Software Transfer Document

MyUI

2IPE0, Software Engineering Project - Q4 (2022)

Group 8

Full Name	Student ID
Aydoslu, Eren	1414593
Boelhouwers, Daan	1457152
Farla, Richard	1420380
Guha, Nishad	1408674
Hellenbrand, Jeroen	1414844
Kustermans, Quinten	1439561
Lamme, Jeroen	1443062
Nijsten, Samuel	1422324
Radu, Adriana	1418548
Thoumoung, Yemoe	1412051

Supervisor: de Beer, Huub
Project managers: Andrade, Ricardo & Ilin, Cristian
Customer: Budziak, Guido

Eindhoven, July 5, 2022

Abstract

This document is the Software Transfer Document (STD) of MyUI, which is a web-based application that lets users generate user-specific UIs based on the underlying dataset and user permissions. The STD explains the steps to build the application, and it gives a summary of the acceptance test performed. Furthermore, the requested changes and software modifications done after the acceptance test are also described. This product has been developed by a group of students at the Eindhoven University of Technology as part of their Software Engineering Project.

Contents

Document Status Sheet

General

Document title:	Software Transfer Document	
Document identifier:	STD/0.2	
Authors:	Aydoslu, Eren	1414593
	Boelhouwers, Daan	1457152
	Farla, Richard	1420380
	Guha, Nishad	1408674
	Hellenbrand, Jeroen	1414844
	Kustermans, Quinten	1439561
	Lamme, Jeroen	1443062
	Nijsten, Samuel	1422324
	Radu, Adriana	1418548
	Thoumoung, Yemoe	1412051
Document status:	Final version	

Document Change Records

Date	Authors	Changes
25-06-2022	Jeroen Lamme	Chapter 5, 6, and 7
26-06-2022	Adriana Radu	Initial Abstract and Chapter 1
27-06-2022	Jeroen Hellenbrand	Chapter 8
27-06-2022	Nishad Guha, Jeroen Hellenbrand, Yemoe Thoumoung	Chapter 2
29-06-2022	Nishad Guha	Chapter 3, 4
29-06-2022	Daan Boelhouwers	Chapter 9
05-07-2022	Everyone	Implement supervisor feedback on all chapters

1 | Introduction

1.1 | Purpose

The Software Transfer Document explains the process of transferring the MyUI project to the customer Connect.Football B.V., represented by Guido Budziak. This includes the prerequisites needed for deployment and instructions on the steps needed for building the application. Furthermore, the STD reports on the acceptance test performed with the customer and changes made to the software during the transfer phase.

This document is intended for readers interested in deploying or further developing the MyUI application. It is assumed the reader has some experience with the command-line interface and Docker [1] to be able to build and run the application.

1.2 | Scope

MyUI is developed by a group of Bachelor's Computer Science and Engineering students for the Software Engineering Project at the Eindhoven University of Technology. This project aims to create an open-source contextual UI generator on top of Hasura [2]. These concepts will be explained in the following paragraphs.

Hasura is an open-source web application server that connects to a Postgres database [3] to give instant real-time GraphQL APIs. Hasura supports the query language GraphQL [4], a language used for querying and describing the data in an API. The permissions that users have to access data from the database are defined in Hasura.

With MyUI, users can construct their personal UI. It allows for user-level personalization by modifying data's place, size, and representation. There currently exists software built on top of Hasura that allows users to visualize their data. However, this software lacks a crucial point: user-level personalization. MotorAdmin [5], for example, is existing open-source software that is used to deploy an admin panel containing graphs, tables, and other data representation elements to analyze the data of a particular database. However, the creation of this panel, or UI, is global. This means that changes made to the UI for one user are saved for all the other users. The only level that this software operates at is at an admin level. This results in users having to make changes each time they access the UI. With MyUI, each user has a permission profile that enables viewing, editing, and/or deleting a subset of data. The user can add dashboards, containing grid views, text, and media of types images, videos, gifs, and audio files. MyUI enables users to save the changes in their UI and view the items they prefer to prioritize. MyUI is software that allows users to visualize and analyze their data quickly. Moreover, it is open-source.

