2DV609

WeCare

Design Document

Author1: Mohammadali Rashidfarokhi
Author2: Khalil Mardini
Author3: Nima Safavi
Author4: Amirhossein Soltaninejad
Author5: Dominik Pappe

Contents

1. Purpose

This document provides detailed information on how the "Wecare" system will be designed. In other words, this document will explain and simplify the critical factors that were considered in the development of the system. In this document, we will share the process and the approach that we used in designing the system. Overall, this document is intended to be a compilation of material that covers all aspects of the "Wecare" system (e.g., diagrams and documentation).

- General information about the “Wecare” system:

To provide a more accurate and rapid diagnosis of Hemophilus paresis, a polymerase chain reaction (PCR) test was developed [1]. It is becoming increasingly difficult for the laboratories to keep up with the fast pace of demand for the PCR test (e.g., Covid 19) because of the increase in demand. Before COVID-19, laboratories used papers and local archives, which meant patients preferred to take their tests at other laboratories to be able to complete the tests faster and to be able to do them online. As a result, the demand for an application that assists patients and the laboratories with PCR and COVID-19 diagnosis is on the rise. Apart from the COVID-19 test that is the focus of this application, other tests have been covered in the application as well. Those tests are known as “HIV”, “pregnancy”, and “Chlamydia”.

2. Design goals and philosophy

The system needs to be flexible, user-friendly, and compatible with any operating system such as iOS, windows, android, and Linux to meet the requirements of the stakeholders. The interface should be simple but smooth so any user can use it. Also from the developing aspect, in case of the need for an update and removing the possible bugs, the implementation should be reusable and maintainable. There might be deployment concerns due to the lack of time for testing all functions of the system, the inability to verify responses in the system, and the failure to observe performance test results.
We do our best to define the goals and success areas for this application to lower the chance of deployment concerns. An adequate number of test cases will be devised to check the performance of the web application.

When implementing the proposed solution, it is imperative to achieve the following goals:

- Scalability

Since the number of users on the platform can vary depending on the interest of the users in a test, it is important to design the application very flexibly so that, for example, we can react to a large rush for PCR tests as soon as the incidences are relatively high without compromising the stability of our system.

- Ease of Use

A feature's usability and user experience must be satisfactory. If the user makes mistakes in the application, he/she should be informed of the errors.

- Extensibility

The system must be designed in such a way that it can be extended by other functions in the future. This enables an increased value for the end-users, as they can use further functions.

- Single Responsibility

The single responsibility principle specifies a single function per class and helps to make code modular to avoid interdependencies. Classes and elements for which there is only one responsibility are easier to explain and understand in a codebase environment.

- Consistency and standards

To make the usability of the applications as easy as possible for the users, we should orientate ourselves towards industry standards. This makes our application more intuitive and outwardly consistent.
Consistency should also be ensured within our application. This means that elements that have the same tasks are also the same in appearance and behavior.

3. Assumptions and dependencies

The system introduced in this document will be a prototype, which will be developed fully with the assumption of having support from the following resources.

- **Budget resources:**

Because our project is a student project and does not benefit from any student grants or specific budgets, this project will be designed and implemented without a budget.

- **Scheduling accuracy:**

The group members believe that the deadlines and milestones are achievable, and they will be delivered on time. There are 3 module deadlines, and the delivery will be done as scheduled.

- **Human resources:**

The project is expected to be completed with 5 members. The members should have the possibility of working on the project 20 hours per week till the regular deadline for delivery of the project.

- **Activity performance**:

The project is expected to be done with a high performance of the group members so each member should be able to document and implement the specific requirements.

Dependencies

Start To Start: The process of implementation will start after the delivery of the first module regarding requirements specifications.

Finish to start: The delivery of module 3 (Software release) will start after finishing the documentation for modules 1 and 2 and the implementation of them according to the requirements which were declared in the documentation.

Design Approach:

The top-down approach is considered to be better to follow in this project because we have already started to break the problem down into smaller manageable parts before any artifacts are implemented. This structure provides an overview of the big picture so that we can plan structural architectural decisions early on. This includes our decision to build a web application or use an MVC design pattern. As the different components of the MVC model can be easily divided up, several people from our team can work on the application at the same time. Further decisions about the subsystem are then made at a more detailed level. Once a satisfactory structure has been achieved through system planning, the implementation of individual functions can begin. Since time is usually a limited resource in software projects, it can take too much time in the beginning due to detailed planning, which means that we should plan enough time for quality control of individual elements of the software.

4. Architecturally significant requirements

Title	Description	Importance
NFR-01 - Performance	The system should be able to respond to the user request in a short period (Less than 3 seconds)	*Performance plays an important role in user satisfaction. The loading time of the start page is decisive for whether the user stays on our page or uses another application instead. Therefore, a compressed data transfer must be used here, as well as images must be displayed in the correct resolution.*
NFR-02 - Availability	The system should be accessible to the users at any time.	*In today's world, services are expected to be available around the clock. Thorough control of bugs, for example, can reduce downtime.*
NFR-03 - Usability	The user interface should be easy for users to work with.	*To make the interface coherent for the user, uniform standards should be used: the same design of buttons, transitions & hover effects. This allows users to anticipate what they can expect from the system and thus find their way around better.*
NFR-04 - Portability	The system should have the adaptability to run on different web browsers.	*To ensure that every user can access our application from anywhere, we have opted for a web application. This way we can offer the possibility to use our application to as large a target group as possible.*
NFR-05 - Security	The user should be able to log in to the system using an SSL certificate.	*The SSL certificate is an important building block of our architecture, as it allows our application to represent trust to the outside world and at the same time also means data security for the users. Especially in the medical field with critical data, this is important to prevent fraud or unwanted attacks*
Maintainability	The system should be easily maintainable in case of possible bugs	*To solve possible bugs more quickly, it is important to implement the application in such a way that it is easy to modify. Possibilities to react faster to bugs are the application of the single responsibility principle as well as consistent use of comments to explain the code.*

5. Decisions, constraints, priorities, and justifications

Web application

- Alternatives: mobile app
- Constraints: With mobile apps, user engagement is higher than with web applications, as it is possible to work with push-up messages.
- Reason: A web application works on all operating systems, with an internet browser, which is why portability is higher and a larger target group can be reached. Maintenance costs are also lower than with a mobile app, as only one product needs to be maintained.

