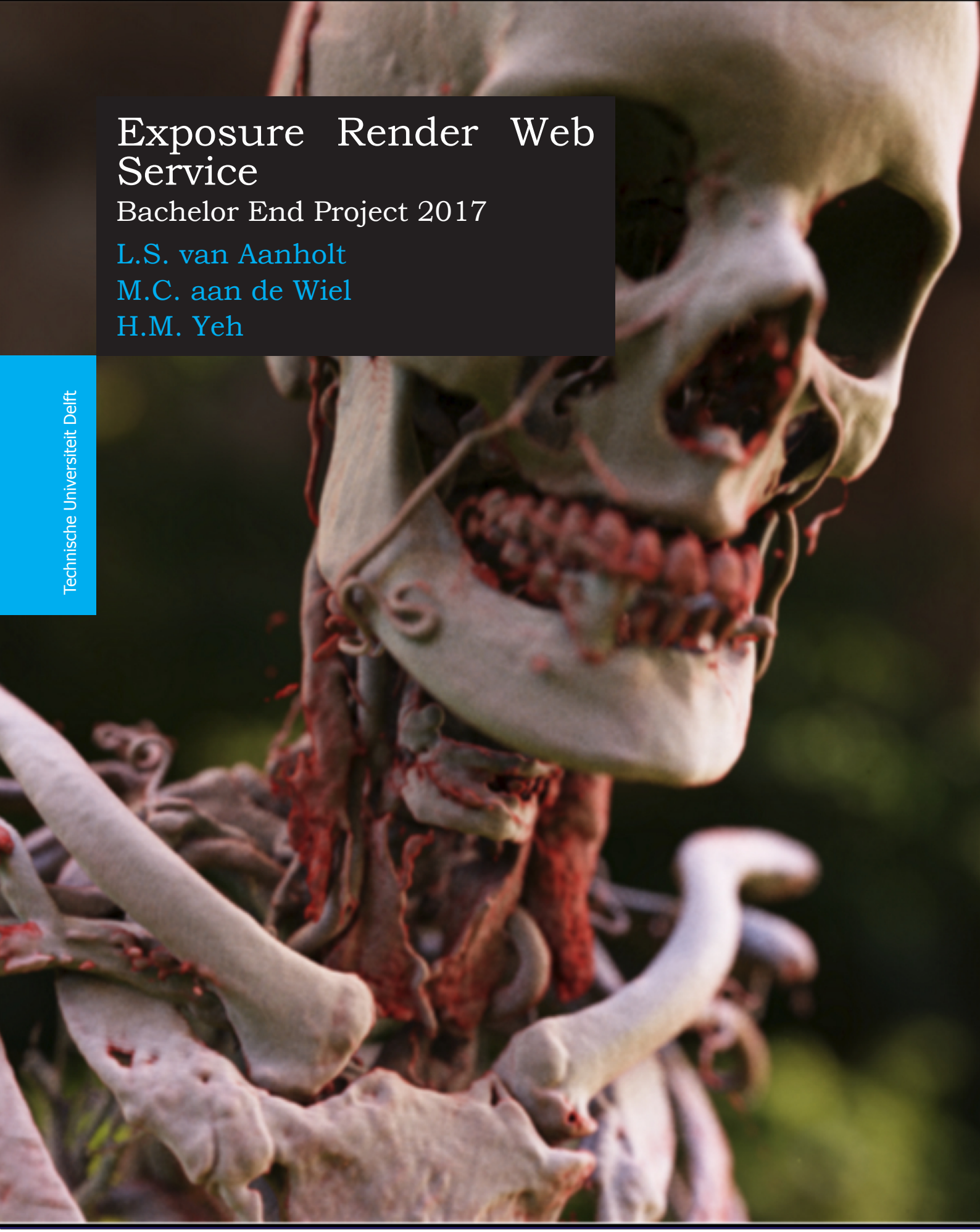
Exposure Render Web Service

Bachelor End Project 2017

L.S. van Aanholt
M.C. aan de Wiel
H.M. Yeh

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

Exposure Render Web Service

Bachelor End Project 2017

by

L.S. van Aanholt
M.C. aan de Wiel
H.M. Yeh

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,

Supervisor: dr. ir. T. Kroes
Client: Prof. dr. ir. E. Eisemann TU Delft

TU Delft Delft University of Technology

Preface

BEP, which is an abbreviation for Bachelor End Project, is the final assignment to complete the bachelor of Computer Science at Technical University Delft. Students work in groups to create a software solution for the client. We have chosen a project that is both challenging and gives the opportunity to further sharpen our skills as professional web developers. The Exposure Render Web Service gave both the freedom to create a good software architecture from scratch and also smartly connect it with the very impressive Exposure Render.

We feel strongly that the product we delivered is of high quality. It answers the requirements of the client while being maintainable, secure and scalable.

The following report is an in-depth document describing our decisions, hurdles, victories and recommendations for the platform in the future. We want to thank dr. Thomas Kroes and prof. dr. Elmar Eisenmann for playing the roles of coach and client respectively and granting us this interesting project.

L.S. van Aanholt
M.C. aan de Wiel
H.M. Yeh
Delft, June 2017

Contents

1

Summary

During the Bachelor End Project we have been tasked with developing a web portal for the Exposure Render. The Exposure Render is able to interactively visualize volumetric data to photo realistic images. This means that the web portal would allow users to view a visualization of their medical data. The requirements of the web portal have been established through interviews with the client and TU coach. During the process of development an agile methodology was used for the division of tasks and iteratively work towards the end product. From the eight sprints a fully functioning prototype has been developed, which satisfies all the main requirements that had been requested by the client. The prototype is separated into three applications: a web interface for the user, a server for persistent data storage and a container for streaming and controlling the rendered scene by Exposure Render. These applications cooperate to form the requested web portal.

2

Introduction

CT and MRI scanners are widely used in the medical sector. These scanners produce volumetric data (a 3 dimensional data set based on a group of 2 dimensional slice images). Doctors are quite familiar with this technique. However, a patient who does not have a medical background will most likely not understand this data.

The TU Delft Computer Graphics and Visualization Group is currently working on tools to make this volumetric scanner data more understandable to users without a medical background. The Exposure Render is an example of such a tool [1]. It can be used to interactively generate photo realistic images of the volumetric data produced by a CT or MRI scanner.

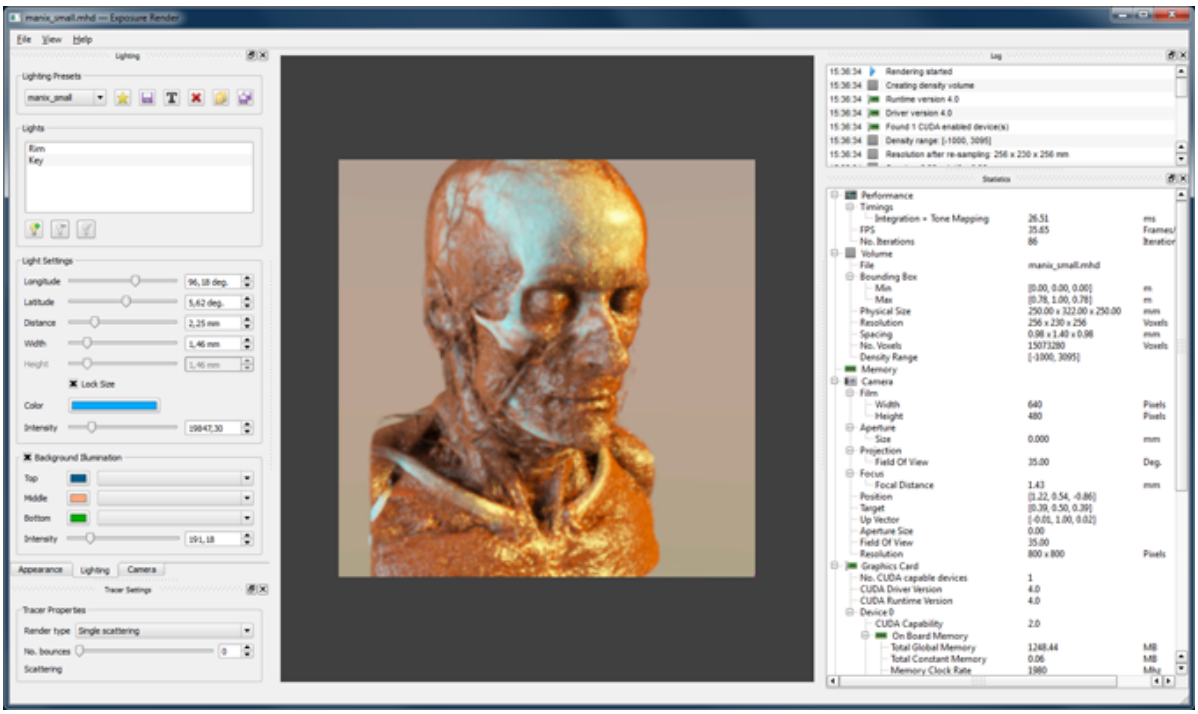


Figure 2.1: Local running Exposure Render application of the open source implementation at: https://github.com/ThomasKroes/exposure-render

The aim of this project is to make a web portal running an instance of the Exposure Render in order to make this tool available to everyone. This portal allows the user to upload their own volumetric data and visualize this without the need of additional hard- or software.

[1] https://github.com/ThomasKroes/exposure-render

The report will be a detailed guide about the web portal, what is created, how it works and our recommendations for the future. In chapter 3 the requirements of the client will be described and explained in user stories so they can be implemented piecemeal. Chapter 4 describes our process, what management methodologies were used and how tools like version control and continuous integration were used. In chapter 5 the complete design of the system will be explained. Interestingly a system has been developed in 2014, called RemoteRender, that also aimed to make the Exposure Render available from the browser, but does not fit all requirements of the client. The code was made accessible to us and in the section 5.2 an assessment of the previous project is made for use in the future. Chapter 6 describes the implementation of the product. Multiple software packages have been developed to fulfill the requirements. Lines of communications, underlying technologies and the mapping of design to implementation will be made clear. Some topics important to the client, like performant streaming and security of handling sensitive data will be clarified in more detail. In chapter 8 we will reflect on the product that has been delivered. Does it solve the requirements of the client? Is it maintainable and scalable for developers and easy to use? Then recommendations will be brought forth. If we continue development of the project what functional and non-functional requirements could be worked on. Advice for further development will be shared and interesting ideas for future projects with the service. Finally in chapter 9 the report is concluded.

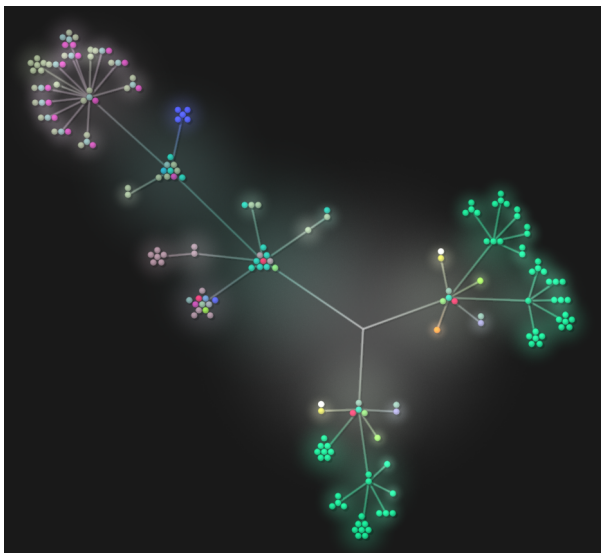


Figure 2.2: Visualization with Gource close to the end of the project of Exposure Render Web Service with the three main repositories: Portal (left), Server(right) and Connector (Below)

3

Problem definition and analysis

The main goal of the Exposure Render Web Service is to let users upload their own MRI and CT scan data and to see their scans rendered as a volume in their browser. The frontend needs to be user-centric, while the backend must handle data securely, save the data and be scalable.

An interesting detail is that a similar platform has already been built in 2014, called the RemoteRender. It used an adapted version of Exposure Render with a Python server and jQuery frontend. To not redo work already done, an in-depth assessment about this previous iteration of the service was necessary in order to see whether this project should be expanded upon or if a whole new platform should be setup. Optionally there could be some elements then worth bringing to our new platform. We have analyzed the previous platform, what worked and what did not to be precise. A detailed assessment is given in chapter 5 of RemoteRender.

The following requirements have been acquired after interviewing the client. The minimal/main requirements for the Exposure Render Web Service are the following (must have):

- User interface (UI) for uploading data
- Processing of uploaded data by the server to a format, readable by Exposure Render
- Viewing the results of the Exposure Render in the browser
- Secure user and data management
- Port Exposure Render to Linux

Additional requirements (should have):

- Multiple containerized Exposure Render instances can run on one GPU
- Easy to use tools to transform the data representation

Non prioritized requirements (could have):

- Scaling out on a GPU cluster

The service needs to have a user-centric interface. This means the user, which is anyone with MRI data, should be able to easily use the features of the system. The user will interface solely with the service through a browser application. There are a number of controls that can be used to change how the scan is rendered on screen. There are self evident ones, like changing the camera position or rotation. Also important is cutting the scan so the user can see inside, which is called clipping and changing the transfer function. The transfer functions in the visualization of scan data is used to show the data in different medically interesting ways. Scan data consists of voxel information with different values for the Hounsfield absorption [1]. The bones have a higher absorption than flesh for example. By assigning RGBA information to a voxel with a certain absorption, meaning giving it color and opacity, different elements of the body can be shown differently or made invisible in the view. By changing the transfer function the user can for example only show the bones or the flesh.

