2DV609

Växjö Volunteers
Design Document

Author1: Atakan Coban
Author2: Albert Henmyr
Author3: Alija Levic
Author4: Kateryna Melnyk
Author5: Richard Oelschlager

Content

1. Purpose

This document describes the high-level design choices that shape the “Växjö Volunteers” system design process.

Section 2 identifies principles that will drive the design philosophy, as well as goals that the whole design process aims to achieve. It also addresses some critical and unusual conditions that should be taken into account in the design stage for the system to function effectively.

Section 3 lists assumptions and dependencies that will drive the design decisions and contains the motivation of the design approach that will be applied to the software development process.

Section 4 is focused on the requirements that significantly affect the design as well as on the description of their implementation.

The purpose of Section 5 is to summarize the accepted design decisions and constraints that are imposed on them. This section also provides the different alternatives to the design decisions and justifies the choice of the final solution.

Thereafter the critical concepts that define the design of the system are highlighted in Section 6.

Section 7 discusses the architectural patterns and whether the taken architectural choices will address the design principles in Section 2 and lead to consistent system architecture. Whereas Section 8 illustrates the architectural decisions in component, deployment and sequence diagrams.

Finally, Section 9 provides the performance modelling of the system and the analysis of the model simulations based on the indicators System Response Time, Utilization and Throughput.

2. Design goals and philosophy

Iterative design methodology with cyclical revisions and refinements of the software will be applied to the whole process, thus resulting in design-driven and test-driven software development. The design philosophy will be focused on developing a product that will provide a meaningful experience to the user.

2.1. Design principles

In general, the software design will adhere to the following principles:

1. **“You aren't gonna need it”** [1]. Design of the necessary functionality that is specified in the requirements without any attempts to integrate features that supposedly may be needed in the future.
2. **External dependencies on well-documented and reliable software**. Integrating reliable and maintainable external frameworks (if such are needed) will reduce the risk of compromising the native application design and its runtime stability. This principle includes utilizing the **reusability** of existing software in the design of the new application.
3. **Flexible design** implies
    a. reducing **coupling,** meaning avoiding the design of interdependent subsystems where changes in one subsystem will inevitably lead to the changes in other subsystems;
    b. increasing **cohesion,** meaning organizing the system’s components into logically related subsystems;
    c. increasing **abstraction,** thus avoiding code duplication and hiding the implementation details of subsystems;
    d. prioritizing **reusability** wherever possible through functional cohesion, i.e. have modules that perform a single common operation and return a result if necessary, without resulting in side effects.
    e. considering potential **extensibility** of the system nevertheless avoiding actual implementation of redundant features or overengineering.
4. **Testability**. Ensuring that the system’s functionality can be automatically tested without the reliance on GUI input.
5. **Pattern-oriented approach**. In order to “not reinvent a wheel”, the design should make use of known architectural and software design patterns to deal with typical design and implementation problems.
6. **Reducing complexity through the “divide and conquer” principle**. Breaking complex problems into smaller ones that can be successfully managed one step at a time. The software system will be divided into smaller pieces that can be developed in parallel. It is possible for a single

software engineer to specialize in their component, but not for the entire system.

7. **Avoiding overengineering by following the KISS principle** [2]. Conforming to this principle must not go against the implementation of the system-wide requirements.

2.2. Design goals

The following design goals should be met as the foundation of the overall process:

1. **Compliance with the system-wide requirements and business rules**. It is important to achieve the functionality that is defined in the requirements specification in order to fulfill the stakeholders' expectations.
2. **Minimization of development costs**. "Växjö Volunteers" mobile application is a non-profit project, therefore the design process should aim to integrate open-source external software if such is needed.
3. **Efficiency**. Aiming to effectively utilize the development time, the capacity of the database storage and human resources.
4. **Traceability**. All the changes to the software design should be traceable and available for analysis over time.

It is worth mentioning that the design should address the issue of the critical load on the system by estimating the approximate number of concurrent users that are sending simultaneous requests to the application system based on the size of the target audience (potential volunteers in Växjö). By extension, it is implied that the problem of user requests waiting in a queue for the service for a long period of time should be taken into the consideration.

3. Assumptions and dependencies

The assumptions and dependencies are listed in no specific order.

3.1. Mobile application

The customer has requested the main component to be deployed on mobile devices (particularly smartphones), as such the software decisions will reflect this.

3.2. Concurrent user sessions

The application will be designed with the assumption in mind that at any given point in time there will be at most 100 users accessing the application. This number was determined through consultation with the potential users and picked to not have to rely on a distributed design as doing so would increase developer hours for not much gain for the targeted use-case.

3.2. Google Service Availability

As the application will utilize the Google OAuth service for validating identities and populating data per requirement USR-001 (*When a user registers, a profile is created and populated with data from Google*), there is a dependency that requires Google to be available at all times; without Google we can not sufficiently link an identity to a user per BKN-001 (*The system must allow a user to register and authenticate through Google*).

3.3. Design

By using Flutter we will approach mixed design. The application and its layout will be applied top-down, while the modules that will encompass the system and integrate with the various parts of the UI will be done bottom-up and then integrated into the system as components.

By combining mixed design we will have a better overall understanding of the system being built. Furthermore, the components can be built by developers in parallel while the top-down design is carried out together as a developer team. By using this design we will rely on interfaces that developers will model the integration parts from. Refer to sections Architectural Patterns and Views for more details regarding our design flow.

In addition, relying on mixed design will have the advantage of building the small components we will utilize, such as OAuth flow, which can be done in parallel while the application itself is also designed.

As an alternative, by designing our system solely as top-down, we would have less coupling but it would be unfeasible to design components on the fly while developing. While doing only bottom-up design we would lose the fine-tuning and

general planning of the design, which in turn would have meant that we have components, but not a proper plan on how to integrate them into the overall architecture.

3.4. Cross-platform development tool

It is assumed that the cross-platform mobile application development (BKN-003 *"The application must run on Android and iOS"*) requires dependency on the third-party tool that will be well-documented and maintained thus reducing the risk of the reliance on the external software and optimizing the development process by using already existing solution for the cross-platform implementation.

3.5. Platform application stores

Our iOS version of the mobile application will be published on the App Store managed by Apple.

Our Android version of the mobile application will be published to Google Play Store managed by Google.

3.6. Geographic restriction

As our application will be used by Växjö citizens we will limit ourselves to Växjö. In turn, this means our hosting and marketing (Word-of-Mouth Marketing) will be strictly kept to Sweden; as such we avoid user data having to be kept in other countries.

As we have strict standards to follow, such as GDPR (General Data Protection Regulation), and other privacy laws that are in effect in Sweden, we will focus on our target country Sweden (specifically Växjö community) to avoid dealing with even further regulations that might exist outside Sweden. In turn, this also reduces our legal liability to not have to account for countries outside using our application as they must physically be present in Sweden.

3.7. User traffic

We estimate around 5184 requests during a single day, 1/6 of them being from moderators. The first number is based on the population of Växjö and our assumptions regarding how many of its citizens will use it and how often. The moderators, while probably only a handful, are expected to be far more active as they carry out their duties.

4. Architecturally significant requirements

4.1. Networking

It goes without saying that the fundamental nature of the application as one for communication between users means that some form of networking is required. Two important requirements that exemplify this to follow.