MVC design pattern

- Alternatives: MVVC, MVP
- Constraints: The controller can get complex which can be a problem
- Reason: The advantage of the distribution of tasks in a model view controller architecture is, on the one hand, the division into logical, independent classes. Thus, a model view controller architecture enables agile development, as different people can work independently on one of the 3 components

Using C# and .NET

- Alternatives: Java, Python
- Constraints: Dependence on the framework
- Reason: The application will be implemented in C# and .NET because we have previous experience with these languages. Therefore, we can significantly reduce the development workload, because it’s easier to use and no training is needed.

MySQL

- Alternatives: No-SQL (e.g., MongoDB)
- Constraints: limited flexibility, prior creation of a DB schema required
- Reason: We use a MySQL database, as we are expecting no huge amounts of data in our database. Therefore, the data can be stored without much effort with a tabular modelled database schema. The ACID principle can also be used to ensure that data is consistent throughout.

No Cloud used

- Alternatives: Cloud Server
- Constraints: Less scalability, no pay per use
- Reason: Having our server gives us more flexibility and is also a cost-effective way to host a project.

Priorities:

The design priorities will follow the characteristics of Software Quality according to Steve McConnell [1]:

Reliability: The software provides combined functionality for the specific task and user objectives.

Usability: Easy to learn and easy to operate.

Correctness: A system's ability to perform the required functions under specified conditions when needed.

Integrity: Information and data protection, so that unauthorized personnel or system cannot read this information and data without refusing or accessing authorized personnel.

Adaptability: The program does not need to adapt activities or methods that are not intended for the purpose to consider the program and may be adapted to different specific environments.

Robustness: As a result of invalid inputs or stressful environmental conditions, a system is expected to continue to function

Maintainability: Software's ability to be changed, expanded, improved, or corrected easily through modification

Accuracy: The amount of error that can be introduced by a system, especially in terms of its quantitative output.

6. Key abstractions

The key abstractions for the WeCare system design that are derived from the main functionalities are as follows:

- **Patients**

One of the key abstracts of the system is the patient, who is considered as the main user of the system.

- **Accounts**

The second part of the key abstractions through which all parts of the system are accessible, and all system users are required to use is the accounts in the system.

- **Authentication**

Another key abstraction is the authentication system. Authentication plays a key role in the system in two ways. On the one hand, in terms of ensuring system security and preventing possible sabotage, and on the other hand, in order to provide proper services to users according to their role and their need for the system.

- **Doctors**

One of the other key abstracts of the system is the doctors of the laboratory, who are considered as one of the main users of the system.

- **Admin**

The next user key abstraction of the system is the administrator whose major role is to define the users for the system.

- **Booking Service**

The next key abstraction is based on the main service of the application. The booking service is the most important part of the system and one of the most important key abstractions that must be carefully planned, so there will be no two tests taken simultaneously, otherwise, there will be chaos and an overload of work for doctors that may lead to a reduction in customers' satisfaction due to longer waiting time.

- **Database**

Another key abstraction is the database. The database should be designed so that it can store critical information correctly and accurately and retrieve it completely without any loss of data when needed.

7. Architectural Patterns

The architecture of our application is implemented with a client-server model. This allows for easy distribution of services and tasks. Numerous services are provided by our server and can be accessed by our clients. Three types of clients can be distinguished, which are assigned different tasks: Patients, doctors, and admin.

1. Patients will be able to book an appointment for a test and receive a result after evaluation. Also, they can obtain a generated PDF containing the test results along with the QR (Quick Response) Code.

2. Doctors are responsible for the evaluation of the tests. Also, the publication of the test result is one of the main responsibilities of the doctors.

3. Admin: The admin of the "Wecare" application has the duty of managing the application. That is, he has the responsibility of creating roles, updating the rules of the users, and creating new users in the application. Also, an admin can remove a user from the application. To sum up, critical activities and decisions for the "Wecare" application require the admin, Clearance.

The "WeCare" application follows the MVC model. Moving on, by separating views from models and controllers, you can make MVC most effective. Having a separate display from the data allows you to modify one without affecting the other. Additionally, the MVC pattern can lead to code reuse and parallel development. Hence, working becomes simpler. Using the MVC pattern, components are created that are independent of each other. Lastly, Since the "Wecare" application is using .NET in the implementation phase, MVC becomes supportive in the manners below:

1. **Code/ program Maintenance:**

MVC allows you to change the way you access the database without affecting the user interface, something that seems impossible without it.

2. **Low coupling:**

All business logic is centralized into a controller, comprised of high-performance modules. In the event of a change in the business logic, the view and model are not required to be updated, only the controller needs to be changed.

3. **Security:**

The user will never be able to see or check the code in the Controller so whatever is in the Controller classes is safe. The built-in security functions and protocols in NET are enormously powerful and effective. And a huge package manager "NUGet" which has everything you need.

Note about NuGet: An open-source package manager called NuGet allows developers to share reusable code. This is a free and open-source software as a service solution.

4. **Abstraction:**

It is easy to find errors in the code in the shortest amount of time, and the code and logic are clearly divided, improving development efficiency.

- **Design Principle 1: Divide and conquer**

Isolating the system into client and server processes is an extremely impressive method for partitioning the system. Each can be independently evolved. Inside the abstract server-client architecture, the three components of the MVC-model can be independently designed.

- **Design Principle 2: Increase cohesion where possible**

The separation of the view and controller components leads to an increase in cohesion compared to a merged single UI (User Interface).