1.3 | List of definitions

1.3.1 | Definitions

Term	Definition
Breadcrumb	A navigational aid that tracks and displays the visited page and its path.
Docker	A software platform that allows you to quickly build, test, and distribute apps. It delivers software in independently executable packages called containers.
GraphQL	Open-source query language used for APIs. Using this language, a complete description of the data in an API is provided.
Grid view	A table data representation element.
Hasura	Hasura GraphQL Engine. It is a web application server that generates APIs instantly and stores the data access permissions of the users.
Open-source software	Software that, depending on the license, is available to anyone to use, study, modify and distribute for any purpose.

1.3.2 | Acronyms and abbreviations

Acronym	Definition
API	Application Programming Interface
AT	Acceptance Test
CLI	Command Line Interface
SDD	Software Design Document
STD	Software Transfer Document
SUM	Software User Manual
UI	User Interface
URD	User Requirements Document

1.4 | List of references

[1] "Docker." [Online]. Available: https://hub.docker.com/

[2] "Hasura." [Online]. Available: https://hasura.io/

[3] "PostgreSQL: The World's Most Advanced Open Source Relational Database." [Online]. Available: https://www.postgresql.org/

[4] "A query language for your API." [Online]. Available: https://graphql.org/

[5] "MotorAdmin." [Online]. Available: https://www.getmotoradmin.com/

[6] Node.js, "Node download." [Online]. Available: https://nodejs.org/en/download/

[7] Yarnpkg. [Online]. Available: https://yarnpkg.com/

[8] "Install docker desktop on mac," Jul 2022. [Online]. Available: https://docs.docker.com/desktop/mac/install/

[9] "Install docker desktop on windows," Jul 2022. [Online]. Available: https://docs.docker.com/desktop/windows/install/

[10] "Ubuntu-wsl." [Online]. Available: https://ubuntu.com/wsl

[11] Craigloewen-Msft, "Install wsl." [Online]. Available: https://docs.microsoft.com/en-us/windows/wsl/install

[12] "Make for windows." [Online]. Available: http://gnuwin32.sourceforge.net/packages/make.htm

[13] "Overview of docker compose," Jun 2022. [Online]. Available: https://docs.docker.com/compose/

[14] "docker node." [Online]. Available: https://hub.docker.com/_/node/

[15] "Bash - GNU Project - Free Software Foundation." [Online]. Available: https://www.gnu.org/software/bash/

[16] "What is server-side rendering? definition and faqs." [Online]. Available: https://www.heavy.ai/technical-glossary/server-side-rendering

[17] "Advanced Load Balancer, Web Server, & Reverse Proxy," Jun 2022. [Online]. Available: https://www.nginx.com/

[18] "NGINX Docker Image." [Online]. Available: https://hub.docker.com/_/nginx/

[19] "Gitlab runner docker image." [Online]. Available: https://hub.docker.com/r/gitlab/gitlab-runner

[20] "Gitlab pipelines." [Online]. Available: https://gitlab.tue.nl/sep2022q4g08/myui-react-typescript-docker/-/pipelines

[21] "Jest." [Online]. Available: https://jestjs.io/

[22] "Eslint." [Online]. Available: https://www.npmjs.com/package/eslint

[23] "Myui gitlab repository." [Online]. Available: https://gitlab.tue.nl/sep2022q4g08/myui-react-typescript-docker

[24] NishadGuha, "Nishadguha/myui-sep-group-8." [Online]. Available: https://github.com/NishadGuha/myui-sep-group-8

[25] Eindhoven University of Technology, "MyUI: User Requirements Document," 2022.

[26] ——, "MyUI: Software User Manual," 2022.

[27] ——, "MyUI: Software Design Document," 2022.

[28] ——, "MyUI: Acceptance Test Plan," 2022.

[29] "NPM docs." [Online]. Available: https://docs.npmjs.com/

[30] “yarnpkg.” [Online]. Available: https://yarnpkg.com/package/documentation

[31] “Ant design breadcrumb.” [Online]. Available: https://ant.design/components/breadcrumb/

[32] “React grid layout.”