USR-003: A user shall be able to list all volunteer requests.
USR-005: A user shall be able to submit a volunteer request. It shall be at most 1,000 characters long.

4.2. Client-server vs. peer-to-peer

Notable requirements:
MOD-001: A new post or changes to an existing post must successfully pass through an automated moderation filter to be approved for public view.
MOD-002: A moderator should be able to delete posts by other users or moderators from the public view.
MOD-003: Rejected posts should only be visible to the author and moderators.
MOD-006: Moderators can issue and lift bans on user accounts.

BKN-002: The system must store data about users and posts and not lose it if it reboots.

Multiple MOD requirements (primarily **MOD-001**, **MOD-002**, **MOD-003** and **MOD-006**) demand the existence of some form of higher authority within the architecture itself. There needs to be a system entity that dictates what posts are visible to whom, enforcing the decisions made by the moderators. This could be handled via a client-server design in which the server plays a central role.

BKN-002 adds value to this solution as opposed to a more peer-to-peer-oriented approach in which the server is only peripherally involved. An unspoken assumption is that every user has access to the most recent list of posts, and this is ideally accomplished by a server with a database always ready to receive and distribute updates to this list.

A peer-to-peer solution would make it take longer for changes to propagate through the user network, as users may opt to not participate in the peer-to-peer exchange unless they are actively using the application, and even then could be stingy about using bandwidth to upload. Of course, the reliance on a server does instead mean that it has to provide high availability.

4.3. SDK

BKN-003 *(The application will be designed for both iOS and Android. All USR tests must pass on both platforms)* is a major requirement with major implications. The ideal solution involves using SDK for cross-platform mobile application development that contains a set of tools (libraries, frameworks, compile toolchain, etc.) and therefore allows per-platform fine-tuning of application as necessary. This will result in the highly desirable effect of a shared codebase between the platforms, meaning that development and maintenance only need to be done in one place, minimizing the risk of divergent design due to different understanding between developers, or forgetting to apply a change in one place.

Here we must also consider **BKN-001** *(The system must allow a user to register and authenticate through Google)*, further constraining our approach to **BKN-003** to be one that reliably interfaces with Google's authentication services. On top of this, the UI-related **LAF**, **LAN** and **CON** families of requirements (section 3 of the requirements specification), while each minor, suggest that we should select SDK that can easily meet a majority of these.

5. Decisions, constraints, priorities and justifications

The topics are ordered based on priority for the project..

5.1. Flutter - Framework / SDK

Flutter was decided as we have to support both iOS and Android per assumption, mobile application. Flutter allows us to develop all underlying components to be modular and take advantage of what Flutter can offer us.

It allows saving developers time as the code-base will be shared between both operating systems with minimal platform-dependent code. The system is heavily dependent on this decision as Flutter is the main framework and as such every other component has to be built with Flutter in mind.

Other alternatives consisted of using React native which would have been possible to develop the system as it also supports both platforms and has a similar feature set to Flutter. As developer skills were more aligned with Java, the decision to stick with Flutter, which is more similar as it uses Dart, was made. Development time would most likely have been higher to adapt to React native.

Another alternative was to develop the Android application fully to be native in Kotlin and also create a native iOS app made in Swift; this would have split our codebase up in two and also increased development time as reuse of components would have been minimal. Benefits to making native apps to both would have been performance, but as our system is not latency heavy this was not a justification taken when deciding upon a library.

An additional constraint on using Flutter is that we will only support the iOS and Android versions that Flutter supports at a minimum compatible level, not accounting for any third-party libraries that may be used.

5.2. Client-Server model

The system will be designed with client-server architecture for our major logic. By utilizing the client-server design pattern we can have a more uniform consistent approach of keeping database logic separated from client devices and having it abstracted behind an API endpoint which will give us more flexibility when doing breaking changes.

As our client, in this case, the mobile application built on Flutter will communicate with our backend. We can abstract away all the database logic making edits easier as none of the SQL logic is kept on the client application.

The framework used for the server backend will be Javalin. Javalin is a web framework that facilitates creating rest communication through a modular approach, in turn this allows developer focus to be assigned to more important tasks. There are

other alternatives for web frameworks, such as Spring, but as we require a very simple approach Javalin works as we do not require all enterprise functionality of Spring.

There are other alternatives such as the broker pattern also mentioned in Section 5.6. MVVM - Client.
Refer to section Section 7.2 Client-Server Pattern for reasoning and explanation regarding the pattern.

5.3. OAuth - Third Party authentication provider

Relying on OAuth and verifying identities through Google ensures we leave the security & handling of the authentication data to a known and trusted provider that has the legal team to deal with any issues that may arise.

By also deciding to go with OAuth we do not have to store any passwords and can construct our system by focusing more on other aspects.

Other alternatives would have consisted of having the user authenticate with the device ID, which in turn means it's not portable, but would have the added effect of no accounts. They would be persistent only to the device itself; as users switch devices a lot this would not have scaled well.

Having a local authentication server that the user will authenticate to would have been an alternative, but in return would have added more complexity, as we would handle the authentication process of verifying and recovering users' accounts. In this case, we would have the added complexity of having to allocate resources to verifying users if they lost their accounts as we handle the entire process.

5.4. PostgreSQL - Database

PostgreSQL is a database server that operates on the CRUD principle, create/read / update/delete, where these operations must be atomic. It will be the storage component of our system. The requirement BKN-002 states that we must not lose data if it reboots; PostgreSQL has good error recovery.

PostgreSQL is also a structured query language. Alternatives here consisted of MySQL, but due to licensing constraints, it was not a better pick as the feature set of PostgreSQL contains more useful features.

Alternatives to SQL would have been NoSQL databases such as MongoDB. As we have a sane structure for the application in terms of handling submissions, users picking MongoDB would have made the data unorganized and as such all NoSQL alternatives were not considered after the planning was made.

The entire feature set of PostgreSQL will not be used but only the functionality required for our application. Even in this case picking PostgreSQL is a sane option.

5.5. MVVM - Client

The client, in this case referring to our mobile application, will be designed using the MVVM design pattern. MVVM (Model-View-ViewModel) is a design pattern that can be integrated with Flutter without having issues.

There are other alternatives such as the BloC pattern, which is a more abstracted event-driven pattern similar to MVVM. Picking this over MVVM would not have made much difference in our design and the choice was mostly down to familiarity.

Other alternatives would have included any pattern suitable for picking here, such as a broker pattern that would have been more tightly integrated with our Client-Server Model per Section 5.5 in the same section.

Refer to section Section 7.1 Model-View-ViewModel Pattern for reasoning and explanation of the pattern.

5.6. Styling - Material design

We have opted to design the entire application with material theme design and flow. This will give the application more accessibility and an easier user interface which will mimic Google's design of applications and Android design itself.

As justification, by utilizing material themes we avoid the need to design color palettes and styling guides as the information is publicly available to follow. Alternatives to material design are many, but by omitting having to design our own styles we can focus more on the logic of the application.

6. Key abstractions

6.1. Making requests

The first half of the main concept is pairing those who request help with those who are willing to.

The application must support a way for users to present others with details of an event they have the chance to volunteer for.