- **Design Principle 3: Reduce coupling**

The MVC pattern allows you to create applications that separate the various aspects of the application while providing a loose coupling between these elements. This separation helps in managing the complexity when building an application by allowing you to focus on one aspect of the implementation at a time.

- **Design Principle 4: Keep the level of abstraction as high as possible**

Separate dispersed parts (server & client) are in many cases great abstractions. For instance, you do not have to comprehend how a server works.

- **Design Principle 5: Increase reusability where possible**

If our MVC components are built according to the single responsibility principle, it is possible to reuse them in other projects. Although the view components are explicitly designed for our project, and it will be hard to reuse then in other projects without the need to change the design.

- **Design Principle 6: Design for flexibility**

Since the system is designed to be open to changes and extensions, using the MVC pattern provides an easy way to customize the UI by changing the view but keeping the logic behind it the same.

- **Design Principle 7: Design for Portability**

Having the portability feature within the product (WeCare app) is significant since it will allow the product to be utilized in different web browsers without having the need of changing the system design. So, it is essential to code the three components of the MVC (Model, View, and Controller) in a way that will be adaptable for the different web browser environments.

- **Design Principle 8: Design for Testability**

In the MVC pattern, the model is independent and not dependent on the controller & view. Since the model has no dependency on other components, this part can be tested excellently with a unit test. Unit tests are often used for testing business logic. Therefore, we can automate them to test the application separately from the UI.

- **Design Principle 9: Design defensively**

It will primarily concentrate on decreasing the number of human errors and various common risks through designing backup plans and anticipating the potential risks that might occur during designing the MVC components.