[33] “Ant design of React - Ant Design.” [Online]. Available: https://ant.design/docs/react/introduce

[34] “Next.js - The React Framework for Production.” [Online]. Available: https://nextjs.org/

2 | Build procedure

This chapter explains how the MyUI application can be built for development on a tester's or developer's local machine. There are two scenarios that we will consider: building the application on a machine running Windows, and a machine running a Unix-based operating system.

2.1 | Prerequisites

In order for a developer to build the application on their machine, the following software tools need to be installed in advance. This ensures that all the different containerization methods and necessary web development libraries are installed on the system as mentioned in this section.

- Node package manager (**npm**) can be installed by downloading the *.msi* or *.pkg* installer file from the official NodeJS download page [6].
- Yarn package manager [7] can be installed via the **npm** command line interface with the following command: `npm install --global yarn`
- Docker:
  - Docker can be installed for macOS and other Unix-based operating systems from their official download page [8].
  - For Windows systems, an installation of a Linux distribution in the Windows Subsystem for Linux is needed. This can be done by executing the command `wsl --install` in the PowerShell command line to first install the Linux subsystem and henceforth, downloading and installing a Linux distribution [9]. We consider the Ubuntu distribution that can be downloaded from the official Ubuntu download page [10]. The Windows Subsystem for Linux (version 2) can be enabled to use this Linux distribution with the following command: `wsl --set-version Ubuntu-<version> 2` [11]. After this step, Docker can be installed from their official windows installation page.
- The Hasura command line interface can be installed on Unix-based systems with the command: `curl -L https://github.com/hasura/graphql-engine/raw/stable/cli/get.sh | bash`.
  For Windows systems, it can be installed with the **npm** command-line interface by executing the following command: `npm install --global hasura-cli`. The installation of the Hasura command-line interface is necessary since the application build procedure requires the execution of numerous Hasura-specific commands that can only be executed by the latest stable version of the Hasura command-line interface.
- For Windows systems, the **make** command line interface needs to be installed. The installer can be downloaded from the official GNU make utility download page [12]. After a successful install, the path variable needs to be updated with: `C:\Program Files (x86)\GnuWin32\bin` and the system needs to be restarted.

2.2 | Build instructions for development

In this section, a step-by-step guide will be provided on how to build the application on a system. Additionally, some commands will be explained that facilitate the necessary setup procedures for the application.

2.2.1 | Steps for building the application

1. **Clone the respective git repository**: The git repository of the project can be cloned into the user's machine by executing the command:
   `git clone https://github.com/NishadGuha/myui-sep-group-8.git`
2. **Access project directory**: Subsequently, the user needs to access the appropriate project directory by executing the command: `cd myui-sep-group-8`.

3. **Execute build command**: After successfully cloning the repository and accessing the project folder, the user should execute the command: **`make setup`** in the terminal of the project directory. This command automates all other steps necessary for development. In the following section, a breakdown of the different steps involved in the **`make setup`** command is shown. Which can be used in case errors occur during the build process.

2.2.2 | Breakdown of the `make setup` command

The **`make setup`** command comprises the following build steps:

1. **Building the docker containers**: The first step of this command executes the command: **`docker-compose -f docker-compose.yml -f docker-compose-dev.yml up -d --build`**. The execution of this command leads to all the necessary docker containers being built for the application to run on the user's system. On inspection of the **`docker-compose.yml`** file, one can discern that this command builds three containers:
   - **`react-app`**: A container running the front-end React application.
   - **`graphql-engine`**: A container running the Hasura GraphQL engine.
   - **`postgres`**: A container running the Postgres database.
2. **Installing node dependencies**: The second step of this command executes the command: **`npm i --force --silent`** which installs all necessary node dependencies on the user's system. The **`--force`** and **`--silent`** flags are used so that there are no unexpected blockers during the build process and the user does not receive an extensive list of warnings during the build procedure.
3. **Installing yarn dependencies**: The third step of this command executes the command: **`yarn install --ignore-engines --silent`** which installs all necessary yarn dependencies on the user's system. The **`--ignore-engines`** and **`--silent`** flags are used for the same reason as in the previous step, in order to ensure a smooth build process for the user.
4. **Applying Hasura metadata**: The next step of this command makes sure that Hasura is set up correctly for the user. The command executed in this step is **`cd hasura && hasura metadata apply`**. This first accesses the **`hasura`** directory in the project folder where all metadata files reside in the **`/metadata`** sub-directory. Subsequently, the command **`hasura metadata apply`** ensures that all metadata files are implemented in the Hasura GraphQL engine container that was built in the first step.
5. **Applying database migrations**: This step of the command makes sure that the database is populated with some basic demo data. This includes the creation of three users *admin*, *editor*, and *user* with credentials *sep2022*, *connectedfootball2022*, and *myui2022* respectively. This gives the user a way for immediate access to the application before having to access the Hasura console and making their own users. The command executed for this step is **`hasura migrate apply --up all --all-databases`**. This makes sure that all database migration (*.sql*) files in the **`/migrations`** sub-directory are executed and the database is populated as intended.
6. **Ensuring metadata consistency**: The last step of this build process executes the command: **`hasura metadata reload`**. This command acts as a safeguard and makes sure that there are no inconsistencies in the metadata that was applied to the user's Hasura GraphQL engine instance. This step ensures that there are no further errors in the Hasura console setup and the user can immediately access the MyUI front-end without having to make changes in the Hasura console.