Security is a very important requirement for the system. This means correct authentication of users, authorization of which scans to view and safe from man in the middle attacks.

Scalability is also a big concern, meaning if the demand increases more hardware can be added to serve the users without additional programming needed. The main bottleneck for scaling would be the Exposure Render, because it can only serve one scan. Exposure Render needs to be easy to deploy on new machines to share the load. Therefore Exposure Render needs to be containerized to make automatic deployment possible. Exposure Render was written for usage in windows only, so porting work needs to be done to make it run in Linux so it can be containerized by the defacto standard in container technology, Docker.

We also specify non-functional requirements. These are guidelines to maintain control over the quality of the project.

- Tests as runnable documentation of the code.
- Service should be written with maintainability and valuable for future development.
- Adequate amount of tests covering added functionality and line coverage above 75%
- Continuous integration during development (Unit Tests and integration tests of the whole suite)
- The Exposure Render instance through the browser should feel responsive and render quickly.
- Browser GUI usage is self-evident for normal users.

A tree structure of user stories will now be formulated. If these user stories can be answered with runnable code answering the non-functional requirements then the project is complete. The main user stories are:

1. As a patient I want a user-centric web portal where I can upload CT and MRI data and view this data, using only my browser.
2. As a medical professional I want to be able to prepare the scan of the patient in such a way that the patient can easily see different properties.

1. Can be divided in the following stories.

- As a user I want to upload the data through the browser.
- As a user I want to be able to start a scan I have permission and view it in the browser.
- As a user, when I want to view a scan but the service is currently busy with other views, I want to be queued and have the FIFO principle be used to get to view my scan eventually.
- As a user I want to interact with the scan.
  - I want to change the camera rotation and positioning
  - I want to change the clipping of the object
  - I want to change the transfer function so I can see different components of my body. E.g. only my bones or only the flesh around it.

2. Can be divided in the following stories.

- As a medical professional I want a GUI in the browser that I can use to set up the options for the patient to see different interesting properties
- As a medical professional I want this data to be stored so it can be called upon when the patient views the scan.

4

Process

The following chapter describes the efforts to manage the development of the application. The most important aspects of our process are: Communication, planning and version control.

4.1. Communication

We have employed a Scrum[2] process that emphasizes tight lines of communication. But we take liberties with the strict rules of Scrum and make it work for us. Scrum uses daily Scrum and sprint milestone discussions to keep communication tight, but because we are a team of three and mostly work on the project in the same room communication is always possible and we do not use strict times for discussion.

In our process, the most important communication is coding together, problems that arise with one group member can expect direct feedback and help from another member. Design decisions are discussed until an agreement is met in the group. There has not been an event where we have not gotten a satisfying conclusion for all members about a certain decision. Implementation decisions are also discussed but are ultimately decided by the member assigned to the implementation. In this way a balance is struck between the cumulative strength of group knowledge and the individual responsibility of the member working on a certain feature. The individual can become an expert on a subject and lead decisions about the subject in the future.

4.1.1. Communication with the Coach

The collaboration with the TU Coach has been adventagous in connecting our service stack with the Exposure Render[5.1.3]. An important point brought forward in chapter 5.2 is that there is significant technical debt in the previous system, there is no external documentation or tests written and some API calls are hard to reproduce. But fortunately the coach was the person who developed Exposure Render and helped to get these pieces up and running.

The first half of the project we had weekly meetings with the coach, ranging from an hour to a whole morning and after that the coach took a more reactive stance through chatting platform Slack where we could get key explanations about Exposure Render whenever we needed it.

4.2. Planning

The Scrum methodology is used to give shape to the planning, and we use its definitions of sprints, Product backlog, Product owner and Scrum master. Sprints of a week are used and we had eight sprints to complete the project. Sprints go from monday to friday and the weekend can be used for members that feel like doing more work. There have been many days where we have worked until the closing of the building, not because of stress for a deadline, but because of the addictive feeling of programming together on a worthwhile project.

To manage the sprint the Github projects feature is used. This is a new feature where tickets can be put on a Kanban board to manage the sprint. For every sprint, the backlog of the sprint is put in a new column. Every ticket of the sprint is then assigned a team member that is responsible for

its delivery at the end of the sprint, either by coding it himself or by checking in periodically with the implementer. Columns for tickets that are currently worked on(In Progress), waiting for review(Testing) and that are done are available and team members can drag tickets to show the status on the board. A screenshot of our board during the project is shown in 4.1. The product backlog is continuously filled

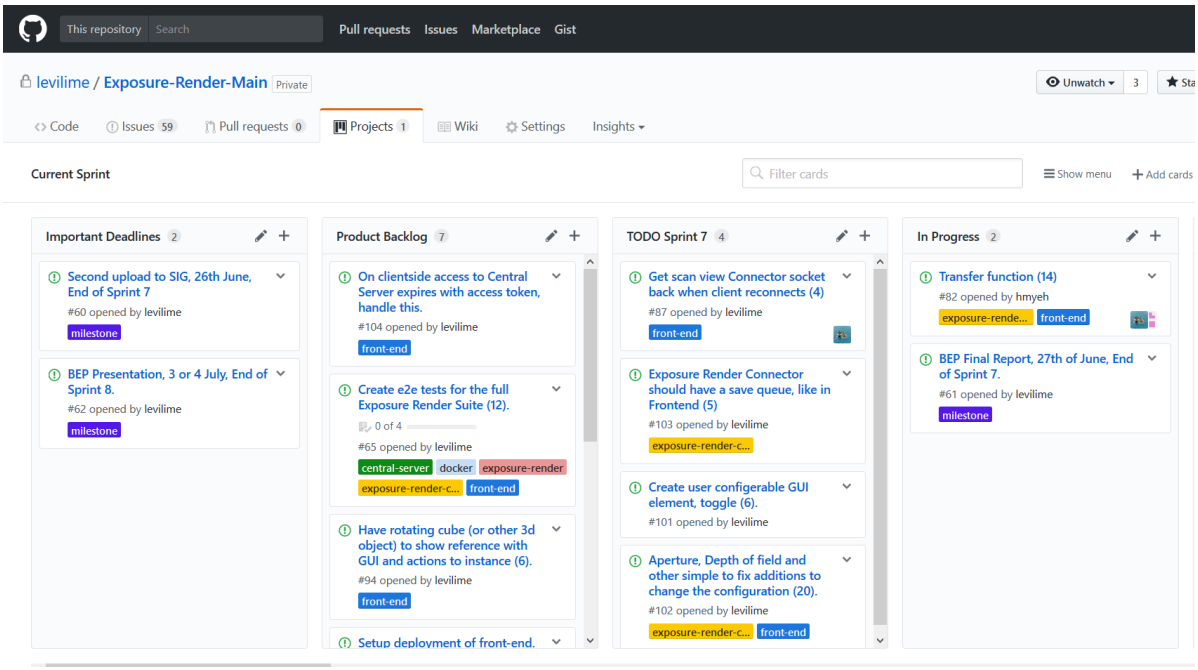


Figure 4.1: Github Projects screenshot of the project

with tickets that all members can bring forth and at the end of a sprint a new sprint is composed from tickets in the backlog. Tickets are chosen that bring forth a new runnable version of the system with new features and the emphasis is on creating new functionality quickly while not cutting corners that endanger non-functional requirements. In our process the Scrum master and the Product owner are the same person. But individual opinions and preferences of group members on the backlog are often granted, deciding upon the new sprint is a group decision.

4.3. Version Control

The Exposure Render Web service should be seen as a number of separate software parts that together create the complete service. These are brought into separate repositories. This emphasizes them being separate entities and preventing hard to maintain monolith constructions. All parts have their own continuous integration and their own development environment to help members work on parts simultaneously.

We use the industry standard git as version control technology. We work together with this system using git flow, meaning the distinction is made between the master, develop and feature branches. Team members create new feature branches from the develop branch and create Pull Requests to the develop branch which are reviewed by other members. The feature is allowed to go into develop if: the Continuous integration passes, the reviewer finds the static code quality agreeable and it implements the user story it was tasked to fulfill. At the end of every sprint, a new runnable version should be available at the tip of the develop branch, a pull request can then be made to bring the develop into the master and have a new working version. The version is always working because all pull requests will only be accepted if the continuous integration, which shows the validity of the version, passes

5

Design

5.1. Previous Works

Before the start of this project, a prototype of the Exposure Web Service (RemoteRendering) had already been designed and implemented by the TU coach. RemoteRendering consists of an adapted version of the Exposure Render, a backend server and a frontend application. The frontend application is able to show and control the rendered scene of the Exposure Render. The backend server handles the communication between the frontend application and the Exposure Render. The adapted Exposure Render renders and streams a scene to the frontend. The architecture of RemoteRendering is visually shown in figure 5.2.

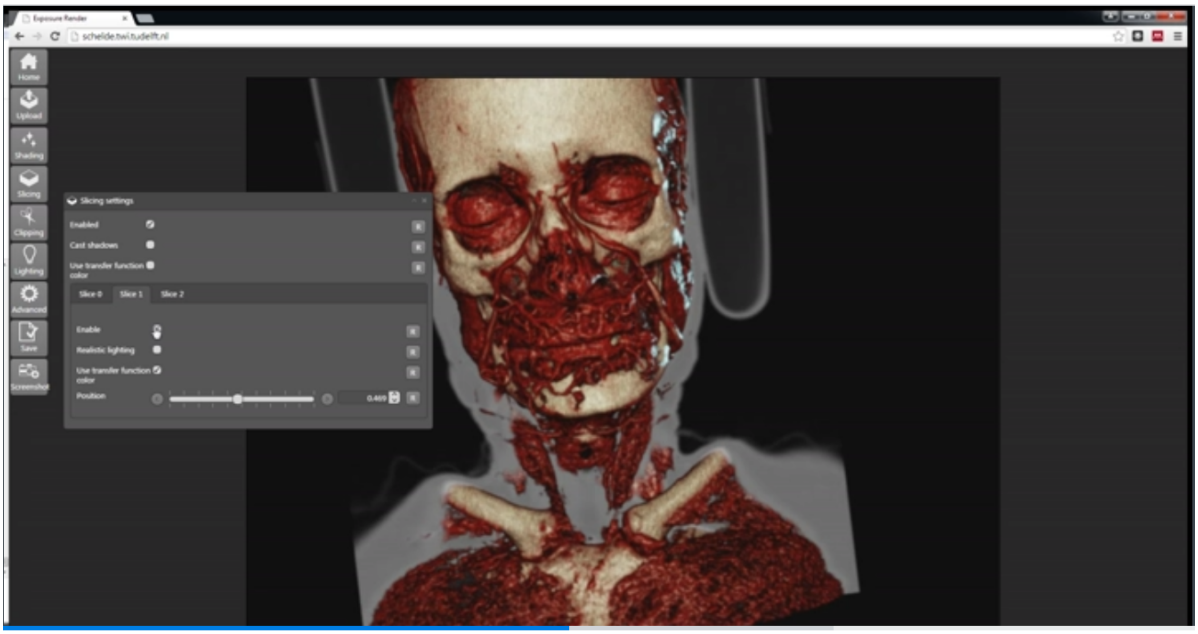


Figure 5.1: Screenshot of previously made WebGUI RemoteRender with Jquery Widgets

5.1.1. Frontend application

The frontend application was tasked with uploading DICOM files, showing the rendered scene and changing the settings of the Exposure Render. The application makes use of a number of open source JavaScript libraries: jQWidgets, JSMpeg, JSDicom and ThreeJS. A screenshot of how it looks on the frontend is shown in figure 5.1

The application is able to upload DICOM files to the server. The user is able to select a directory and the application will analyze the specified directory for DICOM files. The analysis of the directory is

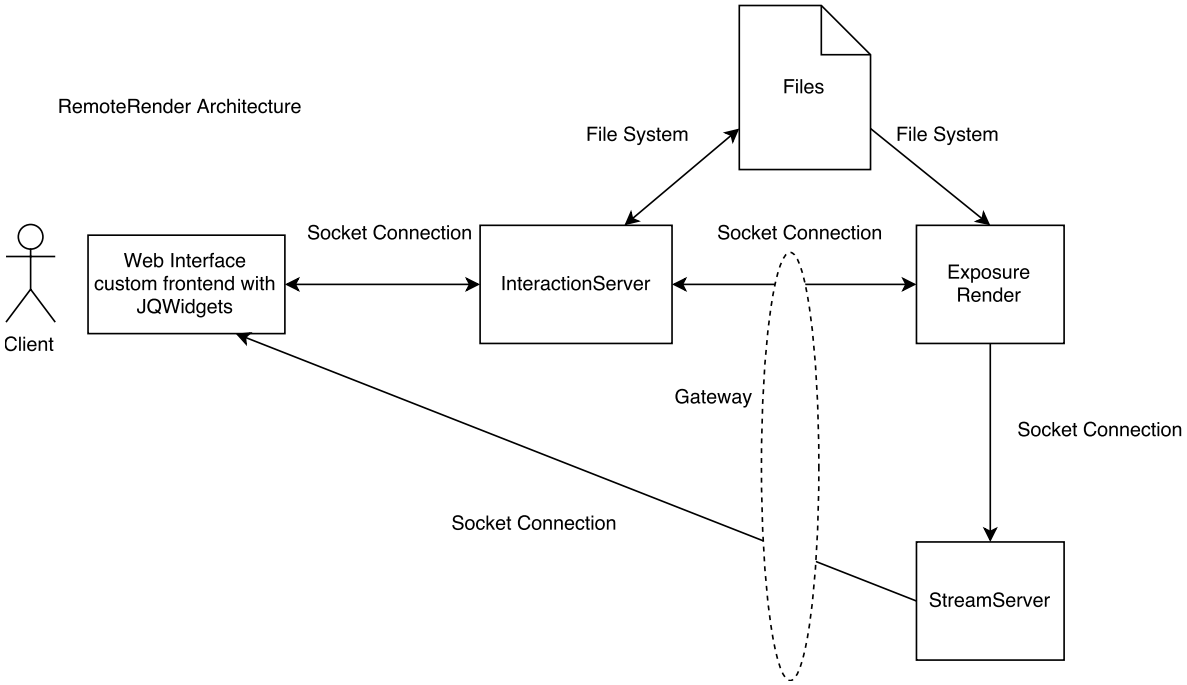


Figure 5.2: Architecture of previous project: RemoteRender

done by reading the headers of the DICOM files with the help of the JSDicom parser. After finding the different sequences of DICOM files, the user can select a sequence and upload it to the server. After that, the DICOM files are sent as chunks to the server with a WebSocket connection.