8. Architectural views

- **Component Diagram**

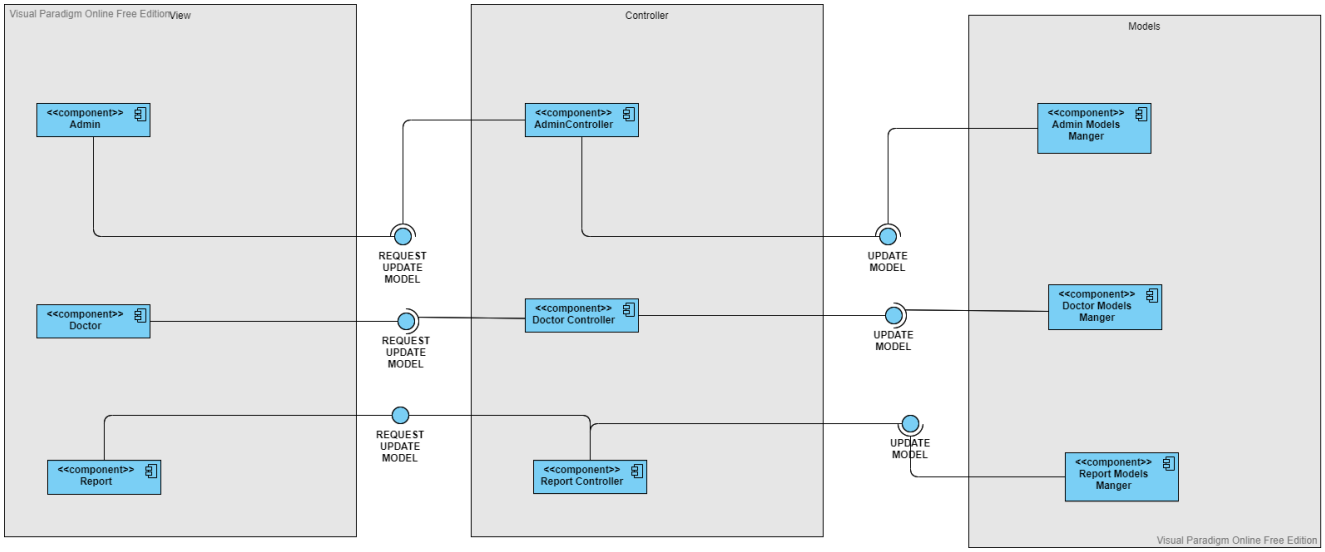


Figure 1, component diagram

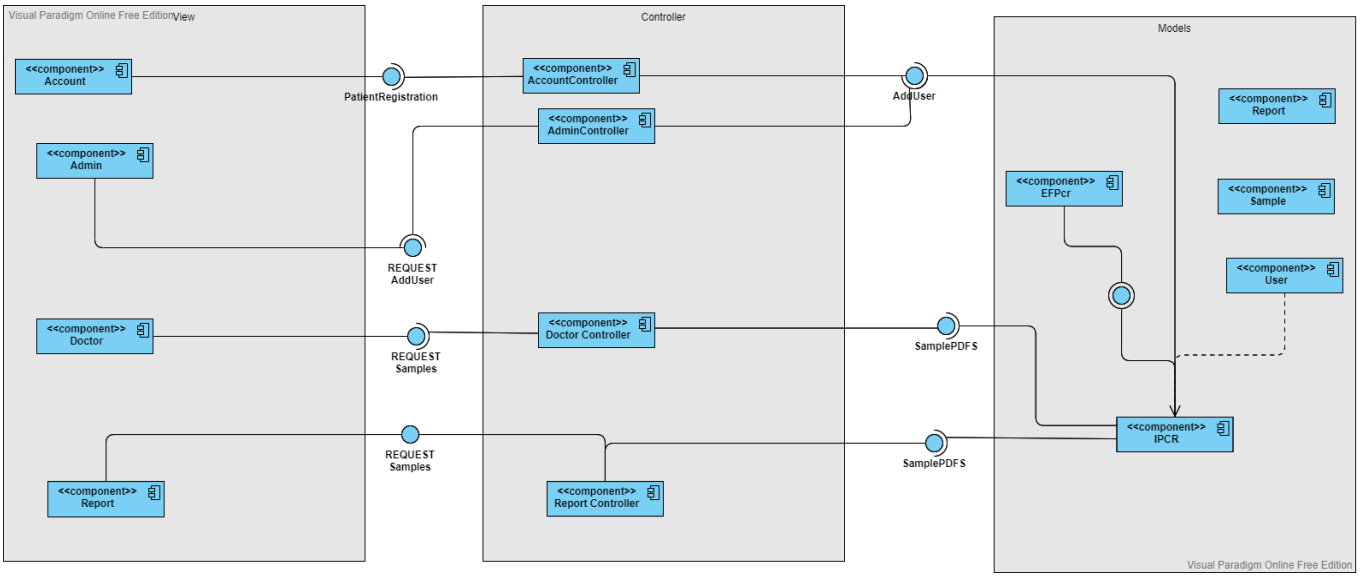


Figure 2, Component Diagram

Our component diagram will illustrate how the MVC components handle certain development flows of the WeCare application. The view component will manage the UI logic and appearance structure of the WeCare app such as the patient view (dropdown, images, buttons, etc.).
The model component is based on the logic-related data in which the user deals, especially the data being transferred between the controller and the view components. Finally, the controller component is essential since it establishes an interface between the view and the Model components which handles the incoming request between the client-side and the admin side.

- **Sequence Diagram**

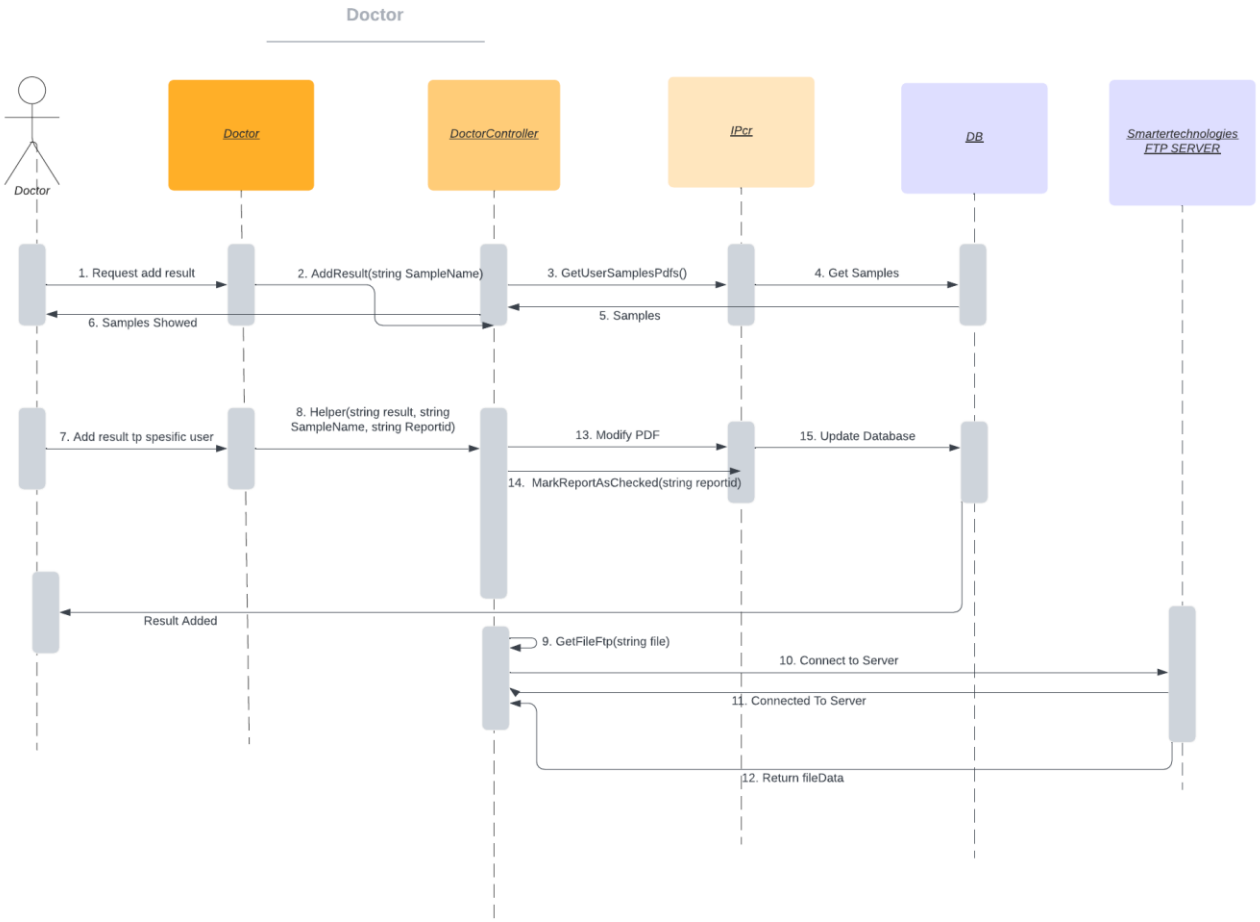


Figure 3, Doctor Sequence diagram

The sequence diagram above illustrates the sequencing process that was achieved by the doctor which includes adding test results to the database system. Furthermore, the doctor will determine the test result for a specific user (positive and negative) depending on the type of test chosen and insert it into the pdf report and then store it within the database system of the WeCare application.