2.2.3 | Commands supported by MyUI

The commands stated below act as secondary commands that are supported by the project. Some of these commands allow the user to access other parts of the application such as the Hasura console, while some commands are stated that can come in handy for faster build times as well as restarting and stopping the application, which act as aids for developers. The explanations for these commands are as follows:

- **`make setup`**: This command combines several other commands to set up the application's front-end and back-end. It builds and runs the necessary containers, installs all the required dependencies, applies the present metadata to the Hasura GraphQL engine, and lastly, executes all database migrations in order to populate the database. Executing this command takes about 300 seconds.

- `make build-dev`: This command builds and runs all the necessary containers to set up a development environment on the local machine after the initial setup steps that have been executed by the aforementioned `make setup` command. Executing this command takes about 150 seconds.
- `make run-dev`: This command only runs all the necessary containers for the development environment on the local machine. This command does not build the containers again and hence is a quick way to run the development environment if the containers have been built once before. Executing this command takes about 3 seconds.
- `make console`: This command is used to access the Hasura console where database changes can be made at a more granular level. This command accesses the `/hasura` sub-directory in the project folder and launches the Hasura console on the port 9695 of the user's system. Executing this command takes about 5 seconds.
- `make restart`: This command can be used as a quick way to restart the docker containers that are used in the project. This is a useful development tool that can quickly restart the application in case there are any discrepancies or inconsistencies of code changes being out of sync unexpectedly. Executing this command takes about 8 seconds.
- `make rebuild`: This command is a more comprehensive way of restarting the application. This command first stops and removes all running docker containers and then builds the containers again. This has a similar purpose as the previous command, however, it takes a much longer time to finish execution. Executing this command takes about 200 seconds.
- `make down`: This command stops and removes all running containers. Executing this command takes about 8 seconds.

2.3 | Explanation of Docker build steps

The application is built and installed inside docker containers. These containers are run as services and stitched together in a Docker network using *docker-compose* [13]. In this section, the different steps involved in installing the MyUI application in the docker containers will be explained.

2.3.1 | Multi-stage build procedure

A multistage Docker build method enables the development of an application followed by the removal of any irrelevant development tools from the container. The finished container is smaller as a result of this method. In this project, a multi-stage build procedure is implemented for two main reasons:

- In order to compile Docker containers that are built from different docker images.
- In order to differentiate the build procedures for development and production environments.

2.3.2 | Development environment

The installation procedure for the development environment is based on the **node** Docker image on Docker Hub [14].

Installation Steps

1. **Add a new *admin* user**: First a user called admin is created by executing the command: `RUN useradd -ms /bin/bash admin`.
2. **Set working directory and copy dependencies**: Within the Docker container, the working directory needs to be set in order for the subsequent commands to execute properly. This can be done by executing the command: `WORKDIR /app`. Henceforth, the file that comprises all the dependencies used in the project is copied into the previously created working directory by executing the `COPY package.json .` command.
3. **Install all necessary dependencies**: All dependencies required to run and render the application need to be installed inside the Docker container by executing the command: `RUN npm install --force`. The `--force` flag is used so that there are no extensive warnings or unexpected errors that halt the build process.