Besides that, the application is able to control the settings of the Exposure Render. The User Interface of the application consists of sliders and buttons from jQWidgets. It allows the user to customize the settings of the Exposure Render. The movement of the camera is done by tracking the mouse movements of the user with ThreeJS. The changes the user makes to the User Interface and the mouse movements are sent to the server with a WebSocket connection. For the communication between the client and the server, each message is sent starting with the predefined name of the action. Then the required data is appended and delimited with semicolons. To update a specific setting of the Exposure Render, the data needs to contain the new value of the setting as well as the corresponding GUID from the Qt User Interface of the Exposure Render (see Adapted Exposure Render).

The rendered scene of the Exposure Render is shown in the GUI of the application. The Exposure Render creates a WebSocket for the stream in order for the application to get the video data. With this stream together with the JSMpeg library, the application can show the stream in the application.

5.1.2. Backend server

The backend server had the task to handle the uploading of DICOM files as well as converting the messages between the application and the Exposure Render. The server has two WebSockets for the clients. One of the WebSockets is used for file uploading and the other one for changing the settings of the Exposure Render. As for the communication with the Exposure Render, the server uses TCP. The technology used for the server is the Tornado web server package of Python 2.

The server is able to receive sequences of chunked DICOM files from clients. After having received all the chunks, the server needs to process the files to an Exposure Render readable format. In order to achieve this task, the server uses multiple python packages to parse and read the DICOM files. During the process of converting the files, a preset.xml is created that stores the Exposure Render settings for the converted sequence. Additionally, a thumbnail is created showing a rendered gif of the DICOM sequence.

As described in 5.1.1 the messages from the client are received as string messages starting with the predefined action name. The server will either perform the action or forward the message to the Exposure Render Render in binary format. The binary formatting is done with the Python library for Qt.

The server will then forward the binary message to the Exposure Render through the TCP connection. This allows the client to make changes to the settings under which the Exposure Render renders the scene.

5.1.3. Adapted Exposure Render

The Exposure Render renders the scene of a given DICOM sequence. However to be able to show and interact with it in the client, the Exposure Render needs to be able to handle the communication between itself and the server as well as create the video stream. The Exposure Render uses Qt and FFMpeg to fulfill these additional tasks.

For rendering the scene, the converted files from the server need to be available to the Exposure Render. Then it can read the preset.xml with the volumetric data to render the scene correctly. This has been done by running the server and the Exposure Render on the same machine.

The adapted Exposure Render establishes the TCP connection to the server with Qt. After having established the connection, the binary messages are decoded and encoded with Qt. The decoded binary message contains the predefined action name as well as the data. For changes to the settings of the Exposure Render, the corresponding GUID of the Qt User Interface needs to be supplied with the new value. With the GUID the Exposure Render searches the preset.xml stored in memory for the setting and updates the value. Then the Exposure Render updates the scene according to the updated preset.xml.

The adapted Exposure Render starts FFMpeg as a separate process. Then this process will create a video stream of window containing the rendered scene. The settings for which the FFMpeg process is started with are configured in the Exposure Render.

5.2. Assessment of Previous Works

In this section the RemoteRendering prototype will be assessed. The purpose of this assessment is important for determining the usefulness of the RemoteRendering prototype in this project (e.g., to build upon RemoteRendering or not). The criteria for the assessment consists of: the functionality, maintainability, security and scalability.

5.2.1. Functionality

The functionality of the RemoteRendering prototype has been compared with the functional requirements defined in chapter 3. It fulfills only three of the five minimal requirements. RemoteRendering has implemented a UI with jQWidgets and is able to process the uploaded data to the correct format. The stream of the Exposure Render can also be viewed in the frontend application of RemoteRendering. The rest of the requirements have not been implemented by the RemoteRendering prototype.

5.2.2. Maintainability

Maintainability is important for future development. It allows for simpler development of additional features, since it becomes more obvious how to extend the existing code. We have assessed the RemoteRendering prototype according to the criteria of SIG for maintainable code in figure 5.1 [3].

The application has mostly long blocks of code, which use global variables from other files. There are also cases where the frontend has duplicated code with only a different input string. The frontend has no classes it uses, thus also no interfaces. The frontend does separate the concerns in different modules, but the concerns are not truly separated due to global variables. This also means that the architecture components are coupled. There are no tests written for the frontend application. The code is hard to grasp due to variables and methods from different files as well as no comments.

The backend server also has long blocks of code. There is in the same way as in the frontend a couple cases of duplicated code. There is no use of interfaces in the server. The concerns are more separated than the frontend due to the use of classes for each concern. The architecture components are still coupled due to the use of global variables. There are no tests written for the server. The code is also hard to grasp due to global variables and no use of comments.

The adapted Exposure Render has short and simple units of code. There is also no sign of duplicated code. The interfaces are also small. The concerns are separated in different packages. The architecture components of the adapted Exposure Render are heavily coupled to the Qt user interface. There are no tests written for the Exposure Render. The code is understandable even though a lack of comments.

	Frontend application	Backend server	Adapted Exposure Render
Write short units of code	2	1	4
Write simple units of code	2	1	4
Write code once	1	1	4
Keep unit interfaces small	N/A	N/A	4
Separate concerns in modules	2	3	5
Couple architecture components loosely	1	1	3
Keep architecture components balanced	3	3	4
Keep your codebase small	2	3	4
Automate tests	N/A	N/A	N/A
Write clean code	1	1	4

Table 5.1: Maintainability Assessment based on SIG criteria with scores ranging from 1-5. The scores are assigned by us.

5.2.3. Security

A very important criteria is the security in case of actual deployment of the system. However, there are no security implementations causing anyone to be able to access the medical data.

5.2.4. Scalability

Another criteria is the scalability of the system. The RemoteRendering does support the ability for multiple users to connect to the backend server. However, there can only run a single Exposure Render at a time. Thus it can only show the same rendered scene to different users. In other words the RemoteRendering prototype is not scalable.

5.3. Exposure Render Web Service

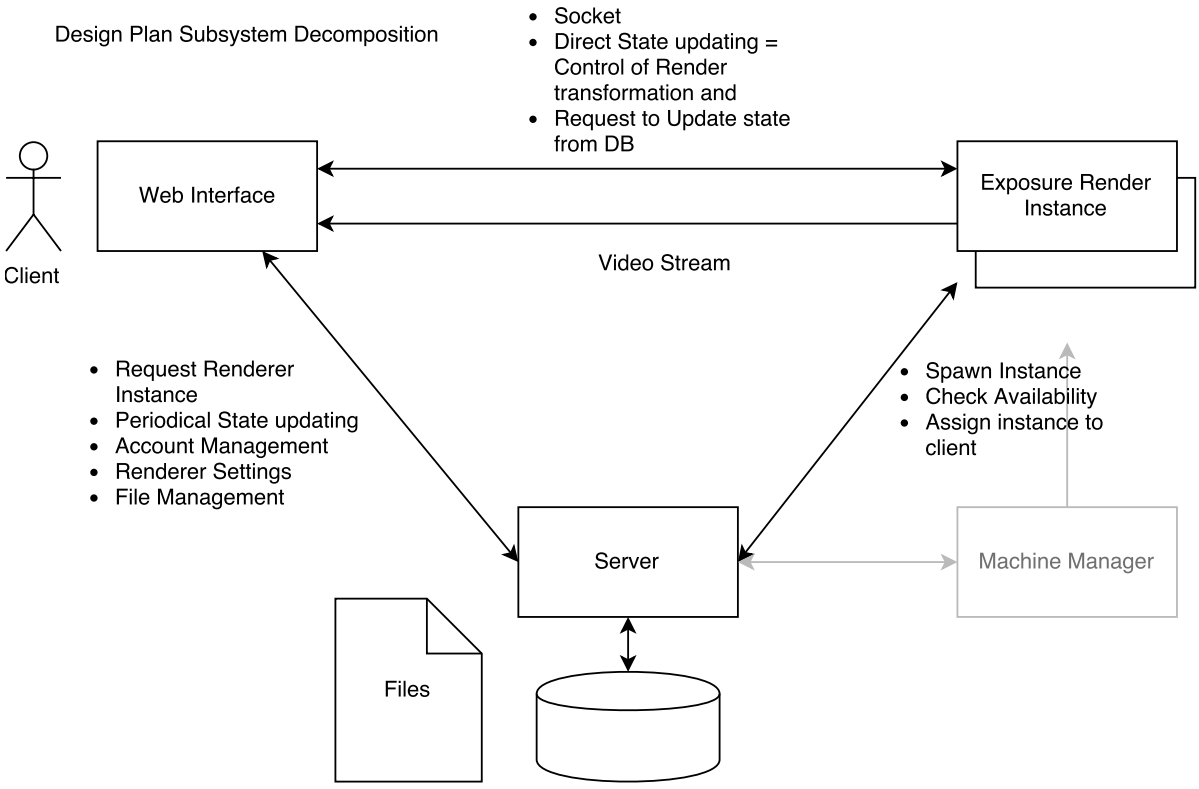


Figure 5.3: Design of High level system capable of fulfilling the requirements

As concluded in the previous section, RemoteRender cannot be easily extended to fit the functional

requirements of the client and our non-functional requirements. The subsystems need to be interchangeable so parts of the service can be redone in future development without touching the whole code base. By using the

The REST architecture is an architectural style that allows machines to communicate with each other using a uniform interface on the internet [4]. A REST-compliant service needs to adhere to the following constraints [5]:

- Client-server model: this separates the user interface from the storage of data
- Stateless requests: each request needs to contain all the required information to process it
- Cacheable web resource: each response defines if the web resource if it is cacheable
- Uniform interface: the client uses predefined operations to access or manipulate the web resources
- Layered system: the client is not able to see beyond the connected party (i.e., unable to see if the connected party is the end server or an intermediary server)

The REST architecture fits the design of the architecture, since the design has been created with the client-server model in mind. The use of the client-server model for the Portal and the Server simplifies the maintainability of each of the code repositories. This advantage is amplified by the uniform interface of REST.

The WebSocket protocol is a two-way communication protocol between client and server [6]. In this protocol the client initiates the connection with the server. Afterwards binary or textual messages can be sent through this connection to the other application. The communicating applications will need to agree on the message types used.The connection stays open until either the client or the server requests closing the connection.

There are also clear use cases for two-way messaging of the application. The WebSocket protocol also fits the design of the architecture. All of the applications need to have a communication protocol for two-way communication, which the WebSocket protocol is suited for. Because the user interface is run in the browser, the service should communicate through REST APIs when using HTTP requests and use WebSockets for two way messaging. These protocols can also be made safe from man in the middle attacks by using SSL certificates. Subsystems can be decomposed in the following way to fulfill the requirements as shown in figure 5.4.

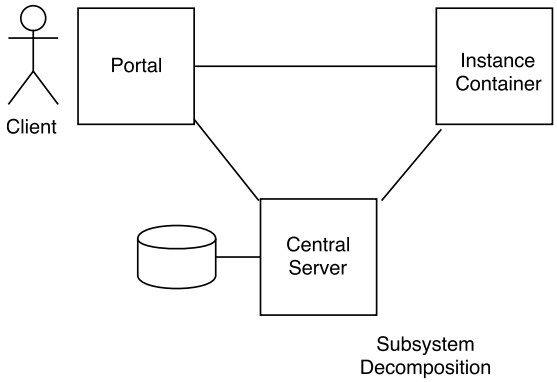


Figure 5.4: High level service decomposition

- A portal that is served to every browser wanting to use the service. Because the design is aimed to be RESTful this will be a single page application.
- A Central Server containing the user information and the scans. It should use a database to be able to manage large amounts of data. It will have REST API for the portal requesting information.
- The instance Container, where all systems are placed that involve the showing and interacting of a scan. There can be many instance containers in the service running at a time, and their amount and availability is dynamic.