Mainly, details of the event and some form of contact information must be presented, as they are the essential elements for others to find interest and then eventually participate. While other features are useful, they are not the essence for things to function as the aforementioned are.

6.2. Finding volunteering opportunities

The other half of the main concept.

Users must have a way to browse through what may potentially become a seemingly endless list of volunteering events with no necessary predictability at sight, allowing for many opportunities. The users should be able to surf over the main idea of each event, not needing to dive into the details before finding one that grabs their attention.

Furthermore, a user must be able to broadly filter requests, allowing them to stay focused on particular types of events while preserving the unbound nature of the results to those that have passed the requirements of the general-purpose of the filter and rules of moderation.

6.3. Moderation

Both posts and the users who read/write them must all be subject to preventative and otherwise constant moderation.

The preventative side of moderation lies in the effectiveness of an automated system, which must check uploaded content for commonly known and unapproved material. Furthermore, the application must always be under surveillance by human moderators with the power to delete troublesome content. They may either stumble upon something that seems troublesome or be pointed towards it by the community of users who must be able to report potential issues during their usage of the application.

6.4. Social interaction

The application may easily be considered a social network - a platform where users interact with each other. The interactions, however, are generally more limited in terms of features and specific in terms of content compared to other mainstream social networking services. The types of interaction that must be present are as follows.

Writing requests, as mentioned above, allows users to present others with details of an event while asking for support regarding it.

The application also serves as a bridge for later communication that may occur thanks to provided contact information, during an event or any relationships that may begin having started from the app.

Finally, users have their own profiles where they describe themselves in the “About me”.

7. Architectural Patterns

The key architectural concept for the application will be **Three-Tier Application** Architecture. This approach to architecting is widely used within enterprise applications and is also part of many design patterns.

The three tiers are :

- *Presentation layer*: The user interface (UI).
- *Business layer*: The business rules and their validation.
- *Data Access layer*: Persistency and storage.

These layers provide a solution to encapsulate the logic that each part performs. Data can move across the layers and appear in all of them and each layer has the capacity of capturing the data and processing it.

7.1. Model-View-ViewModel Pattern (MVVM)

The **MVVM** pattern was created as a replacement for the **MVC** and **MVP** patterns. This paradigm was first proposed by SmallTalk in the 1980s, under the term Application Model, and then renamed to Presentation Model before settling on **MVVM**. [4] (Siddiqi & Vice, 2012)

Our reasoning for opting for **MVVM** is that we need our code to be broken up into classes with a restricted number of well-defined responsibilities. **MVVM** will then serve as a guide on how to distribute responsibilities between classes for a GUI application. **MVVM** is also proven to integrate well with Flutter. A secondary benefit but one of great relevance is that having the code be encapsulated as described will make it much easier to be tested.

The **Model** in **MVVM** is very similar to the **Model** from Model-View-Controller (**MVC**). It is in charge of granting access to resources like datasets, files or servers. There is generally speaking not a lot of code usually housed within the **Model** in an **MVVM** implementation. The **Model** is mostly situated on the *Data Access layer* but it also covers some aspects of the *Business layer*, because data often requires transformation at the *Business layer*.

The **View** has the responsibility of displaying data in a proper format, reflecting the data's status and relaying it to other components, and of recording user interactions and events. Like the **Model**, the **View** consists of only the implementation code that is necessary for it to operate and enable user actions. The **View** resides entirely within the *Presentation layer*.

The **ViewModel** contains the majority of the code within **MVVM**. The idea behind the **ViewModel** as a component is that it reflects the intended state of the **View** and how it responds to user interactions. To that end, it provides a set of rules and structures that portray particular data acquired via the **Model**. It also facilitates communication between the **View** and the **Model** by delivering the relevant data

from **View** to **Model** in an easy to process format. Validation happens within the **ViewModel** as well. The **ViewModel** has a wide range of responsibilities so it covers major parts of the *Business layer* and the *Presentation layer*. The side that implements the logic and the state of the **View** are *Presentation layer* responsibilities whereas logic that enables the manipulation of data comprise *Business layer* responsibilities. [3] (Kouraklis, 2016)

7.2. Client-Server Pattern

The **Client-Server** architectural pattern is a tried and true layered pattern that at its core splits up the code into front-end **client** and backend **server**.

This design pattern is focused on resources, meaning this approach encourages the idea of a centralized **server** that **clients** can connect to for sharing data. The same **server** also serves as a central point or hub for scaling the application.

Our application will consist of many **clients** that will need to be served, at times perhaps simultaneously. This is the key reason why the **Client-Server** architectural pattern must be implemented. A significant number of logical components are already dictated on the **client-side** by MVVM.

Bundling the parts of the application that perform functions that are identical in nature for all **clients** (*authentication*, *persistence*, *moderation*) into a backend **server** will lead to increased performance and a smaller footprint on the **client** device memory.

With the backend **server** being centralized, our system can allow for distributed processing and storage, which is especially useful as the data size increases.

The services that the **server** provides are just an abstraction for computer resources and the **client** is not concerned with how the **server** performs while processing the request and delivering a response.

Clients and **Servers** will communicate with each other in a request-response messaging pattern. The **client** requests and the **server** responds. This is also known as inter-process communication.

The formal implementation of the communication described beforehand is an application programming interface (API). An API is just an abstract layer for accessing a service.

Restricting communication to a specific content format enables the receiver to parse whatever is transmitted.

The **server** might receive many requests simultaneously but a computer always has a limit on the number of tasks it can handle at any given time. The solution to this problem is a scheduling algorithm to prioritize incoming tasks from **clients** and accommodate them.

In order to prevent malicious actions and abuse the **server** will also limit its availability to **clients.**