4. **Change owner of the working directory**: In this step, we set the user as the owner of the working directory and give the user all read permissions by executing the command: `RUN chown -R admin:admin /app`. This is done since there are parts of the application that need to be read from configuration files in the working directory.

5. **Copy code base**: The next step is to copy the code base from the local directory to the container's working directory. This is required in order to have all the files necessary to render the front-end of the application. The command executed for this step is `COPY . ..`

6. **Give user write permissions**: Hereafter, the created user (admin) is given write permissions in the working directory by executing the command: `RUN chmod 755 /app`. This is necessary as there exist functionalities in the MyUI application that write to a configuration file in the working directory directly from the application.

7. **Set user and expose network port**: In this penultimate step, the created user is set as the default user of the bash command-line interface of the container [15]. After this, the port that is used to view the client-side of the application is exposed from the container to the local system. The commands executed for this step are: `USER admin && EXPOSE 3000`.

8. **Launch development build of the client-side application inside the container**: In this step, a development environment of the application is launched inside the container as its final step. The command executed for this step is: `CMD ["npm", "run", "dev"]`. The reason for using *CMD* instead of *RUN* is that this command gets executed as a default last step of the container and has no immediate changes to the files inside the working directory. Instead, the command launches a development build that is used to view the client-side of the application.

2.3.3 | Production environment

In the build process for the production environment, the idea is to make a build setup with as much server-side rendering [16] as possible to reduce any discrepancies on the client-side of the application with regards to inconsistencies in rendering dynamic UI elements. Therefore, in this setup, there is a production build of the application that is produced by the framework and this is then hosted to the user on an NGINX server [17].

An NGINX server is used because it provides a consistent networking layer, reconfiguration without requiring restarts, and an included load balancer. These features have not been tested by the team since the client did not require a deployment of the application. However, the presence of such a setup ensures a smoother deployment process for future developers by providing a containerized setup that can easily be run on a server.

Installation Steps
First, we pull from the **node** docker image hosted on Docker Hub [14].

1. **Set working directory and copy dependencies**: Just as in the setup of the development environment, the working directory needs to be set in the Docker container in order for the subsequent commands to execute properly. This can be done by executing the command: `WORKDIR /app`. Subsequently, the file that contains all the dependencies used in the project is copied into the previously created working directory by executing the `COPY package.json .` command.

2. **Install all necessary dependencies**: All dependencies required to run and render the application need to be installed inside the docker container by executing the command: `RUN npm install`.

3. **Copy code base**: The next step is to copy the code base from the directory on the host machine to the container's working directory. This is required in order to have all the files necessary to render the front-end of the application. The `COPY . .` command is executed for this step.

4. **Generate a production build of the application**: After the initial steps, a production build version of the application is generated with the command: `RUN npm run build`. This command optimizes the project files and dependencies to be as much server-side rendered as possible.

5. **Host application on NGINX server**: Finally, we pull from the NGINX Docker image on Docker Hub [18]. Within this image, the production version of the software that was just built is put

in the appropriate folder of the NGINX container by executing the following command: `COPY --from=build /app/build /usr/share/nginx/html`.

2.4 | Continuous integration

This section will describe the setup used for a Continuous Integration (CI) system. This is a commonly used tool for development that ensures that changes to the code do not break the testing coverage or violate the coding standards and are thus automatically and safely integrated into the software project. This provides a benefit over developing, building, and testing solely in a local environment. This section is provided to explain how the CI system was implemented on Gitlab for this application.

The Continuous Integration system uses Docker containers. Using Docker containers enables the system to be transferred to alternate CI systems without making any changes. In order to host the continuous integration pipeline, a Gitlab runner service is added to the docker-compose build. This service bases itself on a publicly hosted GitLab runner image [19] and facilitates in implementing of any future CI-CD setups by providing a portable and virtualized service that accompanies the build procedure of the application and is open to extension.

Prerequisites
The project folder of the application on the user's machine must have a copy of the CI file for this project. This file can be obtained by a developer with appropriate institutional access from the following Gitlab link: https://gitlab.tue.nl/sep2022q4g08/myui-react-typescript-docker/-/blob/CI-CD/.gitlab-ci.yml.