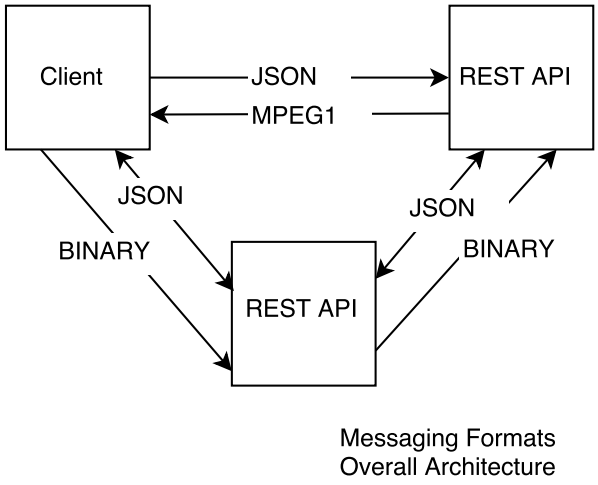


Figure 5.5: Messaging between the subsystems

In the research document the Machine Manager system has been described and is shown in figure 5.3 in grey. The machine manager can create more Instances when the demand increases but also takes into account how much hardware is available. It is not a part of the current Exposure Render Web Service because it is very dependent on the deploy environment. This is further discussed in chapter 8.

With this subsystem decomposition and lines of communication, a design is created that fulfills the requirements when implemented. The next chapter be an in-depth documentation of the implementation.

6

Implementation

6.1. Overall Implementation

Exposure Render Web Service will be made out of technologies that converge naturally to the requirements by design. Meaning frontend technologies will be used that advertise an user-centric experience, and backend technologies will be chosen that has support for both http requests and socket connections and helps deliver both secure and immediate communication. Figure 6.1 shows the design with the used technologies. Each subsystem will now be described in detail using the naming for each system in Figure 5.4.

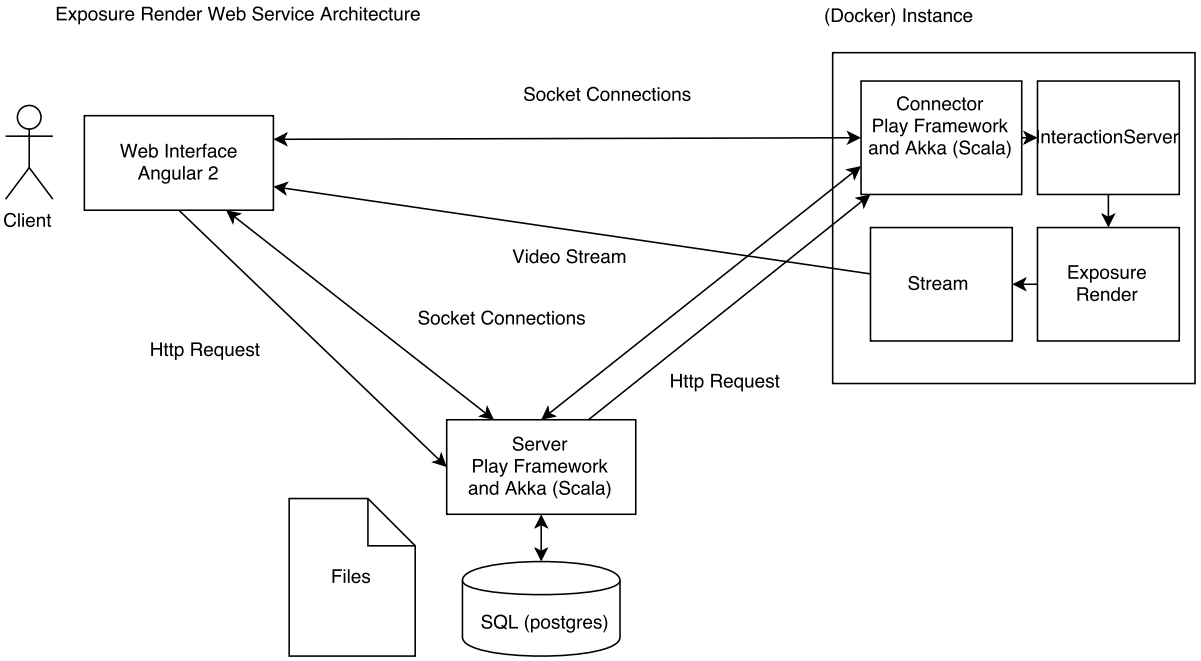


Figure 6.1: Technologies used for subsystems

6.1.1. Portal

The portal is a single page application by design. It communicates with other entities through JSON messages, either through HTTP requests or WebSockets. A mobile screenshot of the portal can be see in figure 6.2.

Framework

Angular is chosen as the frontend technology for the service. Angular has a clear paradigm to create a modular design and it has good support by design for two-way data binding. This means no additional

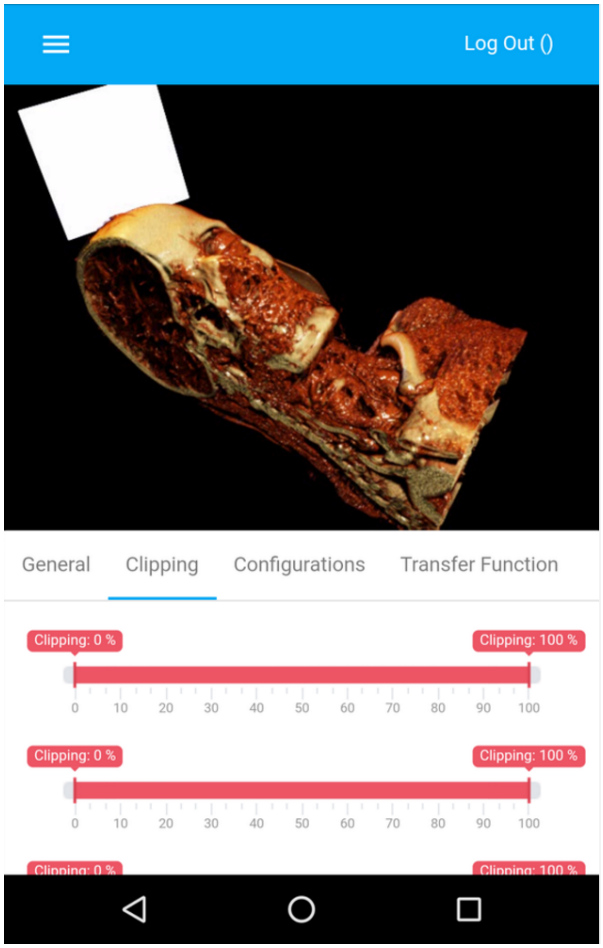


Figure 6.2: Screenshot of the portal on a mobile device

code has to be written to change the interface when the data changes and the other way around. The design is shown in Figure 6.3.

Modules

There are four main modules that each map to a page and summarize the features of the frontend.

- The Home module contains the home page where the user can login with the third party Authorization service. This service helps to make the service secure and is described in detail in section 6.5.
- The upload module contains an upload page where users can drag and drop their MRI data where it is partially anonymized and readied for usage on the backend.
- The Scans Module shows the available scans of the user and from their the user can request a scan for viewing.
- In the Scanview Module the controls and stream window are in place to interact with the scan running remotely. A flux style paradigm is used to change the settings through actions like controlling the camera or changing sliders. One settings object, a JSON, contains all controls of the scan. In this way the settings can easily be stored on the backend, and be loaded in the frontend to map the GUI to the data. In the code, settings and configuration are used interchangeably for this object. The streaming window in the ScanView Module has an involved implementation and will be discussed in 6.3. The Scanview Module also handles all WebSocket connections with Central Server and the Connector and Stream Server of the instance.

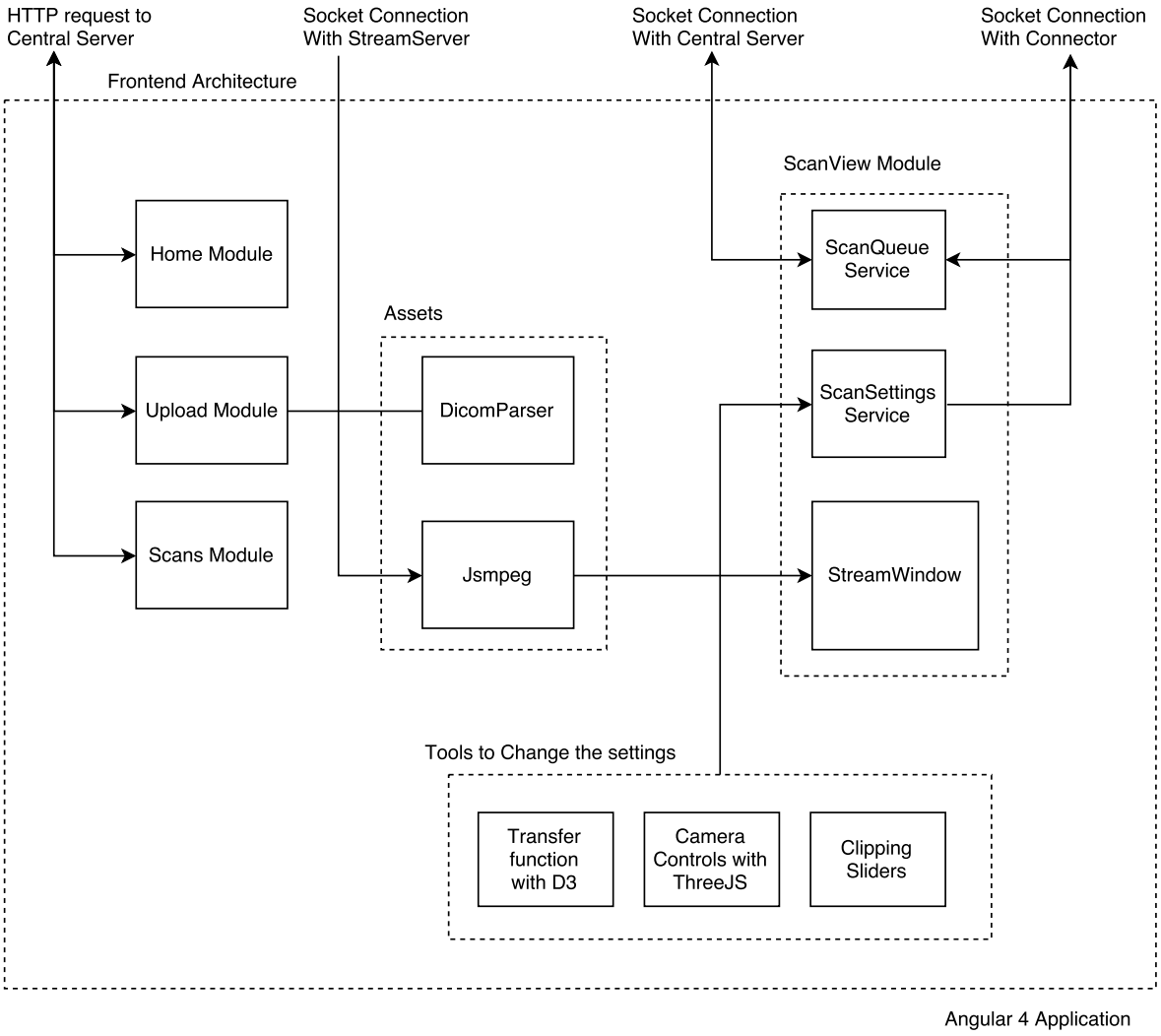


Figure 6.3: Design of the Front-end with Angular

Dependencies

Apart from Angular that forms the main framework on which is built, there are more proven libraries used in the frontend. They have been added to create functionality faster and to keep the code base small.

To create a user-centered interface, it should be easy to use, be attractive looking and be responsive. Responsive means that the interface works for different screen sizes, from a smart phone to a big monitor. The decision was made to use an involved style library that would unburden the team from writing such code ourselves. Material-UI was chosen and its styling is used consistently throughout the frontend.

For the transfer function D3[1] has been added. It is used for data-driven visualization and the transfer function on the frontend should be an interactive graph representation that when interacted with changes the rendering in Exposure Render. D3 saves a lot of time, because it handles DOM management and lets the developer focus on how the data is visualized and bind events to let the visualization change the data.

