ENCOMPASS

A picture containing icon

Description automatically generatedA picture containing icon

Description automatically generated

|  |  |
| --- | --- |
| Name | Student Number |
| JULIANNA VENTER | U20433752 |
| MORGAN BENTLEY | U18103007 |
| Keabetswe Mothapo | U21543462 |
| Ronin Brookes | U19069686 |
| Sameet Keshav | U21479373 |

Architecture Document

Contents

[Design Strategy Based on quality requirements and decomposition. 3](#_Toc138497940)

[Quality requirements: 3](#_Toc138497941)

[Decomposition: 3](#_Toc138497942)

[Architectural Patterns and Styles 3](#_Toc138497943)

[**Event-Driven Architecture** 4](#_Toc138497944)

[**Monolithic & CQRS** 5](#_Toc138497945)

[**Multi-Tier** 6](#_Toc138497946)

[**Model-View-Controller** 8](#_Toc138497947)

[Architecture Design: 9](#_Toc138497948)

[Architectural Quality Requirements 12](#_Toc138497949)

[**Maintainability** 12](#_Toc138497950)

[**Scalability and Performance** 12](#_Toc138497951)

[**Security** 13](#_Toc138497952)

[**Usability** 13](#_Toc138497953)

[**Reliability** 14](#_Toc138497954)

[Architectural Constraints 14](#_Toc138497955)

[Technology Choices: 14](#_Toc138497956)

**Design Strategy**  
Based on quality requirements and decomposition.

Quality requirements:

Based on determined quality requirements and chosen architectures reflect what the system needs to accomplish these requirements. These include quality attributes and non-functional requirements like scalability, usability, security, reliability, and maintainability.

Decomposition:

Based on taking a monolithic system and breaking it into smaller components, breaking the complex system into more manageable subsystems, helping with understandability and maintainability.

**Architectural Patterns and Styles**

Encompass, as a versatile social media application, adopts a robust architectural approach that leverages various patterns to ensure scalability, maintainability, and flexibility. The architectural patterns utilized in Encompass include Event Driven, Monolithic with CQRS, Multi-Tier, and Model-View-Controller (MVC). Each of these patterns brings its own set of benefits and addresses specific aspects of the application's design and functionality. By combining these patterns, Encompass achieves a well-structured and modular architecture that enables efficient handling of events, supports a scalable and efficient monolithic architecture with Command Query Responsibility Segregation (CQRS), separates concerns across multiple tiers, and maintains a clear separation of data, presentation, and business logic through the Model-View-Controller paradigm. This integration of architectural patterns empowers Encompass to deliver a robust and feature-rich social media experience to its users.

Event-Driven Architecture

The event-driven architecture used within Encompass is essential to mapping out the application’s real-time functionality. This is particularly important considering the need for real time updates on posts, messaging, notifications, as well as the recommendation aspect of the content.

The user establishes a new connection with the websocket API, which sends the user actions or commands to the interface, which then transforms the user input into command messages. These command messages are sent to the Message broker, which acts as the central communication hub for the event-driven architecture, which then routes these commands to the appropriate application service.

Once these commands are routed to their service, they are passed to their respective event handler, which validates the command, executes the business logic of the command, and updates the system’s state. Those generated events then get passed back to the message broker. The message broker then sends the notifier those published events, which in turn delivers them to the websocket.

The respective database (either key/value pair or media, depending on the nature of the event) gets updated based on the processed events, and returns relevant data for any queries.

Why use event driven?

* simplifies horizontal scalability in [distributed computing](https://en.wikipedia.org/wiki/Distributed_computing) models
* More resilient to failure
* Good for real-time functionality

A picture containing diagram, sketch, technical drawing, plan

Description automatically generated

Monolithic & CQRS

All the components and modules of the system are tightly coupled and deployed together. This is used in combination with CQRS to divide the commands (write operations) and the queries (read operations). The system is self-contained and independent, but lacks flexibility, thus implementation of multi-tier and adding CQRS to compensate.

*Commands:*

* Responsible for modifying the state of the chat system.
* Processes message sending, user authentication, and other write operations.
* Writes events to the event store.

*Queries:*

* Responsible for retrieving data from the chat system.
* Handles message retrieval, user profile display, and other read operations.
* Reads events from the event store and generates query-specific views.

