Critical Design Assessment

FDM05

Contents

1. Introduction

This report will critically assess two proposed technical design solutions (Design A and Design B), using the chosen design goals as a major basis for comparison: reliability, reusability, and maintainability. Each proposal has been determined through separate design strategies to meet said goals based on different approaches, the effectiveness of each strategy will also be compared.

2. Design Strategies

Strategy A uses an academic approach to software design, relying on liberal use of design patterns and careful consideration of SOLID principles. The reasoning for such is to assess the validity of ingrained practices within the software industry against the chosen design goals.

On the other hand, strategy B looks to current trends within the industry for high-level system architecture, in order to meet the design goals. This methodology aims to determine how a well thought out network architecture compares to a design which places low-level code at the forefront. It is thought that architecture is fundamentally much more difficult to change down the road than low-level code implementations

The aim is to compare two approaches at the opposite end of the spectrum, posing the question: is a solid architectural foundation more crucial than the codebase itself?

3. Data Model

Strategy A's methodology influences the decision to use relational technology based on its maturity and synergy with many classic design patterns and good practices, due to its popularity when many of these were discovered. A major characteristic of relational databases is the rigid schema requirements they impose, making data extremely easy to maintain but making changes to the defined schema can be a difficult process. However, the draw to these technologies is the ACID guarantees that they provide, essentially ensuring data validity on transactions. Unfortunately, relational technologies such as SQL are, as expected, prone to SQL injection and their ability to scale only vertically provides a single point of failure for exploitation and service failure. Both are issues that affect the reliability and security of a software solution.

Strategy B focuses on the use of a non-relational / NoSQL technology (specifically document storage) due to being architecturally agnostic and its prominence in software trends at present. One main advantage of NoSQL databases is the ability to scale horizontally. Due to its distribution of data, NoSQL databases can scale horizontally across multiple servers or containers, which brings reliability through additional obstacles for data security and failover prevention. Another major factor in NoSQL databases is the loose schema support they provide. Both benefits are subject to the fact that they create additional points of consideration to ensure each data source is up to date and consistent, as there are generally no ACID guarantees on data manipulation.

The other potential choices for implementation of the data model were non-relational technologies. The first option was a KV (Key-Value) store, which was avoided due to its similarity to tabular and relational databases, and thus providing less functionality to a document store. The latter provides support for JSON and XML data, meaning the stored data can be much more complex if needed. The second potential option was graph data. While it seems to be popular within the industry, it can be more complex and provides much more functionality than is required for this project; its use of graph theory also makes most classic design patterns inapplicable to the technology.

4. Architecture and Codebase

Strategy A influences the high-level design into use of a monolithic approach, which makes use of many moving parts in a single codebase and lends itself well to robust, well thought out systems that require use of SOLID and careful consideration of good practices. Much like a relational database, monolithic architecture can only be scaled vertically, creating a single point of failure for reliability and security. However, the use of a single codebase ensures components and low-level implementations can be well designed to ensure maximum reusability. The main issue with monolithic architecture is the potential size of the codebase, and the issues that this presents for maintainability.

Strategy A also aims to embody design patterns, SOLID principles, and overall OOP good practice through meticulous low-level design. The main benefit of this approach is being able to map out the code before any code is written, which should theoretically support a smoother development process. This is done through mapping out low-level functionality and ensuring code sticks to good practice from the outset, encouraging good reusability and maintainability. However, the design could still be subject to change as it is difficult to foresee issues that will arise during the process; with the rise of agile methods in the industry UML and meticulous design may be outdated advice.

Strategy B presents endless options for high-level design, as this is where the most impact will be made. Microservice architecture meets the design goals in a multitude of ways, so this is the approach that was taken. It can be useful to look at microservices similar to objects within object-oriented programming (OOP), and they share many of the same attributes. Microservices enable loose coupling when designed appropriately and enable high-level reusability through the use of utility packages that may be inherited by each service. This thought process also allows services to be easily maintainable due to high cohesion and small scope. Additionally, the ability to scale microservices horizontally encourages reliability through many features such as server distribution and load balancing. Finally, it is typical to make use of an API gateway when connecting to microservices from a client, creating an extra layer of security through obscurity.

Due to strategy B's focus on architecture, low-level details are determined to be up to interpretation throughout implementation. This allows a developer to use the best tools for the job, as appropriate. For example, use of secure libraries for implementation of certain functionality, to save time or provide more secure and reliable implementations of security features – such as data encryption. However, a major issue with this approach is the lack of forethought from best practices and the lack of documentation, which can be an issue for maintainability.

A third major contender for this part of the solution was the use of n-tier architecture, whereby the application is separated into many consecutive layers of servers. Each layer corresponds to a high-level function, such as data access, business logic, and client-side code. This solution was avoided due to each additional layer within the stack creating an extra layer in which a request must pass to perform an action. The original brief places application performance in high regard, which monolithic and microservice architecture seem to be more able in catering for.

5. Discussion

Both strategies meet the desired design goals in various ways and measuring this metric will ultimately decide which approach is best for the final software solution.

Strategy B shows significantly more promise in reliability than strategy A. Each component in question supports horizontal scalability, meaning a more reliable service can be provided to the end user for various reasons. Strategy A falls short of this design goal in many ways, as there are too many single points of failure within the solution, both in terms of service reliability and security.

Strategy A follows a traditional OOP approach in conjunction with monolithic architecture, which provides various methods to ensure design can accommodate maximum reusability. Strategy B can support this if microservices are viewed in a similar way to objects. However, the highly decoupled architecture can cause issues for code re-use as it requires more consideration to achieve than a monolithic approach.

Strategy A's large, monolithic codebase may be difficult to maintain as it can be difficult to understand the whole thing when working on functionality. Large codebases can be intimidating, even if they stick to best practices and are well-designed. However, the use of UML and low-level documentation from the outset would allow a new developer to understand the relationships of objects before looking at the code. On the other hand, strategy B's microservices can be easily maintained when the scope is small, as there should only be a small amount of code to understand when working on a single service.

In conclusion, both strategies hold benefits and drawbacks in relation to the design goals, with strategy A being more reusable and strategy B being superior in reliability. The strategies are arguably very similar in terms of maintainability.