6.1.2. Central Server

The Central Server is the central authority for persistent data in the service. It also has the connections to all running instances and decides their activity. A user can get a running instance showing the

[1] https://d3js.org/

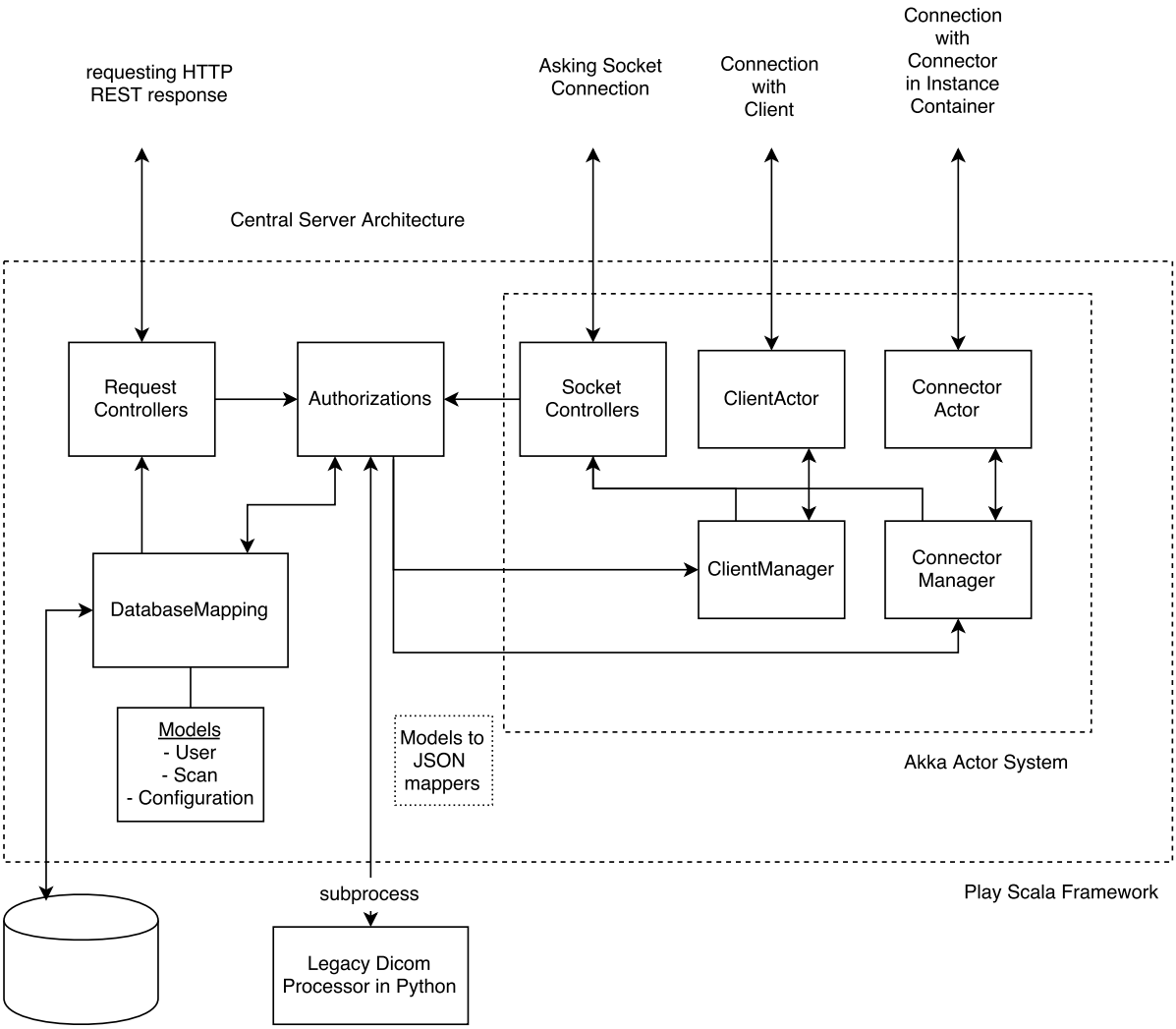


Figure 6.4: Design of the Central Server

requested scan data by asking the central server. This will be explained in more detail in section 6.2.1. In figure 6.4 the design is shown of the central server. The Central Server has an REST API for HTTP requests and should process messages in WebSockets sent in a JSON format.

Framework

We have decided to use the Play Framework in Scala as technology to fulfill the central server requirements. It has support for both HTTP requests and WebSockets and has well known libraries for database mapping and processing authorization requests through a third party authorization service.

Modules

The Central Server handles traffic with five separable systems.

- HTTP REST requests
- Persistent Storage
- Running of the Legacy DICOM Processor in Python ported from RemoteRender
- Socket connections
- Writing to files in the filesystem.

HTTP REST requests

The Central Server offers an REST API. All calls to the API are checked on authentication.

```
GET     /users/current              Get the representation of the current
user in the database

GET     /users/:id                  Get the representation of a user, can
only get self
GET     /scans                      Get scans that the user has permission to
POST    /scans                      Upload scans
GET     /scans/:id                  Get the representation of the scan
GET     /scans/:id/configurations    Get all configurations belonging to
the scan
POST    /scans/:id/configurations    Add a new configuration to the scan

DELETE  /configurations/:id         Delete the configuration
```

Persistent Storage

The central Server must be able to connect authenticated users to their scan data and grant others permissions based on identity. The central Server must store these permissions persistently. A database technology is necessary, therefore the central server should be able to map data from a database to its internal model representation. Slick is used to make this mapping. Additionally flyWay is used to handle data migrations during the life of the application.

Authorizations

As seen in figure 6.4 all traffic goes through the Authorizations module. It contains methods to authenticate the users of the request and validate whether they are allowed certain resources and actions. It is coded modular by creating authorization functions that accept other authorization functions, creating a chain of closures. This is done, to avoid code duplication while handling asynchronous actions.

Writing to the filesystem and Running the Legacy DICOM Processor as a subprocess

The Client can upload scan data by doing a POST request to the Central Server after which it is processed. The implementation of this functionality is described in section 6.6.

Handling connections with state, Websocket connections

The Akka Actor System is used to handle connections with state in the application. This is built in to the Play Framework, because Play is built on Akka. The Actor system is a nice way to handle these connections, modelling every connection as an Actor, which can receive and send messages from outside the application and supports internal traffic within the application. These messages are all asynchronous and allow an event-driven programing flow. Which is appropriate given WebSocket messages from outside can be received sporadically. Messages can be sent to other actors within the system. Managers are used to keep track of actors and send messages between different kinds of actors. In the Central Server ClientActors manage incoming connections of the front-end application, that want to receive a scan to view. Connector Actors manage incoming Connector connections that are available to receive scans and connect with the frontend. The following endpoints are used that accepts WebSocket connections.

```
GET     /instance                   For the connector registering to the
Central Server

GET     /scans/:id/execute          For the frontend requesting a Scan with
id to be rendered in an instance and made available.
```

6.1.3. Instance Container

The Instance Container is a Docker container containing the systems required to render the scene, control the Exposure Render and stream the scene. In figure 5.4 the instance Container composition is shown. In the instance Container the following systems reside:

- Connector: connects the Portal, Server and Interaction Server
- Interaction Server: adapts the messages from the Connector into the correct format
- Exposure Render: renders the scene with the medical files
- stream server: streams the scene

6.1.4. Connector

The Connector has the task of connecting the Portal, Server and Interaction Server. It communicates with each of these applications through WebSockets or HTTP requests. In this way it can coordinate the actions the Instance Container should take.

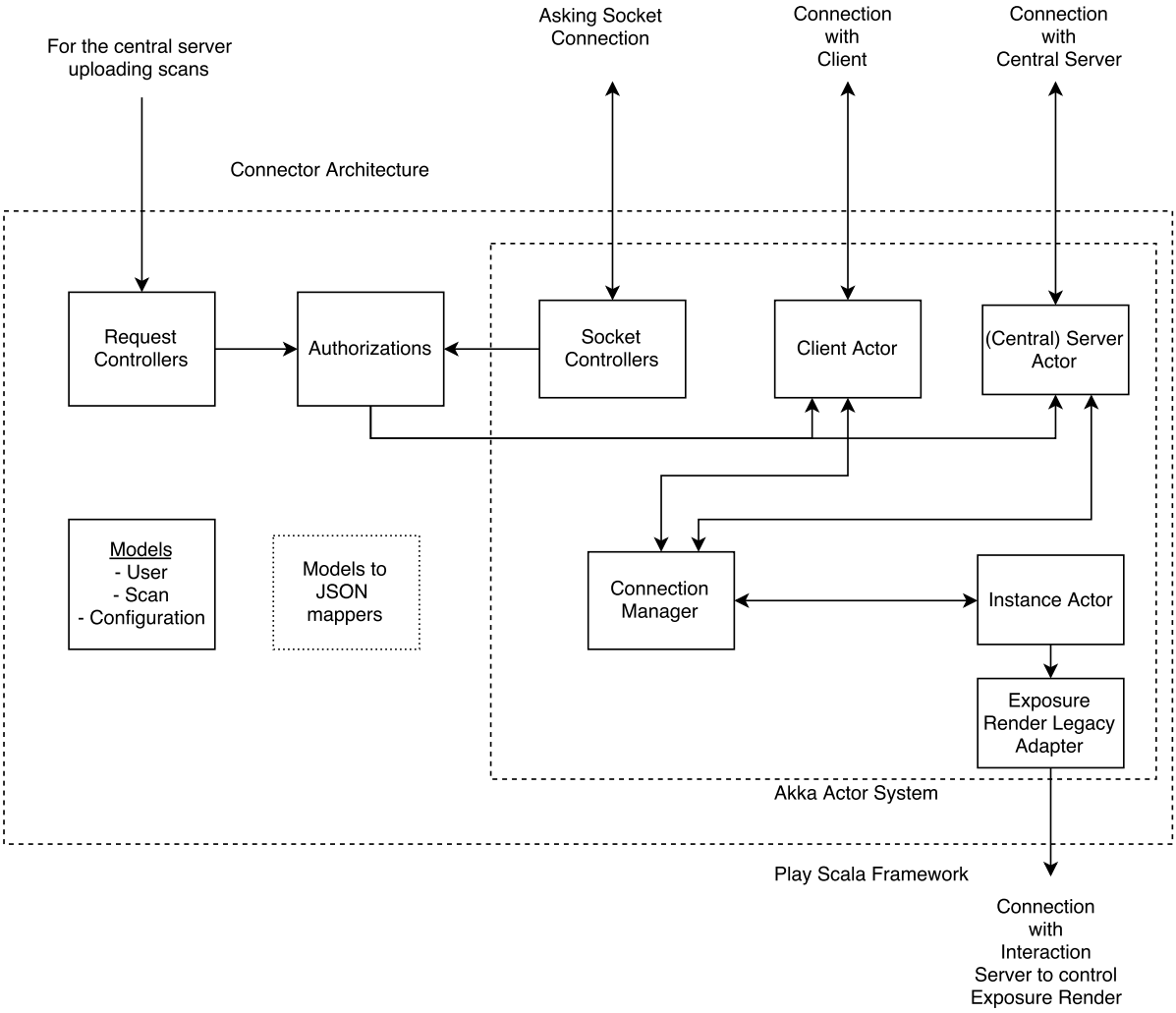


Figure 6.5: Design of the Connector

Framework

Similar to the Server, the Connector has been built using the Play framework in Scala. As mentioned before, it supports both HTTP requests as well as WebSockets. Besides that, the Play framework was chosen due to obtained familiarity from developing the Server.

Modules

The Connector coordinates the different applications with the following systems:

- HTTP REST request: for uploading the files to the Connector

- WebSockets: acts as both a WebSocket client for the central Server and Interaction Server, as for the Portal it acts as a WebSocket server
- Exposure Render legacy adapter: adapts the JSON messages to the correct format for the Interaction Server

HTTP REST request
The following REST endpoint is available on the Connector:

```
POST    /scans                      Upload the scan
```

Authorizations
When WebSocket connections are established, the establishing party is expected to send an authentication message. These messages goes through the Authorizations module to authenticate the connecting WebSocket parties. The Authorizations module contains methods to validate the identifiers.

WebSocket connections
The Akka actor system is again used for handling the WebSocket connections. An actor is created when a WebSocket connection is opened. As can be seen from figure 6.5, there are three types of actors for the connections: ClientActor, InstanceActor and ServerActor. After the initialization of the actor, it will register itself to the ConnectionManager. The ConnectionManager, which is also an actor, manages all the open connections and decides the types of messages which will be forwarded or actions will be performed. The following endpoint is available for the Portal to connect:

```
GET     /connect                    For the frontend controlling the Exposure Render setti
```

6.1.5. Interaction Server
The Interaction Server is tasked with adapting the messages of the Connector to the correct format for the Exposure Render. This task is achieved by creating a TCP server for the Exposure Render and sending the messages as binary streams. The implementation has been slightly adapted from the RemoteRendering prototype (see 5.1).

Dependencies
The Interaction Server is developed with Python 2 and requires a number of Python2 packages. This is due to the fact that the DICOMprocessor functionality is still contained within the Interaction Server. The main dependencies required for the Interaction Server to run as an adapter piece are the Tornado and scipy package. The Tornado package is used for creating the WebSocket server for the Connector and the TCP server for Exposure Render. The servers listen to the ports specified in the configuration file. The scipy package is used for interpolating the transfer function points of the Portal to a smooth graph. Then values will be read from the graph to create a texture for the Exposure Render to use.

6.2. Messaging of the applications
The communication of the applications is essential for the coordination of tasks between the different applications. To achieve this coordination, each application needs to understand the received messages as well as send the appropriate types of messages to the other applications. Therefore messaging protocols have been specified to fulfill the coordination of tasks.

This section will give a high-level overview for the flow of requesting and controlling an Exposure Render container. Then the communication between each of the applications in the flow will be explored in further detail.