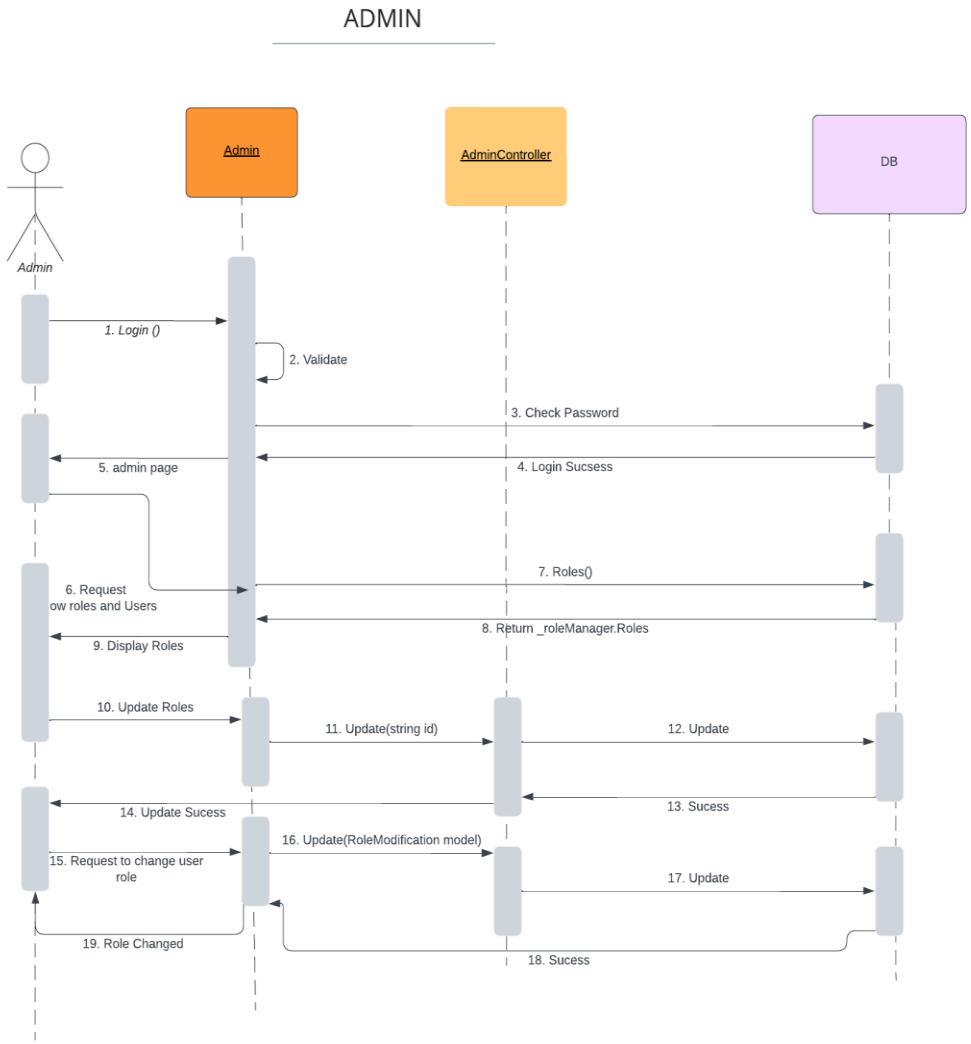


Figure 4, Admin Sequence diagram

The sequence diagram above illustrates the sequencing process that is achieved by the admin first it starts by logging in into the “WeCare” system as an admin, secondly, the admin can access the roles of the users within the system by checking each single user role through retrieving the data from the “WeCare” database server. As a final point, the admin has the privilege of modifying the roles of the users by adding deleting and assigning new roles to the users (such as prompting the doctor to be an admin in the “WeCare” system and vice versa)

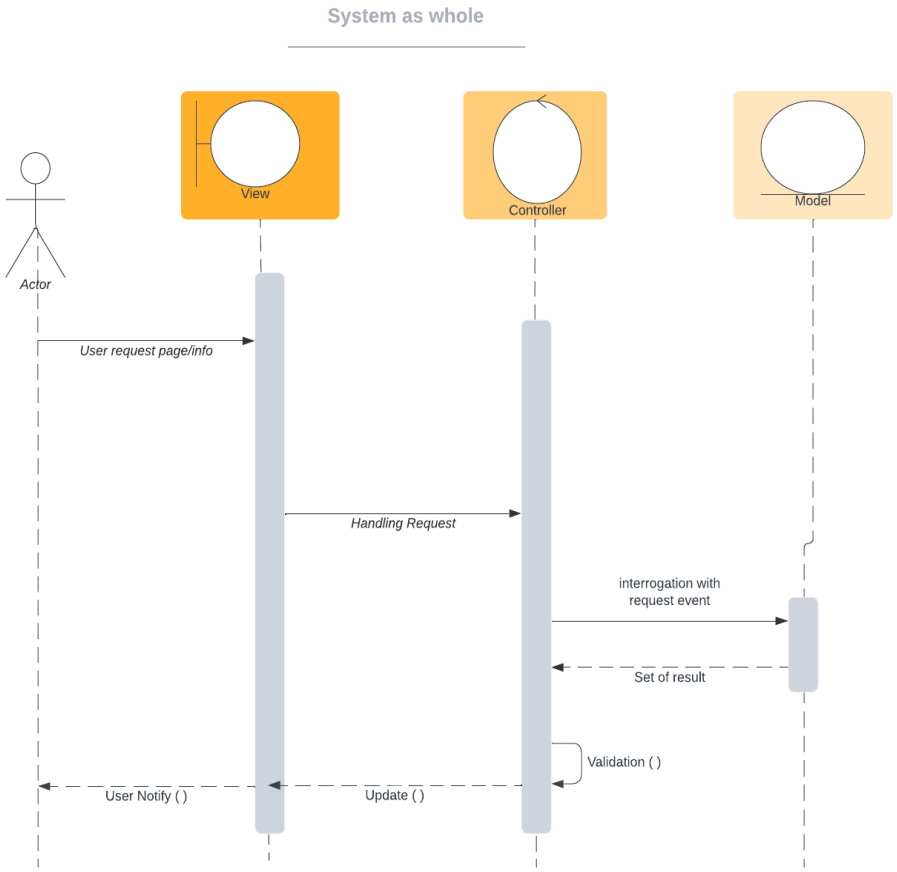


Figure 5, MVC Sequence diagram

The sequence diagram above displays the sequencing process for the WeCare system as a whole, it starts by initiating a request to the view model to access the page, then the controller model will handle the request from the view model which will consequently send it to the model part that will take care of the integration with the request event.