8. Architectural views

8.1. Component diagram

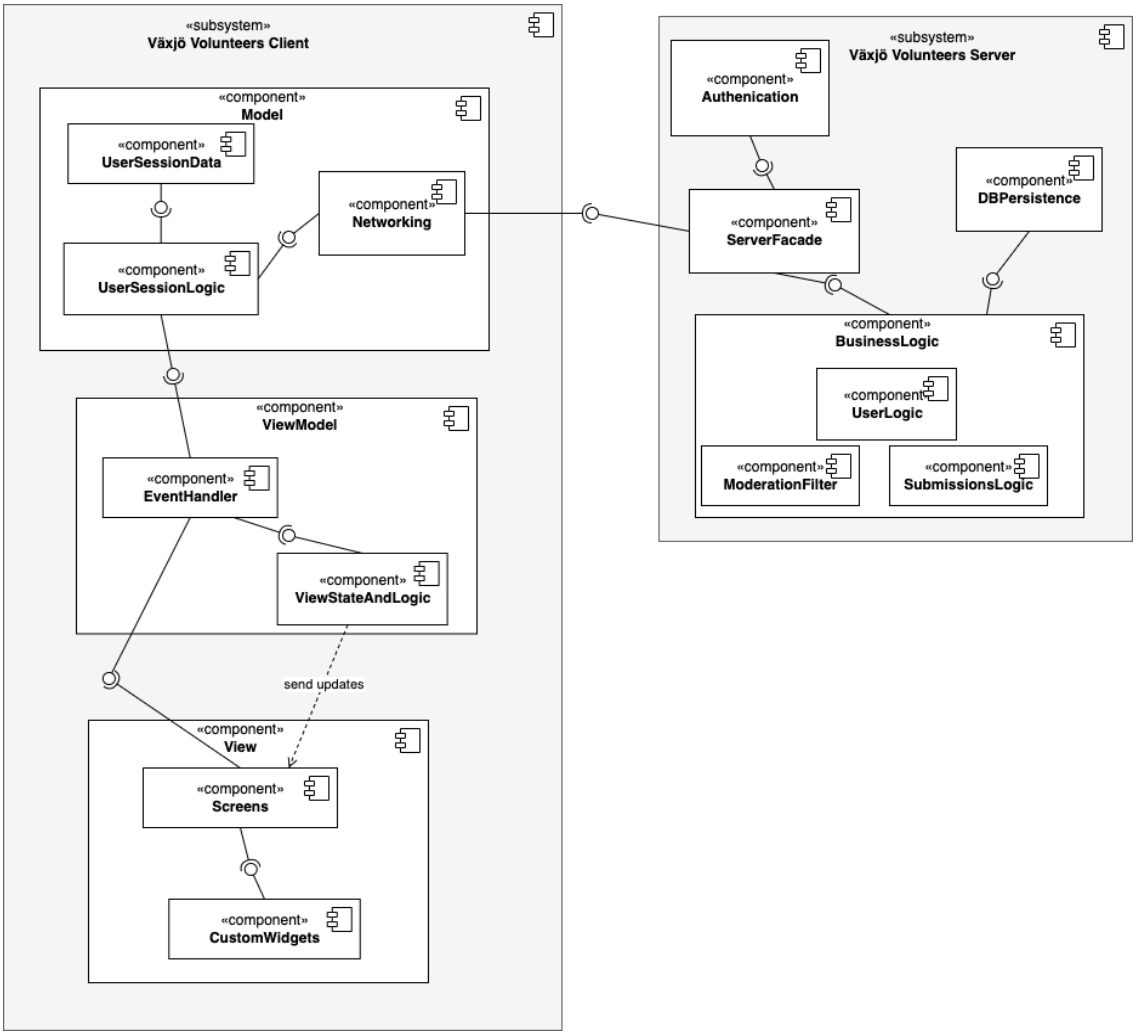

Figure 1. Component diagram. Växjö Volunteers system.

The client subsystem consists of 3 main components:

1. **Model**
    a. **UserSessionData** represents the temporary storage for the user data within the current user session. It exists in memory only during 1 user session and is updated by the calls to the application server.
    b. **UserSessionLogic** holds operations with the help of which the **UserSessionData** can be retrieved from Model as well as access to the **Networking** component on the client-side of the application system.

c. **Networking** is a set of functionality to handle calls to the application server-side.

2. **ViewModel** facilitates communication between the Model and the View and binds them together.
    a. **EventHandler** provides data to and from the user based on events that are happening on the View side**.**
    b. **ViewStateAndLogic** component preserves and updates the state of the View as well as contains logic to manipulate the data that is received from the Model in order to prepare it for the View display or fetched from the View for the subsequent passing it up to the Model. This component can be updated by the EventHandler functionality.
3. **View**
    a. **CustomWidgets** is a collection of separate GUI building blocks that are custom-made based on the inbuilt Flutter widgets.
    b. **Screens** component represents a grouping of **CustomWidgets** into a logically complete user interface views.

Server subsystem:

1. **ServerFacade** represents the entry point to the server subsystem**.** It encapsulates all the logic of how to handle the concurrent multiple clients' requests to the server. Besides its own functionality, it also uses the interfaces of the other components of the server subsystem (Authentication, Business Logic and indirectly DBPersistence), because by itself it doesn't contain any data or business logic to satisfy the client's needs.
2. **BusinessLogic** component holds the algorithms to handle/manipulate the client's data for the application system to provide real-world usefulness. It contains UserLogic (operations on the user's personal data), SubmissionLogic (operations on submissions/requests) and ModerationFilter (all algorithms that are related to the implementation of the automated moderation filter).
3. **DBPersistence** realizes the storage of the data within the application system.

8.2. Deployment diagram

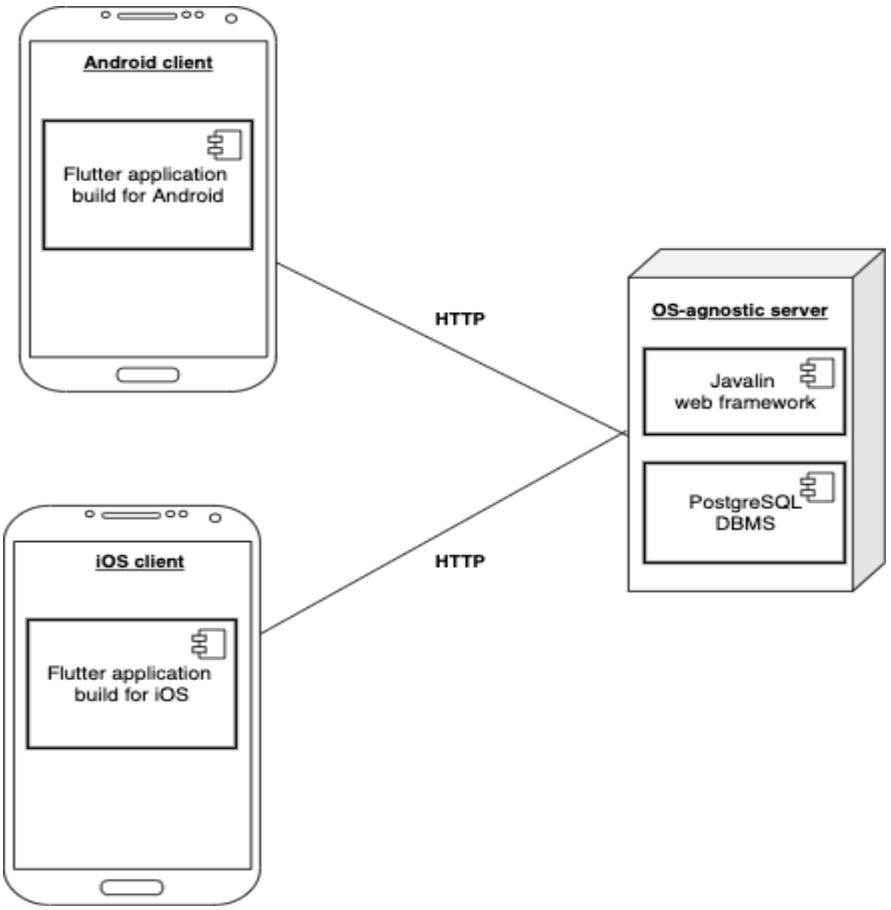

Figure 2. Deployment diagram. Växjö Volunteers system.

The client-side of the application system is deployed as the cross-platform flutter build for the respective mobile operating system (Android or iOS).

With regard to the backend, the server-side is using the Javalin framework which can be deployed as a self-contained web application. The Javalin documentation does not mention the specific list of OS that are supported implying that this framework is supported through many platforms where Java is available.

The database persistence functionality is realized using a self-managed PostgreSQL server running on the same machine. PostgreSQL is supported on a variety of OS such as macOS, Windows, Linux, FreeBSD, OpenBSD, and potentially further ones.