6.2.1. Overview of message flow
In figure 6.6 the message flow is shown from the user requesting an Exposure Render container until the user can view and control the rendered scene of the medical data. This flow will be explained in detail because it shows the interconnection of all subsystems. It also shows a very important feature, securely requesting and receiving a interactive rendering instance in a complex multi user and multi container environment. In the flow the user is able to request an Exposure Render container from the

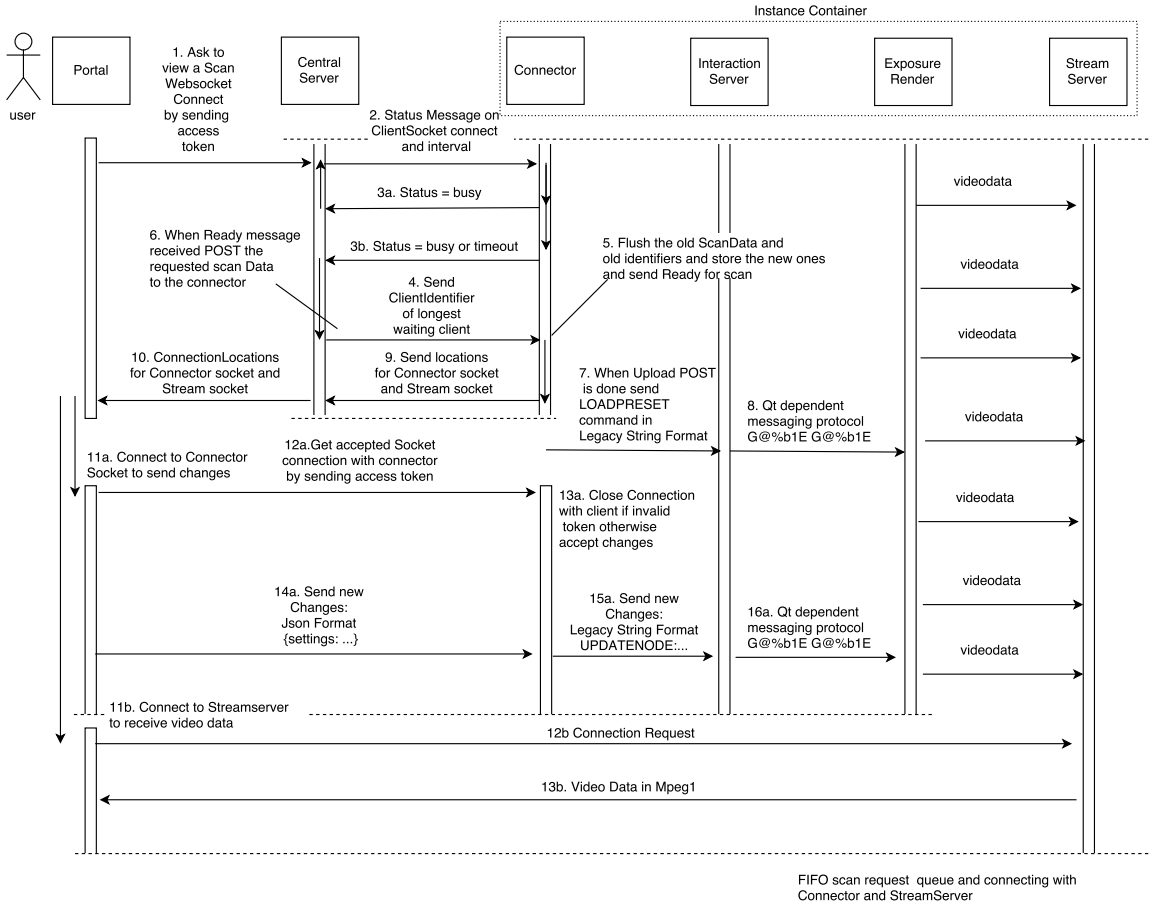


Figure 6.6: Complete Process uml of browser running the portal requesting a scan to view remotely from the Central Server. And receiving the location of an instance to connect to and interact with the scan in the end

central Server. When a container is assigned to a user the container will prepare itself by removing the old data and storing the new data. The user will then be able to connect to the container, after having received the necessary information from the central Server. The user can now view the stream of the rendered Exposure Render scene. Besides that, the user is able to control the settings of the Exposure Render with the UI by sending control messages to the container.

6.2.2. Communication Portal and Server

The communication between the Portal and Server is shown in step 1 and 10 of the message flow figure 6.6. At these steps the Portal requests an Exposure Render container, the Portal initiates a WebSocket connection with the Server. Before the Server registers the Portal to the queue of waiting Portal clients, the Portal needs to send an authentication message to the Server to identify if it is an authorized user. While the Portal client is waiting for to get an Exposure Render container, the Server will poll the status of the containers. The moment a container becomes available, the Server uses a First In First Out queueing system (from 3) to assign the container to a Portal WebSocket connection. With the location of the Connector and the corresponding stream server the Portal can establish WebSocket connections with both the Connector and stream server. The Portal is then able to show the stream of the rendered Exposure Render scene in the UI.

Next to the WebSocket communication in the Exposure Render container message flow, the HTTP communication between the Portal and Server is also implemented as a REST architecture as described in section 5.3. Thus the Portal can request resources from the Server through HTTP requests. The Server will respond with a JSON message containing the requested resources. Finally, the Portal interprets the JSON message and shows the requested resources appropriately in the UI.

6.2.3. Communication Server and Connector

The communication between the Server and Connector is shown in step 2-6 and 9 of the message flow figure 6.6. Before the steps from the message flow are executed, the Connector first initiates the WebSocket connection with the Server and sends an authentication message containing a shared secret key. This is then validated by the Server.

After the validation, the Server polls the Connector for its status. With the status of the Connector the Server will decide whether the Connector gets assigned to a new Portal client. This will be the case when the Connector is available or it is busy and has spent a maximum duration rendering for the same Portal client. The status message also contains the location of the Connector and the corresponding stream server.

When the Connector is assigned to a Portal client, the Server will send the identifiers of the users with permission to view the scan together with information about the scan to the Connector. The Connector will flush the previous medical files and identifiers, after which it will send an acknowledgement to the Server containing the location to upload the new medical files. The Server is then able to use a POST request to send the files to the Server.

6.2.4. Communication Connector and Interaction Server

The communication between the Connector and the Interaction Server is shown in step 7 and 15a in the message flow figure 6.6. The Interaction Server is from the RemoteRendering 5.1.2 and is used as an adapter for communication with the Exposure Render due to the use of Qt binary messages. The Connector will therefore act as the client of the Interaction Server in the old model. As a consequence, the communication between the Connector and the Interaction Server is done through a WebSocket connection. Even though the communication is one-way (i.e., from Connector to Interaction Server). The Connector sends messages as strings delimited with semicolons and starting with a predefined action name, such as LOADPRESET, TRANSFERFUNCTION and UPDATENODE. The LOADPRESET message is sent to the InteractionServer, when the Server has uploaded the new medical files to the Connector. For updating the settings of the Exposure Render the Connector sends UPDATENODE messages to the Interaction Server, when it receives settings messages from the Portal client to change the parameters for rendering the scene. There is an exception for the settings messages, namely in the case of changing the transfer function. This will instead send a TRANSFERFUNCTION message to the Interaction Server. This message will contain a number of points in order for the Interaction Server to interpolate the graph corresponding to the points. The Interaction Server will then create a texture from the graph and send it as an UPDATENODE message to the Exposure Render.

6.2.5. Communication Interaction Server and Exposure Render

The communication between the Interaction Server and the Exposure Render has not been adapted from the RemoteRendering version. The Exposure Render establishes a TCP connection with the Interaction Server. The messages going through this connection are formatted as TCP binary streams. The LOADPRESET and the UPDATENODE messages are thus converted to Qt binary streams and sent to the Exposure Render. Based on these streams the Exposure Render will perform the corresponding actions (i.e., load preset or change the settings). This is shown in steps 8 and 16a in the message flow figure 6.6.

6.2.6. Communication Portal and Connector

The communication between the Portal and the Connector is shown in steps 11a, 12a and 14a. The Portal establishes the WebSocket connection with the Connector. As with the previous WebSocket connections an authentication message is sent for validation. The Connector is able to validate the user due to the identifiers it has received from the Server as described in 6.2.3. After the authentication step the Portal keeps sending the changed parameters of the settings in JSON format to the Connector on a regular interval.

6.2.7. Communication Portal and stream server

The communication between the Portal and the stream server is shown in steps 12b and 13b. The Portal can establish the WebSocket connection. The stream server sends the video data from the Exposure Render to the Portal, which the Portal interprets and shows in the user interface with the

help of the JSMpeg library.

6.3. Video Streaming

Video streaming is a very crucial aspect of the Exposure Render Web Service. It provides the portal with the graphical representation of the MRI/CT scan data.

6.3.1. Implementation

The video source is provided by the Exposure Render. The Exposure Render instance starts a TCP stream server that outputs the raw scene data 15 times per second. Afterwards, an FFMpeg process is started. This process takes the TCP stream server as input and encodes the video data using the mpeg1video codec. The encoded output is sent to the streamserver from figure 6.7 as a custom NodeJS server.

This NodeJS server listens to the output data and appends it to a read stream. A WebSocket connection is instantiated between the NodeJS server and the client (frontend application). As soon as the client receives video data, it is read and decoded by JSMpeg.

JSMpeg is also able to render the video data using WebGL and Canvas2D. The client uses Canvas2D to handle this job, since it does not require any 3D features and Canvas2D supports more hardware and browsers than WebGL does. The whole aforementioned process is visualized in figure 6.7.

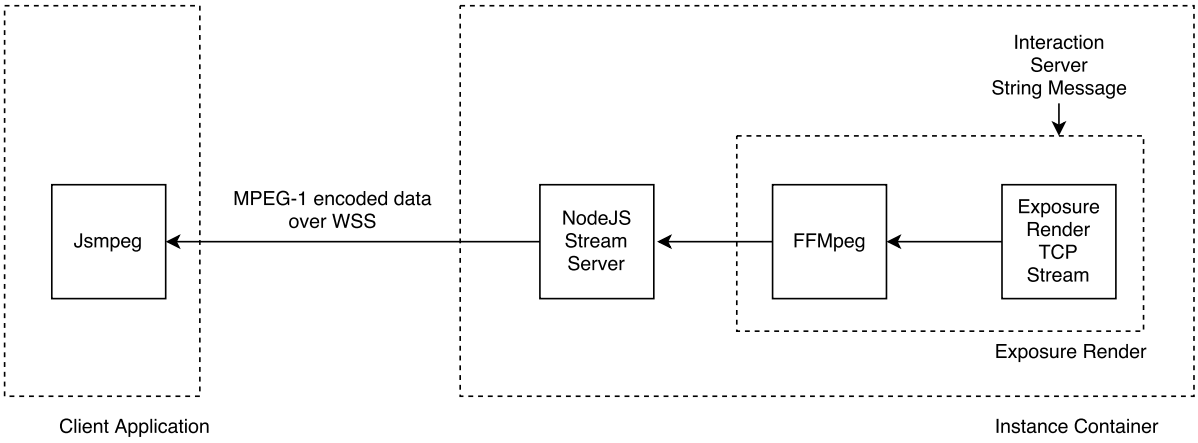


Figure 6.7: Visual representation of the streaming process

6.3.2. Codecs

A video codec can encode and decode video data. An encoded video stream can be sent over a simple WebSocket connection and the client can decode this encoded video data. Several different codecs have been used in this project, the following three to be specific:

- mpeg1video
- libx264
- nvenc_h264

MPEG-1

Mpeg1video (also known as MPEG-1) was the first codec that was used for implementing the live stream for this project. Mostly due to the fact that it is relatively easy to render clientside using JSMpeg[2]. MPEG-1 is known for its relatively high bitrate. A higher bitrate means higher quality, but the higher the bitrate, the more work JSMpeg has to do in order to decode the data. A high bitrate may also cause issues when using a relatively slow internet connection (e.g. WiFi or 3G connections).

[2]https://github.com/phoboslab/jsmpeg

H264 (MPEG-4)

Libx264 and nvenc_h264 on the other hand, are both based on the more recent H264 (also known as MPEG-4) compression standard. Nvenc_h264 distinguishes itself by using the CUDA cores of an NVIDIA GPU to encode video data, whereas libx264 and MPEG-1 use the CPU in order to do this job. H264 has a lower bitrate than MPEG-1, which provides a better performance. However, high detailed images may cause serious compression artifacts due to this high compression. An example of such a compression artifact can be seen in figure 6.8.

Figure 6.8: H264 compression artifact (source: `https://trac.mpc-hc.org/ticket/360`)

6.4. Containerizing

In order to make the application scalable, it is important to make the software modular. In other words, the software should consist out of different modules, each responsible for one certain task. Containerizing facilitates this process.

6.4.1. Implementation

This project is containerized by using a special version of Docker, namely NVIDIA Docker[3]. Docker itself does not natively support NVIDIA GPUs with containers. NVIDIA Docker on the other hand, mounts the required devices and drivers when starting the container on the host machine. A graphical overview of NVIDIA Docker can be seen in figure 6.9.