- **Deployment Diagram**

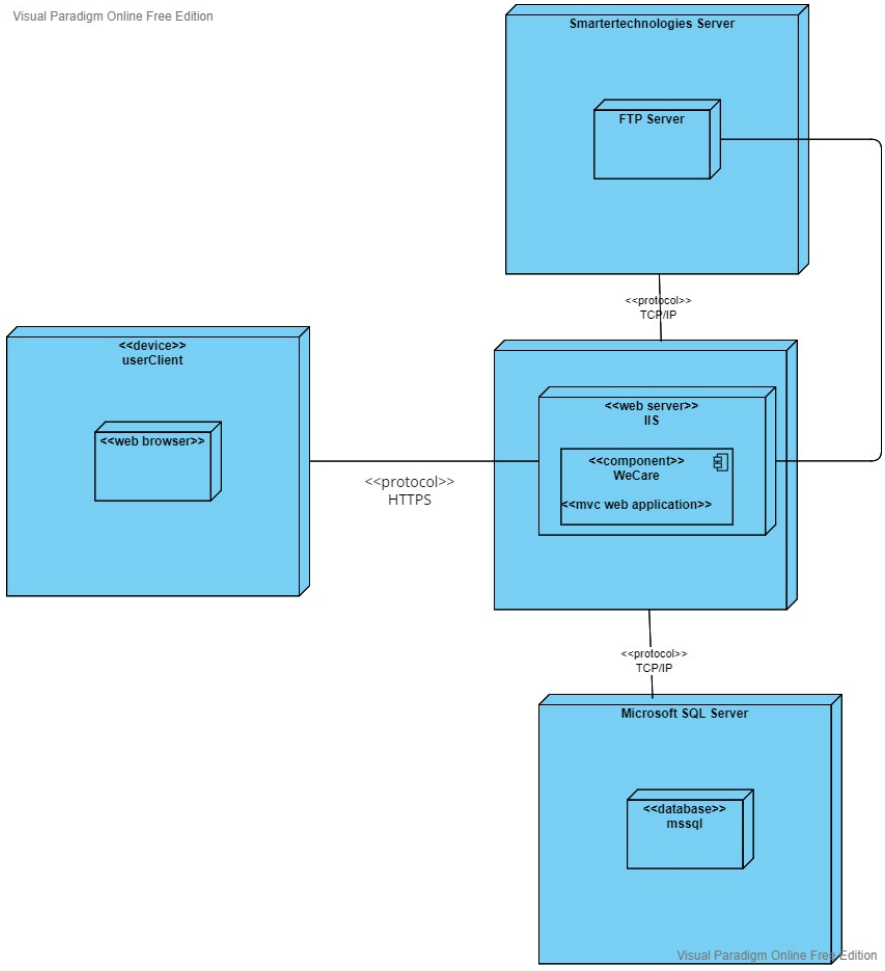


Figure 6, Deployment Diagram

The above deployment diagram represents the execution architecture of the "WeCare" application and displays the relationship of the software components. The system uses an FTP server, and it communicates with the TCP/IP protocol. Moreover, the webserver will communicate with the SQL server database through TCP/IP protocol. In order to increase security, the web server uses HTTPS protocol to communicate with the user client.

9. Performance modelling and analysis

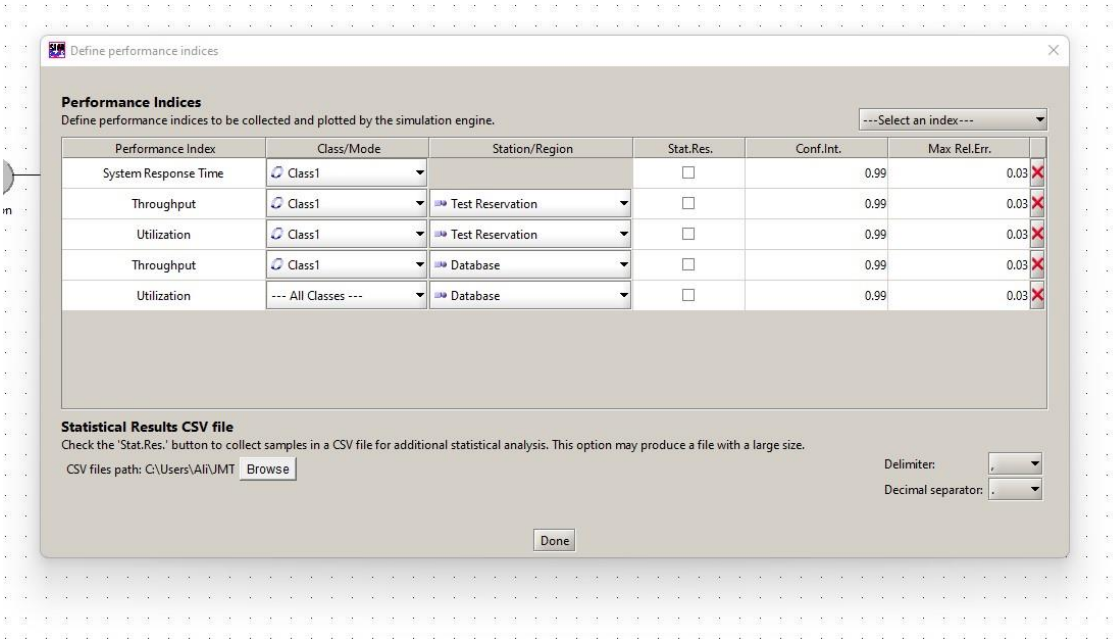
Before modeling, we explored the possibility that a patient could reserve a test through the "Wecare" system. We incorporated two component parts: the database and the test reservation. Finally, the JMT performance model can be found in this section.

We have considered observing the system for 2 hours (7200 sec) and with the arrival rate of 2 requests per second. The service time that is applied on the test reservation server was 0.1 sec and the database has a service time of 0.4 sec. So, by running the simulation we have observed that the system response time is 2.0809 and it can be improved by improving the database response time or adding additional servers to improve the throughput of the database system. We have observed that the database server capacity is almost saturated, and it is good to consider improving the service time of the database in later periods when we are facing a greater number of users of the system.