Thus, our backend implementation stack can be considered OS-agnostic to a reasonable extent with respect to the modern OS supported on physical or virtual servers.

8.3. Sequence diagrams

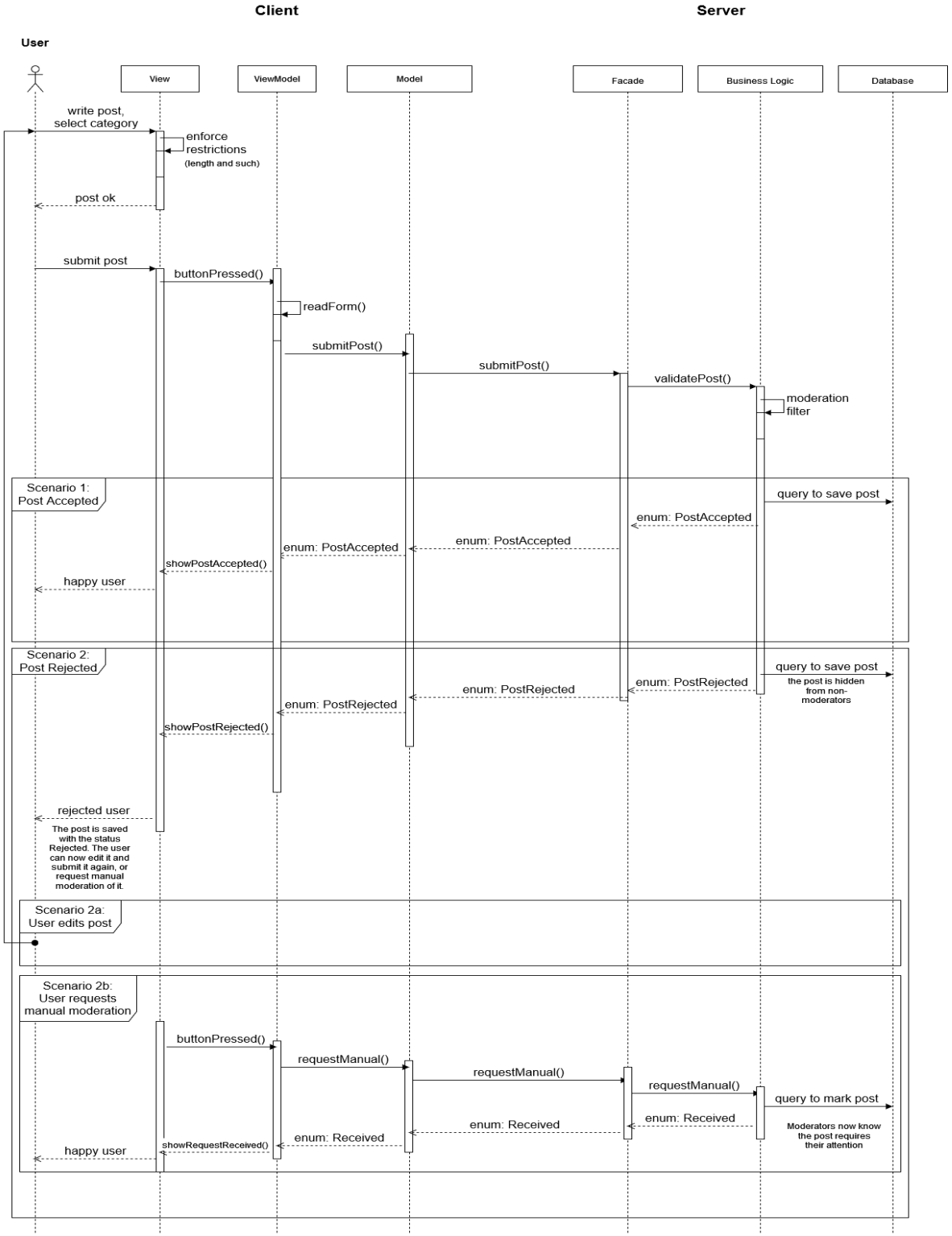

Figure 3.

The above diagram demonstrates the submission of a new request where automated moderation is key to ensuring a high quality of posts within the system. The user creates a post using the View, the ViewModel transforms it into a suitable format for the Model, which then handles the networking aspect. The server receives the post, and if it passes the moderation filter, saves it as a visible post (Scenario 1). If not, it is still saved, but flagged as Rejected and thus hidden (Scenario 2). The user is informed either way.

The user can then either edit the post and submit it again (Scenario 2a) or request that the post be manually moderated (Scenario 2b), in which case a human moderator will inspect it and determine if it's suitable.

Given that this is meant to be an abstraction intended to provide an overview of the core of the process, and that highly specific details of system interaction aren't established until the programming phase, many aspects have been omitted. These include handling network and database errors and the inner workings of each subsystem.

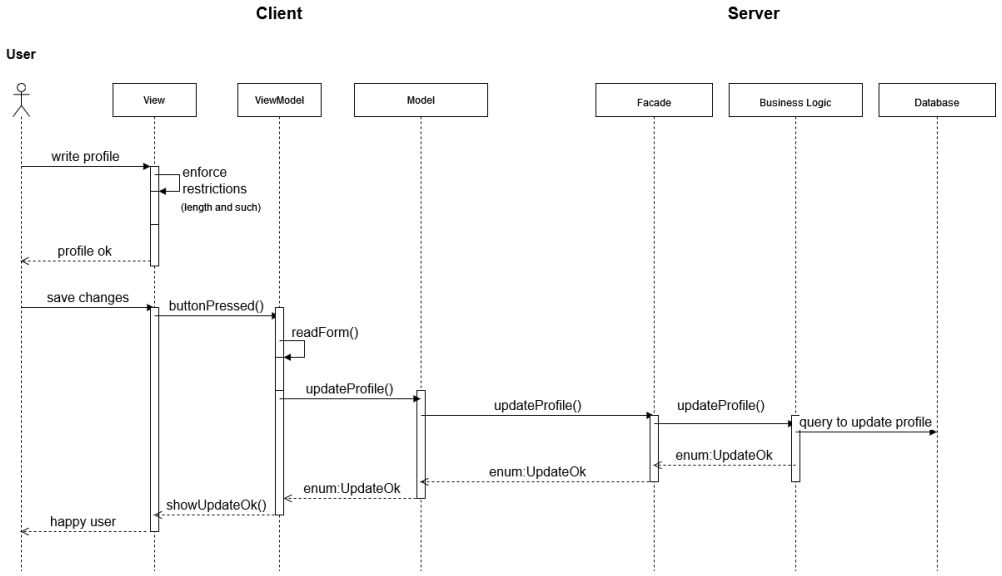

Figure 4.

This smaller diagram illustrates a similar process for editing the "About me" field. As there is no automatic moderation for this, the process is much simpler and only really subject to the same errors that were listed in the previous diagram and omitted for the same reasons.

9. Performance modeling and analysis

Over a span of 24 hours we assumed a total of 4320 user requests and 864 (20% of the user count) moderator requests, resulting in a total of 5184 requests on average where all of them were completed. As per our 3.7. User traffic.

This means that we are assuming our arrival rate is 0.05 for users calculated by dividing requests by total seconds in a single day (86400). In addition, moderation lambda equals 0.01 which means on an average day users will be five times more active.