Multi-Tier

The multi-tier architecture divides the Encompass application into three distinct tiers: the presentation tier, the application/business logic tier, and the data/storage tier. This separation enables modular development, scalability, and ease of maintenance.

*Presentation Tier:*

The presentation tier is responsible for handling user interactions and displaying the app interface. It includes components such as the user interface (UI) and the client-side application logic.

*Application/Business Logic Tier:*

The application/business logic tier contains the core logic of the Encompass application. It handles user authentication, message processing, AI, and business rules. This tier communicates with the presentation tier and the data/storage tier.

*Data/Storage Tier:*

The data/storage tier handles the persistence and retrieval of relevant data to the Encompass app user.

Advantages:

* Introduces flexibility and reusability.
* Modularity: add or remove tiers instead or reworking entire system

Model-View-Controller

Separates the application logic into three interconnected components:

*Model:*

The Model represents the data and business logic of the application. It encapsulates the data and defines how it can be accessed, modified, or manipulated. The Model component is responsible for maintaining the integrity and consistency of the data.

*View:*

The View represents the presentation layer of the application. It is responsible for displaying the data to the user and providing an interface for user interaction. The View receives data from the Model and presents it in a visually understandable format. It also sends user input or actions to the Controller.

*Controller:*

The Controller acts as an intermediary between the Model and the View. It receives user input or actions from the View, processes them, and interacts with the Model accordingly. The Controller updates the Model based on user actions and retrieves data from the Model to update the View. It manages the flow of data and controls the overall behavior of the application.

Advantages:

* Easier maintenance, extensibility, and reusability
* Promotes loose coupling.
* Each component has its own core functionality independent of other components.
* Separates how the information is presented vs how the information is accepted by the user.

**Architecture Design:**A picture containing text, diagram, plan, technical drawing

Description automatically generated

**Architectural Quality Requirements**

* Maintainability
* Scalability and performance
* Security
* Usability
* Reliability

**Maintainability**

The system should remain easy to maintain throughout development and for any future iterations of the application. This would require the system to be set up in a way that makes the system easy to understand and enable developers to change or update a system component without affecting the other system components.

To quantify this requirement we have designed the system with a modular architecture in mind, like MVC, Multi-tier, and CQRS, so that changes to any component in the respective layer or module can be added without need of in-depth understanding of or attention to the rest of the system. Another aspect would be to keep code modular and follow best coding practices.

**Scalability and Performance**

As a social media application that promotes interaction with (possible) vast communities, the concept of scalability is very important. The application needs to be able to handle an increase in users as well as their personalized data, without sacrificing the applications performance.

To quantify this requirement, modular design with the architectures mentioned in the above requirement will greatly improve ease of scalability. With the use of AWS services, a flexible option for storage is guaranteed, especially for use with visual media and the data needed for personalization algorithms. The system will be able to handle at least 20 users, with their recommendation/personalization data, before scaling up to AWS’s paid tiers.

**Security**

As an application that deals with a lot of user data, especially gathered personal data, security is a primary concern. Architectures like Multi-tier allows for measures to be implemented at each layer.

The security requirement will be quantified using encryption and permissions. Users will need authorization and authentication through login details, and details will be encrypted before storage. This extends to non-public data as well.

**Usability**

The primary requirement for a social media application is ease of use. If the application lacks user-friendliness, it will inevitably result in a decline in user engagement and retention. Frustrations and negative user experiences tend to deter users from staying actively involved with the application and its content, potentially leading to user abandonment.

Quantified in the UI/UX design to help users navigate the application without confusion or frustration, as well as interactable elements and gamification to keep attention and activity.

**Reliability**

As a social media app the system must be available and operable constantly. It should also support continuous integration featuring bug fixes and updates.

This is quantifiable by it’s use of AWS hosting for 99% activity once deployed, as well as the ability to deploy new versions once updates are sent out for deployment.

**Architectural Constraints**

The following constraints are imparted on the system, and solutions to handling them have been discussed in previous sections of this document:

* System must be horizontally scalable
* Technologies used must be open source (no monetary cost)
* The system should have security measures for the verification of a user

**Technology Choices:**

**Angular:**

Overview:

Pros:

Cons:

Reasoning:

**NestJS:**

Overview:

Pros:

Cons:

Reasoning:

**MongoDB:**

Overview:

Pros:

Cons:

Reasoning:

**AWS Hosting:**

Overview:

Pros:

Cons:

Reasoning:

A picture containing icon

Description automatically generated