Pipeline
The continuous integration and continuous development pipeline implemented performs two main CI jobs. All these jobs are automated and the user does not need to perform any other steps. The execution of the continuous integration pipeline can be triggered by pushing new commits to the git repository. The execution of the continuous integration pipeline can be inspected on the Gitlab CI/CD page [20] as follows:

1. **Unit tests**: In order to execute this step, first all necessary dependencies are installed with the command: `npm install`. Subsequently, all unit tests are run with the help of the *jest* unit testing package [21]. Running all unit tests with this package with the configuration setup in the project beforehand ensures that test coverage is generated within the CI step.
2. **Linting**: Just as in the previous step, first all dependencies are installed. Henceforth, the *eslint* package [22] is used as a tool for spotting trends and patterns in the code and reporting findings. Any parts of the code that do not adhere to the specified configuration in the project are reported in this CI step.

2.5 | Possible errors

In this section, the errors that may occur during the build procedure will be discussed.

- In case the prerequisites stated in Chapter 2.1 are not met, an error can occur that the certain package is not recognized as an internal or external command. In this case, the package has to be installed before the building can resume. One example of this is the error: `yarn: command not found`. This indicates that the *yarn* package manager has not been installed on the user's machine. This error can be fixed by installing yarn on the user's machine by executing the command: `npm install -g yarn`.
- While applying the Hasura metadata, this happens during the execution of the command: `hasura metadata apply` which is executed as one of the steps of the `make setup` command, the following error can occur: *cannot read config: invalid config version.* If this error occurs, the Hasura command-line interface needs to be updated to the newest stable version. This can be done by executing the command: `npm install --global hasura-cli`. This command installs the latest version of the Hasura command-line interface.
- The `make` commands automates the execution of multiple commands in the project. It is possible that a make command fails to execute fully because of an unfinished exit state of a previous command. The user has two options in this situation:

- Run the `make` command again.
- Inspect the `Makefile` in the project folder, find the specific make command that caused the error and execute the rest of the commands of that make command manually.

Performing one of these steps should fix the issue.

3 | Installation procedure

The application is built and installed inside docker containers. These containers are run as services and stitched together in a docker network using *docker-compose* [13]. In this section, the different steps involved in installing the MyUI application in the docker containers will be explained.

3.1 | Installation steps

In this section, the way of verifying a successful build as well as the procedure to run the application on an end user's system will be covered.

3.1.1 | Verify build

First, the following can be done in order to verify a successful build procedure of the application:

1. Open a terminal window (Command Prompt or PowerShell for Windows, default terminal for macOS/Linux).
2. Access the directory that comprises the project files. This can be done by executing the command: `cd myui-sep-group-8`. It is important to note that the path specified in this instruction can be different for different users.
3. Execute the command: `docker ps`
4. The output should show a list of three containers. This list should comprise namely:
   - `react-app`: A container running the front-end React application.
   - `graphql-engine`: A container running the Hasura GraphQL engine.
   - `postgres`: A container running the Postgres database.

This output indicates that the build procedure has successfully created the necessary docker containers.

3.1.2 | Access application

After verifying that a successful build, the following steps can be followed in order for an end-user to be able to view the application:

1. Open Google Chrome version 101 or above, Mozilla Firefox version 100 or above, Safari version 15 or above, or Microsoft Edge version 100 or above.
2. Subsequently, enter the URL: http://localhost:3000 into the browser's address bar.

Accessing this link will lead the user by default, to the login page of MyUI.

3.1.3 | Access Hasura console

The user is also enabled to access the back-end console of Hasura, where the user can have more granular access to the database connection and setup. The steps to access this console are as follows:

1. Open a terminal window (Command Prompt or PowerShell for Windows, default terminal for macOS/Linux).
2. Access the directory that comprises the project files. This can be done by executing the command: `cd myui-sep-group-8`. It is important to note that the path specified in this instruction can be different for different users.
3. Execute the command: `make console`. This command launches the back-end Hasura console on the port 9695 of the user's system.
4. Open Google Chrome version 101 or above, Mozilla Firefox version 100 or above, Safari version 15 or above, or Microsoft Edge version 100 or above.
5. Subsequently, enter the URL: http://localhost:9695 into the browser's address bar.