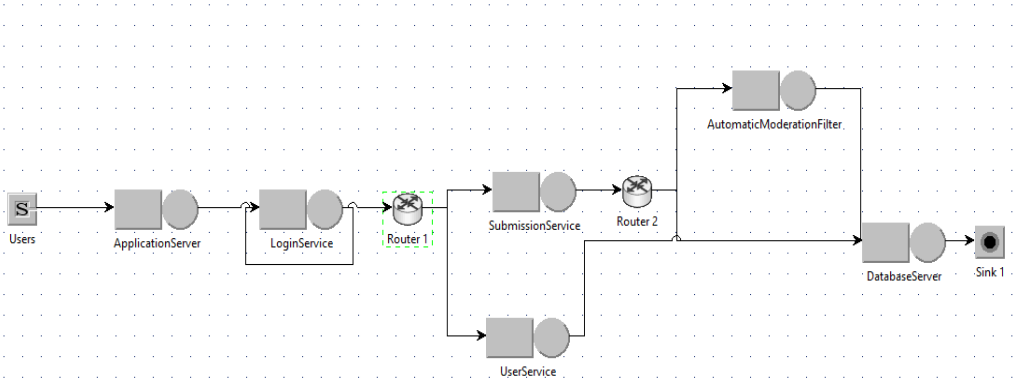

Figure 5. Service center layout

The system is composed of 6 service centers: A ***ApplicationServer***, a ***LoginService***, a ***SubmissionService***, a ***UserService***, an ***AutomaticModerationFilter*** and a ***DatabaseServer***.

The service center **SubmissionService** handles all logic related to submissions, such as posting new submissions and editing submissions, among other duties. **UserService** handles all operations related to users such as editing user profiles among others.

The loopback path immediately following the **LoginService** is there for the cases where login fails; there is no other path than to log in again or abort.

Based on the login credentials the user will then either be catalogued as a moderator or a regular user, both of which can still access both the **SubmissionService** and the **UserService** depending on their intention.

Actions from the **UserService** will not pass through automated moderation and therefore are forwarded straight to the **DatabaseServer**, whereas some actions happening in the **SubmissionService** (such as submission and editing of requests) need to be routed to the AutomatedModerationFilter and only then to the **DatabaseServer**.

As per our assumption 3.2. Concurrent user sessions we will have a max capacity of 100 sessions which is limited in the capacity of the Application Server; this is a problem we will not have with our current user load.

	Application Server	*Login Service*	*Submission Service*	*User Service*	*Automatic Moderation Filter*	*Database Server*
Service time User	0.5	0.05	0.8	0.6	0.4	0.45
Service time Moderator	0.5	0.05	0.8	0.6	0.4	0.45

Table 1. Service times for each service center.

The service time is similar for both user groups as all actions are agnostic and not dependent on the action being triggered by the user or moderator.

Router 1	User	Moderator
Submission Service	0.9	0.5
User Service	0.1	0.5

Table 2. Probabilistic routing for Router 1.

Router 2	User	Moderator
Automatic Moderation Filter	0.3	0.05
Database Server	0.7	0.95

Table 3. Probabilistic routing for Router 2.

For **SubmissionService** and **UserService** there is approximately 90% of the users hitting submission more often and 10% for the user service; this is because users have far more business with submissions as it's what the application is for and almost all they can do with it, unlike moderators who might access UserService as part of their duties (users would only view/edit profiles here). While for moderators there is a more balanced flow of 50% between each as they have to take moderator actions while acting in the capacity of a moderator. See table 2.

Per Table 3 the user browsing submissions without posting a new submission is simulated to be approximately 70%. In this case, that means 30% of user requests hit the moderation filter, as only this portion leads to new submissions. Any requests not going to **AutomatedModerationFilter** hit the DatabaseServer; see Figure 2 for a visual explanation. Moderators, due to their moderation duties, on

average perform more listing of submissions than posting thereof. In their case, there is a likelihood that a moderator posts a submission of around 5%, where 95% is spent on submission queries that directly hit the database.

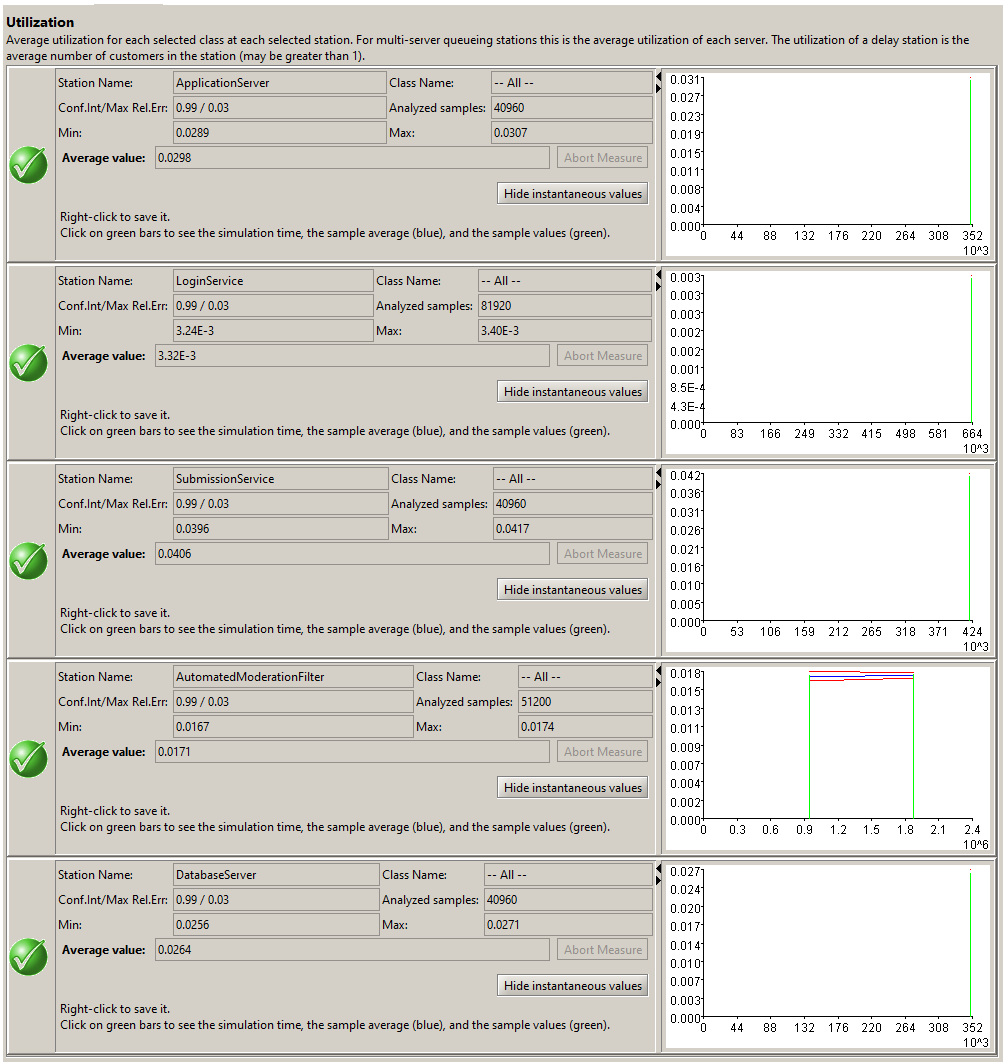

Figure 6. Utilization of all service centers.