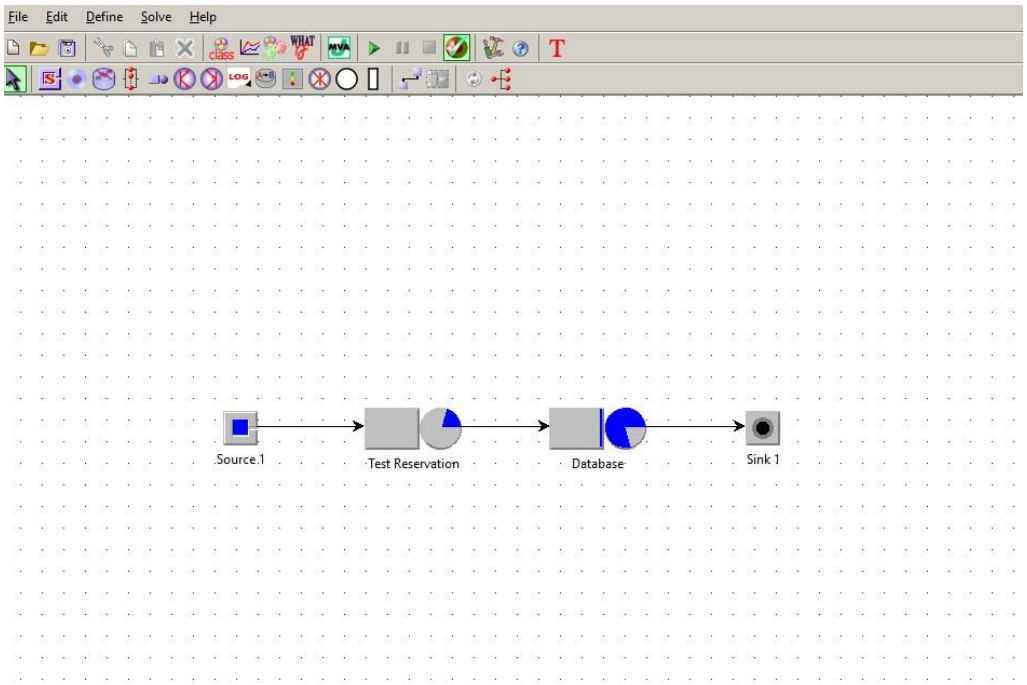
The utilization of the servers is considered good because the values are less than one and for the Test Reservation server it is 0.2030 and for the Database, it is 0.8096. The servers almost have a stable throughput rate with an average of 2 seconds.

In conclusion, the system can respond to the users at a good speed rate, but the Database station service time needs to be improved at later development or adding an extra helping server for the database because the more users using the application. The database section will face the problem of responding to the system.