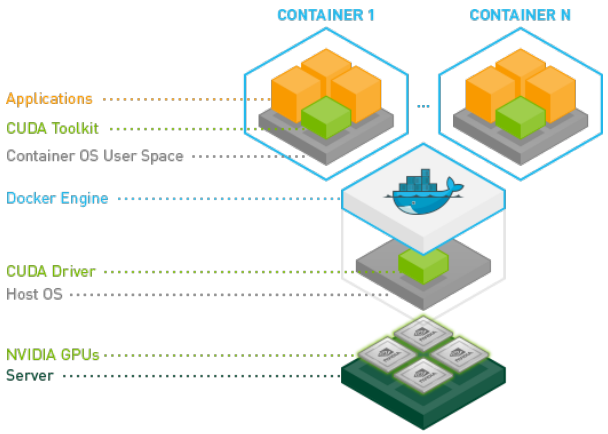


Figure 6.9: Graphical overview of NVIDIA Docker (source: `https://github.com/NVIDIA/nvidia-docker`)

This allows a Docker container to make full use of a CUDA-enabled GPU. This is necessary, since the Exposure Render requires a CUDA-capable graphics card and the CUDA Toolkit in order to work.

As basis for the container used in this project, a CUDA based Ubuntu 16.04 image is used. This image already contains an installation of the CUDA Toolkit and in combination with NVIDIA Docker, it will also have the correct drivers and devices mounted, thus allowing full usage of the GPU(s) in the server.

This project should containerize the following tasks:

- Exposure Render Instance
- Connector
- Interaction Server
- Stream Server

[3]https://github.com/NVIDIA/nvidia-docker

This can be achieved by running all tasks in one Docker container. This is relatively easy to implement, but is less clear than running each task in a different container. The latter can be achieved by using Docker Compose[4]. This tool can be used to define and run multi-container Docker programs. However, for the scope of this project, the networking and scalability using Docker Compose is too complex. The final product uses but one Docker container in order to run these 4 tasks. This is visualized in figure 6.10.

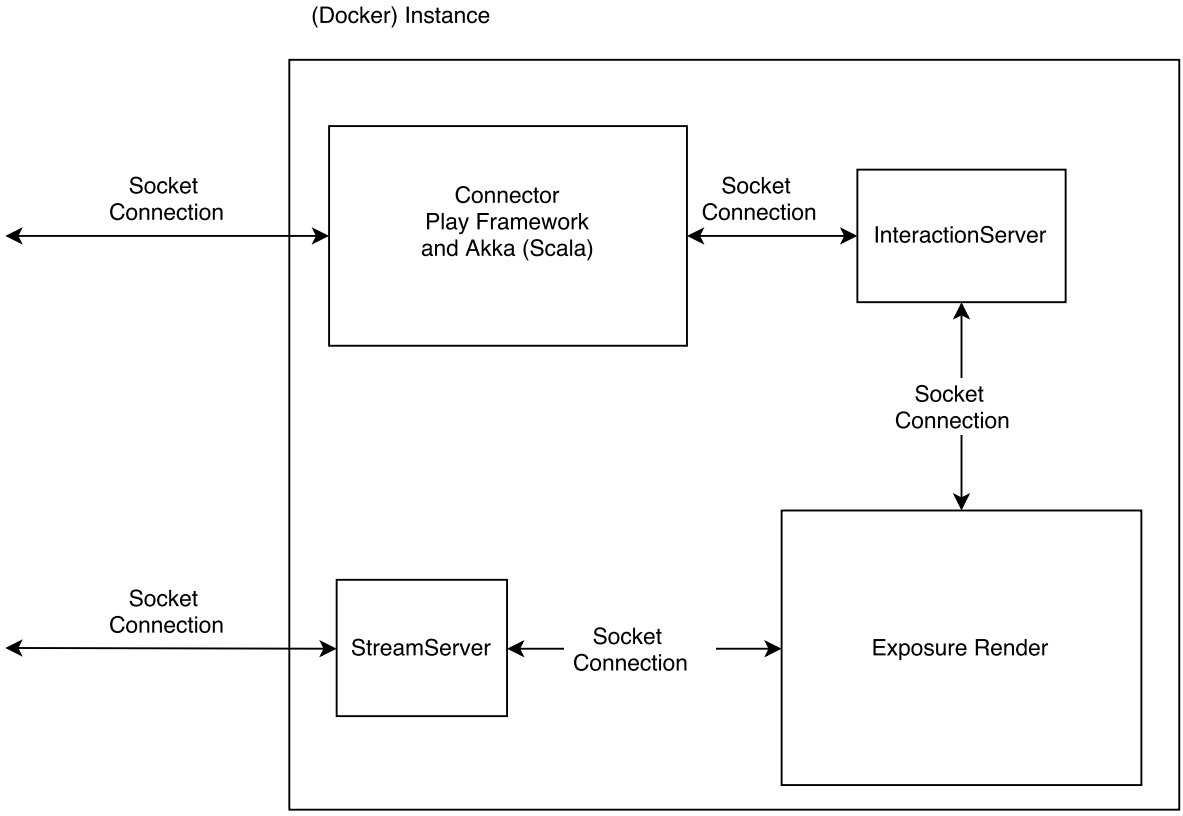


Figure 6.10: Visual representation of the final Docker container

6.5. Security

Another key point in the implementation is security, especially due to the personal nature of the data. Therefore it is important that every application is secure to different kinds of attacks. To be able for users to securely access the data, there needs to be a form of authentication. On top of that, the data needs to be securely stored on the Server.

This section will highlight all the security measures taken to protect the personal data. It will explain each of the measures and the kinds of attacks it offers protection for.

6.5.1. HTTPS and WSS

To secure the connection between the user and each of the communicating applications secure HTTP (HTTPS) and secure WebSockets (WSS) are used. The security layer of HTTPS and WSS is provided by the Transport Layer Security (TLS) [7]. When a connection is established with TLS, a handshake protocol is executed by both the client and the server. In this protocol the client and server negotiate over a shared secret key is used for the encryption of data. This protects the data from being read by intercepting parties. Besides that, both the client and the server send their certificates to each other. With the certificate the client can confirm the identity of the server. These two properties will ensure the protection against man-in-the-middle attacks[8]. A man-in-the-middle attack intercepts a conversation to eavesdrop and potentially send false messages to one of the parties.

[4]https://github.com/docker/compose

6.5.2. Authentication of users

For the authentication of users the implementation has been outsourced to a third party: Auth0[5]. Auth0 has been used by respectable companies, such as Mozilla and NVIDIA. To integrate the Auth0 authentication into the architecture, the Portal sends the login credentials of the user to Auth0 with the help of the Auth0lock library[6]. After Auth0 validates the login credentials, Auth0 will respond to the Portal with if the login was successful or not. In the case of a successful login the Portal receives two JSON web tokens: an access token and an id token. A JSON web token (JWT) is a compact encoded JSON string containing claims for the receiving party, such that it fits in the header of a HTTP request [9]. Such a token consists of three parts: the header, the payload and the signature. The header part contains the type of algorithm which is used for signing the JWT. The payload contains claims for the receiving party. The signature is for validating the JWT. It has to be noted that the claims of the payload can be changed, but then the signature of the JWT would be invalid. The signature is created with the payload in mind. The id token should be used for the frontend application for displaying user information. The access token should be used for the backend application to authenticate the user. Figure 6.11 shows the dependency on Auth0 in the overall design. The Portal needs to send the access

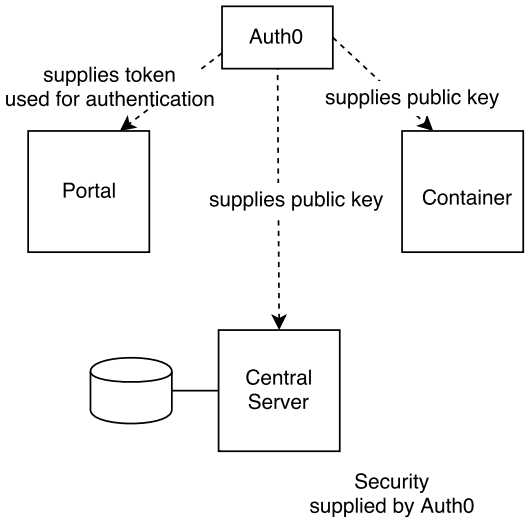


Figure 6.11: Dependency on third party authentication service Auth0

token to the Server, so the Server can authenticate the user and authorize him/her to the requested web resource. This is implemented by supplying the access token in the Authorization header of each request to the Server. Before the Server can validate the access token, the Server needs to be able to handle the signing algorithm. The chosen algorithm for signing JWT tokens is RS256, an asymmetrical signing algorithm[10]. This means that only Auth0 can sign the JWT tokens, since Auth0 is the only one with both the private and public key. Thus the Server retrieves the public key from Auth0 and is able to validate the access token from the user request.

6.5.3. WebSocket authentication message

For the authentication of WebSocket connections the client application is expected to send an authentication message containing either a shared secret or the JWT access token from subsection 6.5.2. For the Connector to be able to validate the JWT access token of the user, the Server will need to pass the user identifier to the Connector and the Portal will send its access token via the WebSocket connection. In this way is the Connector able to extract the user identifier from the access token for authentication. This will protect from unauthenticated WebSocket connections.

6.5.4. Encryption of files

The uploaded medical files need to be stored on the Server. This raises problems when the Server is hacked and the medical files can be read by the attacker. Therefore the files should not be stored in

[5]https://auth0.com/
[6]https://github.com/auth0/lock

their readable format. Instead, the files are encrypted on the server with the Auth0 user identifier of the uploader of the files. If the files are required to be rendered, the uploader supplies his/her user identifier from the JWT access token and decrypts the file for usage. Afterwards the files are encrypted. For other users, who have received permission from the uploader, a temporary duplicated version of the files are created for these users. The files are then encrypted with their user identifier. In this way the files are safely stored on the server.

6.5.5. Hashing of user identifer

For the identification of the user on the Server, the Auth0 user identifier needs to be stored on the Server. However, the user identifier should not be stored on the Server due to the use of this identifier for the encryption of the files. Instead the hashed identifier is stored on the Server. To hash the identifier the 512 bits cryptographic hash function variant of SHA-3 (Secure Hash Algorithm) is used [11]. A cryptographic hash function is a one-way hash function. In this way the stored hashed identifier cannot be reverted back to the identifier.

6.6. File uploading

Uploading files to a server would normally take little implementation effort. This is done by specifying a POST URL to which the frontend can upload the files to. The endpoint receiving the files should place the files in the appropriate place. Nevertheless, the Exposure Render requires a special format of the medical data and this makes the task of uploading the files more complex. For the implementation parts of the old RemoteRendering prototype have been used as described in 5.1 by porting the code to the frontend (figure 6.3) or using it as a sub process in the backend (figure 6.4). In this section the process of the uploading of DICOM files to Exposure Render readable files will be explained.

6.6.1. File uploading Portal

Before the DICOM files can be uploaded to the Server, the Portal needs to analyze the DICOM files. The user can select a directory containing DICOM files to analyze. After which the Portal will parse the tags of the DICOM files. In this way it can group sequences of DICOM files together according to their SeriesInstanceUID, since these are needed together for the conversion step. Then the Portal removes scouts and time series from the grouped DICOM files, since the Exposure Render does not support time series. After all of these steps, the Portal is finished analyzing the DICOM files and the user can select a sequence to upload to the Server. This will send an entire group of DICOM files to the Server. The code from RemoteRendering was ported to Angular, since the old code is not maintainable and was filled with jQueryWidget code.

6.6.2. File converting Server

After a group of DICOM files have been uploaded, the Server will call a sub process of Python to start the DicomProcessor of the old RemoteRendering prototype. This sub process will convert the files into the correct format and create a preset.xml containing the settings of the Exposure Render for the medical data. It uses VTK and DICOM parsers to achieve the conversion of DICOM files. As a consequence of these dependencies has it not been ported to the Server. After this process, the Exposure Render is able to render the converted files appropriately.

7 Ethics

We currently live in a society where protecting privacy is getting more important. This project handles extremely sensitive data, namely medical data. It is a big responsibility to make use of this kind of data and should be handled with great care. In the following chapter the ethical implications or private information by the Exposure Render Web Service is discussed.

Patients entrust the Exposure Render web service with their medical data (also known as personal health information or PHI in short). It is extremely important that the data handled in this project must be kept away from the outside world. This data should not end up in the wrong hands, since this can create some serious problems. This is more thoroughly described in the following section.

7.1. PHI and Cyber Crimes

Personal health information is getting more attractive to cyber criminals[12], since it contains personal data, such as a person's full name, date of birth, SSN etc., which is not changeable (like credit card information) and can be used for identity theft, acquiring medical equipment or drugs and file fraudulent insurance claims.

Whereas some cyber criminals steal PHI for financial gains, others are motivated by destroying an institution's reputation. Other cyber criminals try to make a political statement (hacktivism).

7.2. Anonymization

A lot of personal information can be removed from medical data. This is called anonymization[13] and can be achieved in different ways.

The most basic approach is simply removing the personal information (full name, date of birth etc.) from the medical record. On the other hand, one can also anonymize medical data by simply removing irrelevant visual representations of the patient (e.g. the face of the patient if he/she is suffering from a heart disease).

The Exposure Render Web Service anonymizes data as soon as it is uploaded. It does this by simply removing the patient's personal information.

7.3. Encryption

The data should also be encrypted. In principle preventing is better than curing, but if an external person manages to get his hands this data, it will still not be usable, since it can only be decrypted with a certain key that only the person who owns the data has.

The Exposure Render Web Service encrypts the medical data using a private key, which only the user has access to. Without this private key, the data is useless to cyber criminals.

The data is also protected from so called Man-In-The-Middle attacks (MITM). This is achieved by using a secure encrypted version of the HTTP and WS protocol, namely HTTPS and WSS.