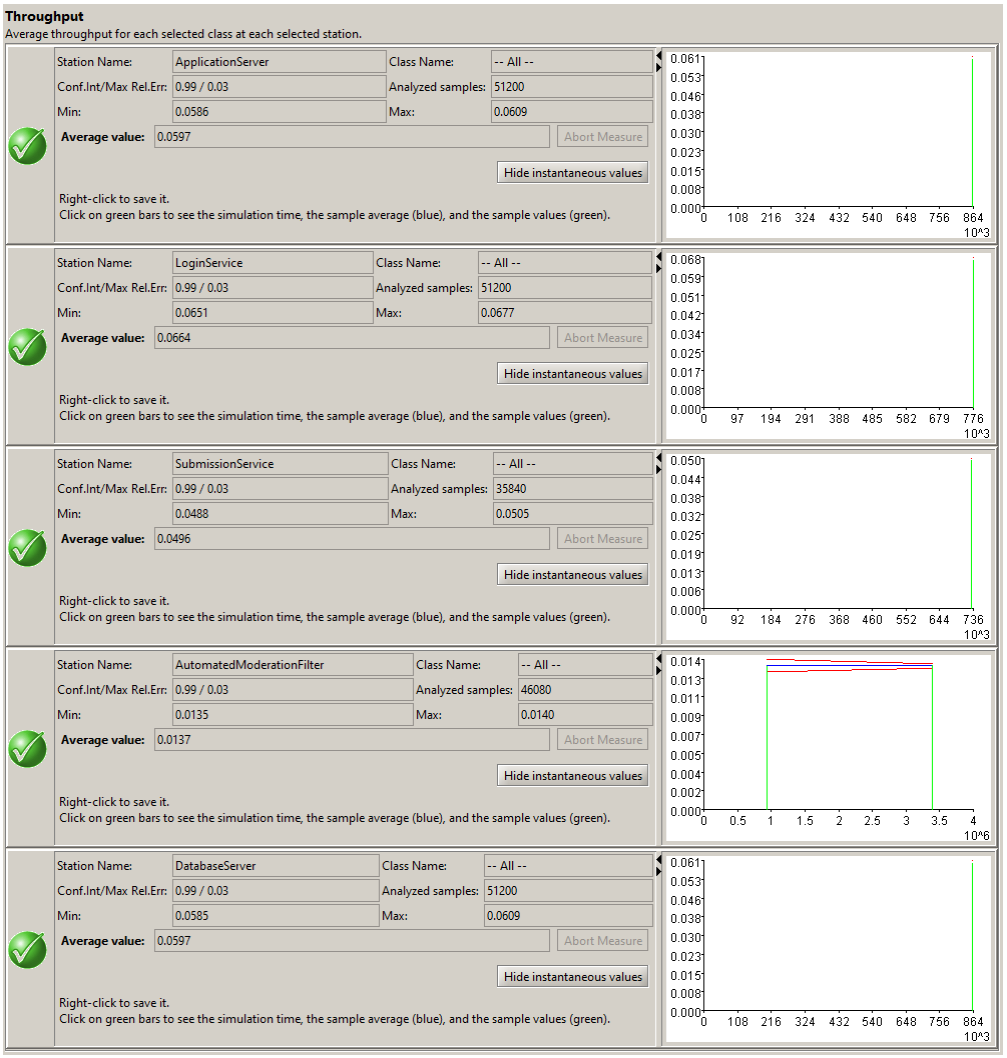

Figure 7. Throughput of all service centers.

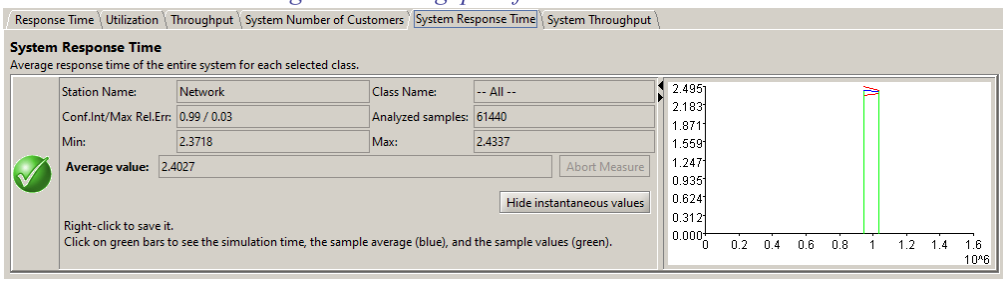

Figure 8. System Response Time.

Our simulation lines up with actual expectations that our app is not heavily utilized and there is room for the application to scale for a long time. As per section 3.2. Concurrent user sessions, our current throughput is 0.06 which is nowhere close to our theoretical limit as per the simulation.