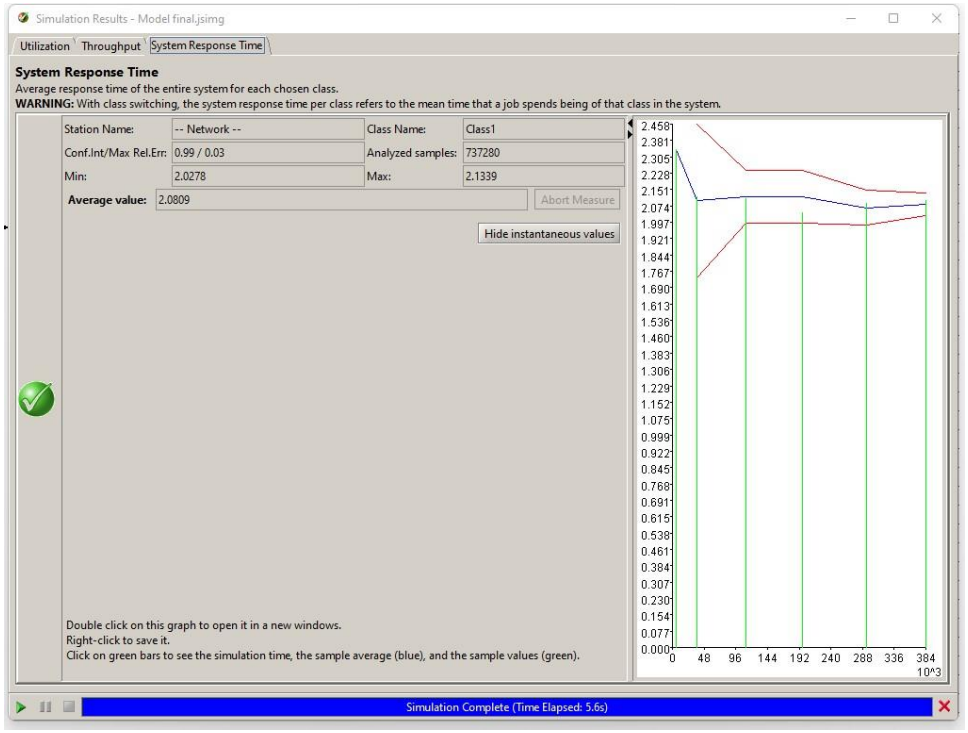
- **Performance indices**



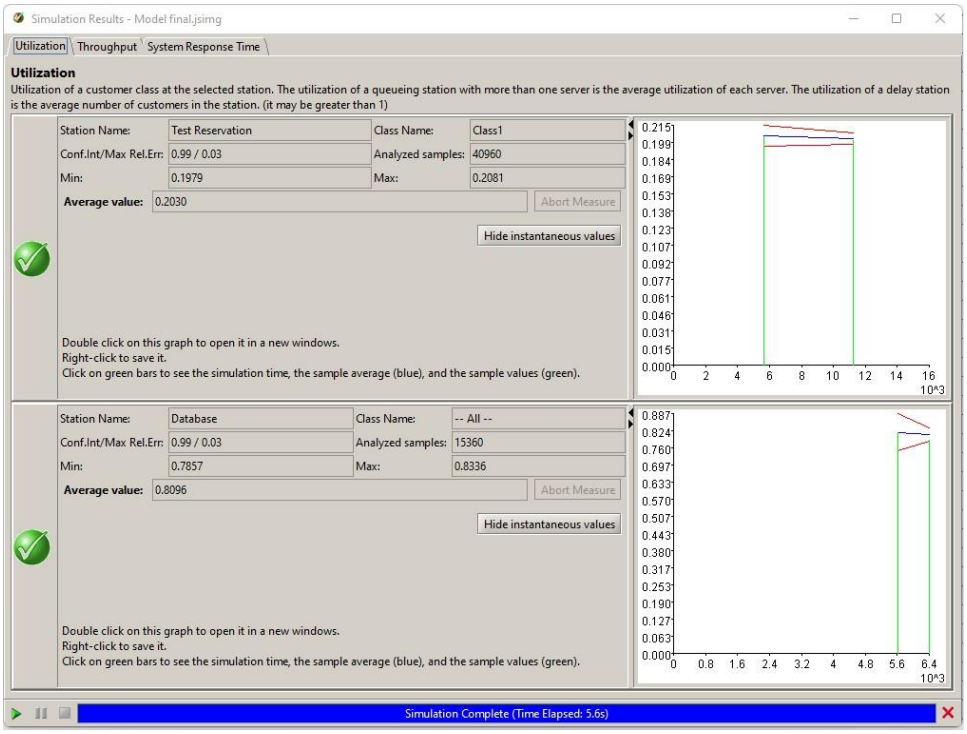
- **Modelled system**



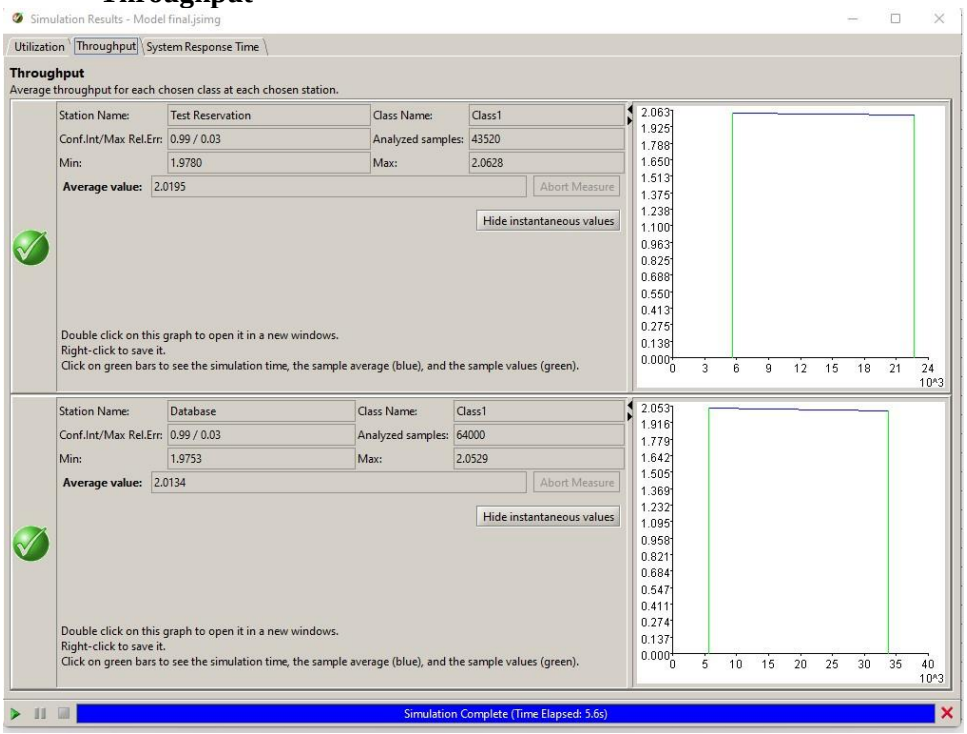
- **System Response Time**



- **Utilization**



- **Throughput**



Appendix – Time Report

Date	Member	Activity	Reviewed By	Time (hours)
2022/05/03	Khalil Mardini	Writing section 3	Nima Safavi	2
2022/05/03	Mohammadali Rashidfarokhi	Writing Section 1	Khalil Mardini	2
2022/05/03	Mohammadali Rashidfarokhi	Writing Section 5	Nima Safavi	4
2022/05/01	Amirhossein Soltaninejad	Writing Section 4	Mohammadali Rashidfarokh	5
2022/05/03	Amirhossein Soltaninejad	Writing Section 6	Mohammadali Rashidfarokh	3
2022/05/03	Dominik Pappe	Writing Section 3	Mohammadali Rashidfarokh	3
2022/05/03	Dominik Pappe	Writing Section 2	Khalil Mardini	2
2022/05/03	Nima Safavi	Writing section 2 & 6	Amirhossein Soltaninejad	3
2022/05/04	Mohammadali Rashidfarokhi	Writing Section 9	Dominik Pappe	5
2022/05/04	Khalil Mardini	Writing Section 7	Amirhossein Soltaninejad	3
2022/05/04	Dominik Pappe	Writing Section 5	Nima Safavi	2
2022/05/04	Dominik Pappe	Writing Section 7	Khalil Mardini	3
2022/05/04 2022/05/05	Amirhossein Soltaninejad	Writing Section 9	Dominik Pappe	5
2022/05/05	Khalil Mardini	Writing Section 7 &Component diagram	Amirhossein Soltaninejad	3
2022/05/05	Mohammadali Rashidfarokhi	Writing Section 8 &Component diagram	Dominik Pappe	5

Reference

[1] McConnell, Steve. Code Complete: A Practical Handbook of Software Construction. Amsterdam: Pearson Education, 2004.