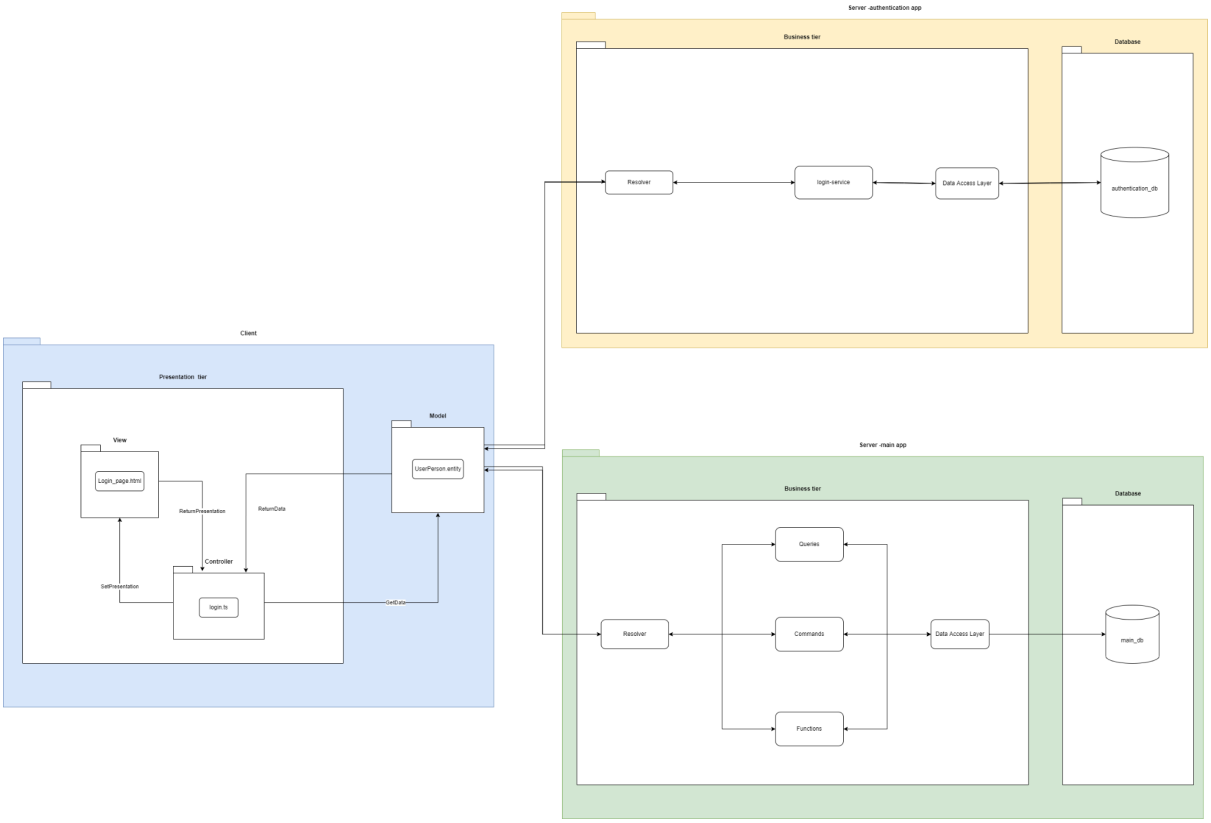
Cons:

- MySQL does not support a very large database size as efficiently.
- The functionality tends to be heavily dependent on the addons.

Conclusion:

Our data is more relational based and that is why we went with Postgres. We have entities that are closely related and rely on each other like teams, projects and employees. Postgres also provides data integrity which improves security.

7. System Architecture diagram



This gets expressed in more detail on our Github wiki page: [Architecture_v5]