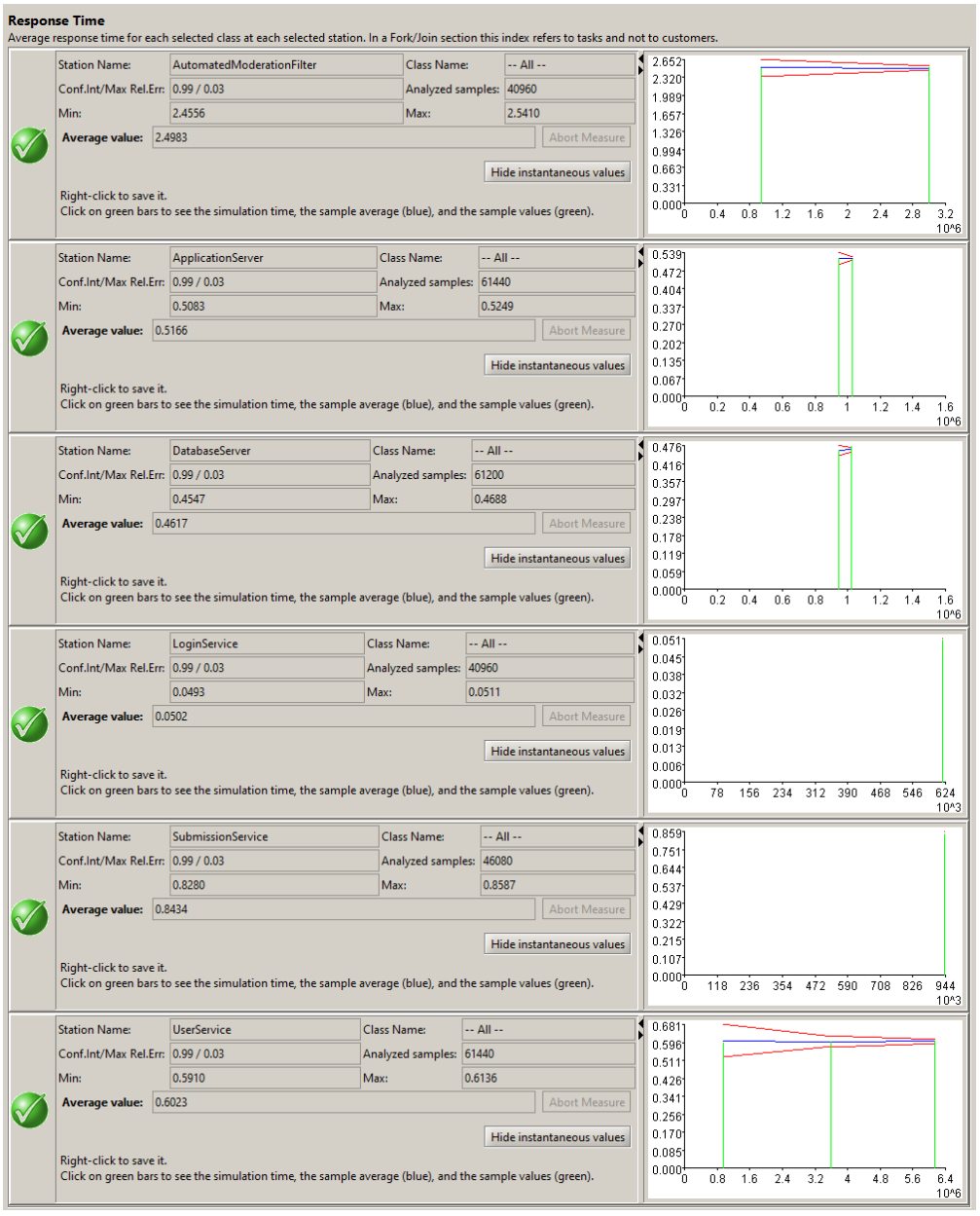

Figure 9. Component response times.

The largest offender of service time is our AutomatedModerationFilter, but as this deals with moderating posts and handling the logic to detect malicious posts, it is of no surprise that this is the heaviest component of our system.

The NFR for this is also fulfilled as per MOD-010 (The automated moderation filter should not take more than 3 seconds to process a post). In our simulations our response time averaged 2.5 seconds which is well within the constraints of the NFR; see figure 9.

Per NFR BKN-008 the system response time should be kept to a minimum of 3 seconds; this non-functional requirement is also satisfied by our system response time of average 2.42 seconds, see table 4.

NFR BKN-009: The login request should be processed in less than 0.5 seconds. In our simulations, the response time for the login service averaged around 0.05 seconds which is well within parameters. This satisfies the non-functional requirement, see table 4.

In addition, our database is constrained by NFR BKN-010 (The database request should be processed in less than 0.5 seconds) which is satisfied by our current simulation results of around 0.46 seconds, see table 4.

See table 4 for a summary of the utilization, throughput and system response time and service time.

Service	Response time	Utilization	Throughput
System	2.4027	N/A	N/A
ApplicationServer	0.5166	0.0298	0.0597
LoginService	0.00502	0.000332	0.0664
SubmissionService	0.8434	0.0406	0.0496
AutomatedModerationFilter	2.4983	0.0171	0.0137
DatabaseServer	0.4617	0.0264	0.0597

Table 4. JMT statistics

To conclude, all four of our NFRs were within the constraints. The constraints were not close to hitting any limits except MOD-010 (The automated moderation filter should not take more than 3 seconds to process a post), which is on average 2.5 seconds with a max constraint set to 3 seconds, which must be taken into account when developing the application and building upon it, BKN-008 (the system response time should be kept to a minimum of 3 seconds) is close the constraint and would also need planning when designing around the system and adding new components.

NFR	Max value	Actual value (average)
MOD-010	3 seconds	2.4983 seconds
BKN-008	3 seconds	2.4027 seconds
BKN-009	0.5 seconds	0.00502 seconds
BKN-010	0.5 seconds	0.4617 seconds

Table 5. Non-Functional requirements response time

Appendix – Time Report

Date	Member	Activity	Time (hours)
2022/04/24	Kateryna	Writing Section 2. Design philosophy and principles	1h 30m
2022/04/25	Richard	Suggestions and additions to section 2	1h
2022/04/25	Kateryna	Writing Section 2. Design goals and some changes to design principles	2h
2022/04/25	Alija	Writing section 5 and draft section 2.	30m
2022/04/25	Albert, Alija, Kateryna, Richard	Group meeting	30m
2022/04/26	Alija	Work on section 3.	30m
2022/04/26	Kateryna	Writing section 2, critical issues	30m
2022/04/27	Albert	Writing section 4	1h
2022/04/27	Albert	Reviewed section 4	15m
2022/04/27	Richard	Section 7.1 and research for 7.2	4h
2022/04/27	Kateryna	Writing section 1, planning to revise design philosophy in section 2	1h
2022/04/27	Albert	Added requirement descriptions to 4	15m
2022/04/27	Alija	Working on 5.2 & 3.2 & 3.3.	2h
2022/04/28	Alija	Finished 3 and 5 sections.	1h30m
2022/04/28	Kateryna	Reviewing section 3 and 4	30m
2022/04/29	Alija	Processing feedback for section 3 and	15m

		leaving some minor suggestions.	
2022/04/29	Atakan	Meeting Section 9	1h
2022/04/29	Kateryna	Meeting Section 9	1h25m
2022/04/29	Richard	Meeting Section 9	2h50m
2022/04/29	Alija	Meeting Section 9	2h50m
2022/04/29	Albert	Meeting Section 9	1h
2022/05/01	Kateryna	Preliminary Component diagram modelling and motivation	4h
2022/05/01	Richard	Rewriting 7.1 and adding 7.2	4h
2022/05/01	Alija	Adding 5.5 and 5.6.	1h30m
2022/05/01	Albert	Reacting to feedback	15m
2022/05/02	Albert	Opining on 7	15m
2022/05/02	Alija, Richard	JMT finishing up section	55m
2022/05/02	Kateryna	Updates to the component diagram and its description	50m
2022/05/02	Albert	Opining on 9, creating sequence diagram	2h20m
2022/05/02	Kateryna	Preliminary Deployment diagram modelling and motivation	2h
2022/05/02	Kateryna	Review of section 9	45m
2022/05/02	Alija	Revise section 5.6	10m
2022/05/02	Albert	Discussions/reviews	30m
2022/05/02	Atakan	Planning section 6	45m
2022/05/03	Alija	Revise section 5.1	5m
2022/05/03	Albert	New version of sequence diagram	1h
2022/05/03	Albert	Language improvements in 1-5, listening to diagram feedback	45m
2022/05/03	Alija, Kateryna, Albert	Meeting regarding document	35m

2022/05/04	Alija	Revising section 9.	2h
2022/05/04	Albert	New version of sequence diagram	45m
2022/05/04	Richard	Rvising section 9	1h
2022/05/04	Albert	Minor diagram update, commenting diagram, language improvements in 8-9	45m
2022/05/04	Albert	Added sequence diagram for editing profile	15m
2022/05/04	Alija	Working on JMT revising	1h
2022/05/04	Kateryna	Reviewing document	45m
2022/05/04	Atakan	Planning and writing section 6	1h30m

Table 6.

References

1. https://martinfowler.com/bliki/Yagni.html
2. https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle
3. Kouraklis, J. (2016, October). MVVM as Design Pattern. *MVVM in Delphi*. (PDF) MVVM as Design Pattern
4. Siddiqi, M. S., & Vice, R. (2012). *MVVM Survival Guide for Enterprise Architectures in Silverlight and Wpf*. Packt Publishing.