8

Discussion and Recommendations

8.1. Self assessment

From the bachelor there have been multiple courses from which we have applied the acquired knowledge to this project. Object Oriented Programming, Software Engineering Methods, Web and Database Technology, Software Quality and Testing, Information and Data Modelling and Concepts of Programming Languages.

8.2. Product Assessment and Recommendations

When assessing the product, we should take a look back at the main requirements mentioned in chapter 3:

- User interface for uploading data
- Processing of uploaded data by the server
- Viewing the results of the Exposure Render in the browser
- Secure user and data management
- Port Exposure Render to Linux

The currently implemented version of the Exposure Render Web Service, fulfills all of the main requirements.

8.2.1. Frontend

The frontend application has implemented the following features:

- Responsive User Interface
- Displaying the stream of the Exposure Render scene
- Mouse controls for the stream window
- Changing transfer function
- Clipping the model
- Uploading the medical files
- User authentication through Auth0
- Changing and saving configuration settings of the Exposure Render

By comparing the implemented features of the frontend and the requirements for the frontend, it shows that all of the requirements have been satisfied.

However, the frontend can be extended in future work. The UI for uploading medical data could also have better support for anonymization, whereas the stream can be enhanced by using different codecs in order to gain performance and/or quality.

8.2.2. Server

The server has implemented the following features:

- User authentication with JWT from Auth0
- Saving configurations, users and scan information in database
- Storing files on the filesystem
- Encrypting and decrypting of files
- Converting DICOM files to Exposure Render readable format
- Manage Exposure Render containers

By comparing the implemented features of the server and the requirements for the server, it shows that all of the requirements have been satisfied. Even though all of the requirements have been satisfied, there are still improvements that can be implemented on the server, such as improving on the current implementation to decide whether an Exposure Render container should be assigned to a client or not.

8.2.3. Containerizing

Keeping software structured, modular and maintainable is extremely important. Containerizing also makes the task of GPU clustering easier, because NVIDIA Docker automatically distributes its workload over all different mounted GPUs.

However, or this project, it would be better to use multiple different containers for running the aforementioned tasks. This would make the application more structured, modular, scalable and maintainable. This can be achieved by running these different containers using NVIDIA Docker Compose. However, for the scope of this project, using NVIDIA Docker Compose is too complex when it comes to networking while scaling, but has definitely more potential than running all tasks in but one Docker container. NVIDIA Docker Compose even allows assigning different GPUs to certain tasks.

8.2.4. Streaming

When judging the performance of the streaming service, there are several factors to take into account, including:

- Codec used
- Speed of the internet connection
- Resolution of the video data
- Frame rate of the video data

Every codec has its advantages and disadvantages. This project concludes that a codec based on the H264 compression technique is not suitable for streaming the video output of the Exposure Render. This is because the Exposure Render uses a stochastic ray tracing technique, called Monte Carlo rendering, which initially contains a lot of noise. This noise is highly detailed and unpredictable. The H264 codec is unable to properly decode an encoded stream of video data from the Exposure Render and causes serious graphical compression artifacts making the application nearly unusable.

MPEG-1 on the other hand is based on the JPEG compression technique. This will give a higher bitrate, but can handle way more detail than H264, thus allowing a live stream from Exposure Render without many artifacts.

8.2.5. Security

The security of the applications is of high importance due to the sensitive medical data. This means that it should be impossible for anyone to access the data without authorization. To stop unauthenticated users, users need to log in with their credentials. This service is handled by the external service Auth0. As an additional layer of security, the files are also encrypted with the Auth0 user identifier. Thus

preventing attackers from hacking the server and downloading the files. Besides that, all the lines of communication are secured with TLS and a form of authentication from Auth0.

On the other hand, there are still ways to be able to access the data without authorization. An attacker would need to steal the Auth0 user identifier as well as hacking the server and downloading the corresponding files. The attacker would then be able to decrypt the files, since the hacker has both the encryption key and the file in possession. For an attacker to be able to steal a user identifier, the attacker would need to either find a way to steal this from Auth0 or steal it from the local storage, while the user is authenticated.

8.2.6. Messaging of the applications

The communication between all the different applications in the ecosystem fully satisfy the requirements, such as streaming to the frontend.

The maintainability of the messaging can be improved by removing the Interaction Server from the Docker container. Then there could be two possibilities to solve the messaging. The first is to ensure the Connector is able to establish the TCP connection with the Exposure Render and can communicate in the Qt binary format. The other way would be to re-implement the Exposure Render communication protocol. Besides that, a re-implementation from XML to JSON for loading the settings of the Exposure Render could be considered. On top of that, a different way of updating the settings than with the GUID is also a consideration.

9

Conclusion

For the bachelor end project we have been tasked with creating a web portal for the Exposure Render. This web portal would need to allow users, among which doctors and patients, to see their CT or MRI data visualized in their browser. As a result of that, hospitals would no longer require high-end hardware on site to view the data. Instead the hospitals would need to upload their medical data to the web portal and the visualization would be shown in the User Interface of the web portal.

Through interviews with the client and the TU coach, we analyzed the different main requirements of the web portal.

- User Interface for uploading medical data
- Processing of uploaded medical data by the server
- Rendering the results of the Exposure Render in the browser
- Secure user and data management
- Port Exposure Render to Linux

During the interviews the client stressed on the importance of the security of the web portal due to the sensitive nature of the data. With the help of the TU coach and the previous prototype RemoteRendering, we researched the approach we would take on building the web portal (see appendix). Eventually we designed a first version for the architecture of the web portal. The design consisted out of four applications: a web interface, a server with a database, a container manager and an Exposure Render container. Throughout the development process changes were made to the initial design and additional requirements were added. In the end, it lead us to a fully working prototype fulfilling all the aforementioned main requirements. The current web portal consists out of a web interface, a server and a Docker Exposure Render container. Furthermore, the container contains a Connector, an Interaction Server, the Exposure Render and a stream server. The web interface is able to send HTTP requests to the server to request a container. Then the web interface is able to connect to the container in order to display the stream in the user interface and control the Exposure Render.

Bibliography

[1] A. G. Webb, *Xray imaging and computed tomography,* in *Introduction to Biomedical Imaging* (Wiley-IEEE Press, 2003) pp. 1–56.

[2] K. S. Rubin, *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, 1st ed. (Addison-Wesley Professional, 2012).

[3] J. Visser, *Building maintainable software*, 1st ed. (O'Reilly, 2016) https://www.sig.eu/wp-content/uploads/2017/02/Building$_M aintainable_S oftware_{CS} harp_S IG.pdf. H. Haas, C. Ferris, E. Newcomer, D. Booth,$ $//www.w3.org/TR/2004/NOTE - ws - arch - 20040211/.$

[5] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Ph.D. thesis, University of California, Irvine (2000).

[6] I. Fette and A. Melnikov, *The WebSocket protocol*, Tech. Rep. (Internet Engineering Task Force, 2011) https://tools.ietf.org/html/rfc6455.

[7] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol*, Tech. Rep. (Internet Engineering Task Force, 2008) https://tools.ietf.org/html/rfc5246.

[8] F. Callegati, W. Cerroni, and M. Ramilli, *Man-in-the-middle attack to the https protocol,* IEEE Security Privacy **7**, 78 (2009).

[9] M. Jones, J. Bradley, and N. Sakimura, *JSON Web Token (JWT)*, Tech. Rep. (Internet Engineering Task Force, 2015) https://tools.ietf.org/html/rfc7519.

[10] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*, 5th ed. (CRC Press, 2001) http://cacr.uwaterloo.ca/hac/.

[11] I. T. Laboratory, *Sha-3 standard: Permutation-based hash and extendable-output functions,* (2015), http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

[12] E. D. Perakslis, *Cybersecurity in health care,* The New England journal of medicine **371**, 395 (2014).

[13] A. Tamersoy, G. Loukides, M. E. Nergiz, Y. Saygin, and B. Malin, *Anonymization of longitudinal electronic medical records,* IEEE Transactions on Information Technology in Biomedicine **16**, 413 (2012).

Glossary

API Aplication Programming Interface. 7, 13, 18, 19

backend is the module of an application that is not visible to the user and contains the most functionality. 5, 9, 10, 15

bitrate is the amount of bits that can be send in a certain time period. 25, 26, 34

clipping is selectively representing visual data in a certain range. 5, 33

continuous integration is the practice of automatically merging all developer versions to a shared main version. 6, 8

CPU Central Processing Unit. 26

CT stands for Computed tomography. This is a scanning technique that uses several X-ray measurements from different angles, allowing the user to see inside the scanned object without opening it. 3, 5, 6, 25, 37

CUDA Compute Unified Device Architecture. 26, 27

DICOM Digital Imaging and Communications in Medicine. This is a medical file format. 9–11, 18, 21, 30, 34

Docker containerizing software. 6, 27, 28, 34, 37

DOM Document Object Model. 17

Exposure Render is an application developed by the TU Delft Computer Graphics and Visualization that generates photo realistic and volumetric renders of medical imagery. 3, 4, 9–11, 21, 23, 25, 27, 30, 33–35, 37

FFMpeg https://ffmpeg.org/. 11

FIFO First In, First Out. 6

frontend is the user interface of an application. 5, 9, 15, 17, 19, 25, 29, 30, 33

GPU stands for Graphics Processing Unit. This is a computer component used to handle complex graphical tasks. 5, 26, 27, 34

GUI Graphical User Interface. 6, 10, 16

GUID Globally Unique Identifier. 10, 11, 35

HTTP Hypertext Transfer Protocol. 13, 15, 18, 20, 28, 29, 31, 37

HTTPS Hypertext Transfer Protocol Secure. 28, 31

JavaScript Programming language, mostly used in web development. 9

jQuery a JavaScript library, mostly used for DOM mutation. 5

JSON JavaScript Object Notation. 15, 16, 18, 21, 23, 29, 35

JWT JSON Web Token. 29, 34

Kanban board is a visual tool to manage the progress of a team project. 7

Linux UNIX based operating system. 5, 6, 33, 37

MRI stands for Magnetic Resonance Imaging. This is a scanning technique that uses magnetic pulses to generate a visual representation of the inside of the scanned object. Only suitable for soft tissues. 3, 5, 6, 16, 25, 37

product backlog part of Scrum: container for tickets that are currently not worked on but can be implemented in future sprints. 7

product owner part of Scrum: person within the team that manages the product backlog. 7, 8

pull request is a way to tell team members that files have been changed and could be merged. 8

Python an object-oriented programming language. 5, 10, 18, 21

Qt is a UI framework, mostly used in C++ and Python. 10, 11, 23

REST Representional state transfer. 13, 18–21

RESTful Representional state transfer. 13

RGBA stands for Red Green Blue Alpha. This is a unit to define a certain color and opacity. 5

Scala is a functional programming language, running in the Java Runtime Environment. 18

scalability is the capability of a system to handle a growing amount of workload. 6, 12, 28

Scrum is a flexible agile software development method, including multiple teams working in sprints. 7, 42

scrum master part of Scrum: person within the team that manages scrum meetings. 7, 8

SSL Secure Sockets Layer. 13

TCP Transmission Control Protocol. 10, 11, 21, 23, 25

transfer function a function that definese a certain color and opacity for a given tissue density. 5, 33

UI User Interface. 5, 11, 33

URL Uniform Resource Locator. 30

version control is a system allowing the user to see changes in documents, files or folders. 8

voxel is a representation of a value on a grid in a 3D space. 5

VTK Visualization Toolkit. 30

WebSocket is an internet protocol, allowing two-way communication. 10, 13, 15, 16, 18–21, 23, 25, 28, 29

WSS WebSocket Secure. 28, 31

XML Extensible Markup Language. 35

A

Initial SIG Feedback

De code van het systeem scoort 4 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt.

Bij jullie project valt op dat de langere methodes vaak veroorzaakt worden door het mengen van data en logica. Over het algemeen is het beter om een duidelijke scheiding tussen die twee categorieën te hebben. Het bestand scanqueue.service.ts is hier een voorbeeld van, het zou beter zijn om de status code data in een ander bestand onder te brengen.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase.

B

Product Description

[1]CT and MRI scanners are common nowadays in the medical domain. The data produced by these scanners (volumetric data) is used to diagnose and track disease progression, and to plan, execute and evaluate the outcome of surgery. Although doctors are quite familiar with looking at the images slice- by slice, an inexperienced observer without a medical background will have a hard time in understanding them. The CGV group is working on tools to make these volumetric images more understandable to inexperienced observers. We have created an interactive volume rendering tool (Exposure Render) that generates photo realistic images of volumetric data, acquired from CT and MRI devices. The complex lighting- and camera effects that we implemented help to convey shape, depth and spatial distribution, and as such help patients to understand the data better.

We are currently working on a web portal that provides our volume rendering tool as a web service. The aim is to make the tool more widely available and more accessible to patients. The portal gives patients an opportunity to upload their data and to view their data online (without the need to install special hard- and/or software). A first prototype has already been built, but we are looking for ways to improve it. There are still a number of challenges that need to be addressed before we can go live with the system (including, but not limited to containerization of the renderer, scaling out on a GPU cluster and the design of the user interface). The group will be closely supported in the implementation tasks and part of the software framework exists already.

[1]Cited from: `https://bepsys.herokuapp.com/`