After following the above steps, the user should see the Hasura console in their browser window.

4 | Configuration item list

The MyUI deliverables that will be sent to the customer are described in the section that follows. The transfer of all documents will be done in PDF format. On the GitLab repository [23], the source code is made available for institutional access. Additionally, a public repository on Github [24] has a mirror of the master branch of the Gitlab repository.

4.1 | Documentation

The customer will receive the following documentation as deliverables:

- User Requirements Document [25]
- Software User Manual [26]
- Software Design Document [27]
- Software Transfer Document (this document)
- Acceptance Test Plan [28]

4.2 | Project files

As mentioned above, the source code of the project is made available for institutional access on the GitLab repository. [23] The final deliverable version of the application is contained in the **master** branch. The continuous integration pipeline setup is contained in the **CI-CD** branch.

The public repository on Github [24] has a mirror of the master branch from the Gitlab instance. There are no additional branches on the Github repository since it adds no functional value for the client as the other branches only exist to facilitate a development team.

An overview of the project files is as follows:

- The `/pages` sub-directory contains all the code for the different pages of the application as well as the internal API routes in the `/pages/api` sub-directory.
- The `/components` sub-directory contains all the code for the different user interface components and front-end queries of the application.
- The `/styles` sub-directory contains all the CSS (Cascading Style Sheets) code for the front-end styling of the application.
- The `/utils` sub-directory contains code that provides functions that are used as utility functions in different parts of the code base. This means that these functions are used to perform tasks that were easily abstracted and isolated in order to aid reuse such as making a call to a NextJS API route.
- The `/consts` sub-directory contains constant objects and values that are used multiple times in different parts of the code base.
- The `/hasura` sub-directory contains all the files that the Hasura GraphQL engine bases its metadata and database migrations on. The metadata and database migrations can be found in the `/hasura/metadata` and `/hasura/migrations` directory respectively.
- The `/config` sub-directory contains any files that act as configuration files for the application. There are parts of the application, such as the global settings, that are allowed to read from and write to files in this folder.
- The `/__tests__` sub-directory contains all the files that comprise test cases to test the functionality for different parts of the code.
- The `/__mocks__` sub-directory contains files that comprise mock functions and objects for external dependencies of the test cases that are specific to the testing library used in this project.

- The `/public` sub-directory contains front-end specific files that are used in the user interface of our application. These include JSON files containing translations and image files.
- There are certain files that together facilitate the setup of the Docker containers as well as configuration files for tools used in the project such as typescript, the testing library (jest), the code quality linting library (eslint), the framework NextJS, the react front-end framework (babel), the translation library (i18-next), environment variables and lastly, the readme and installation instructions. To elaborate, they are:
  - Docker-specific files
    - `docker-compose.yml`
    - `docker-compose-dev.yml`
    - `docker-compose-prod.yml`
    - `Dockerfile.dev`
    - `Dockerfile.prod`
    - `Dockerfile.hasura`
  - `Makefile` for setting up the project and secondary application setup-related commands.
  - Configuration files
    - `babel.config.js`
    - `jest.setup.js`
    - `jest.config.ts`
    - `next.config.js`
    - `next-i18next.config.js`
    - `tsconfig.json`
    - `.eslintrc.json`
  - `.env.local` for the environment variables.
  - `README.md`
  - `Installation.md` for installation instructions

5 | Acceptance test report summary

The acceptance test (AT) for MyUI was performed on June 24th, 2022, and lasted from 14:00 until 15:50. The test procedures outlined in version 0.2 of the Acceptance Test Plan for MyUI [28] were used for this AT. This document was signed by the customer before the acceptance test was performed, thereby agreeing that the acceptance tests and test procedures presented in the ATP would be used for the acceptance test. The customer, the two project managers, the supervisor, and nine team members were present during the acceptance test. The customer completed all 12 test procedures, thereby executing all 53 acceptance tests. According to the ATP's definition of passing, all acceptance tests were completed satisfactorily. The customer consequently agreed that the AT was carried out successfully. The customer did request some changes to be made to the code, with regards to consistency and formatting, which will be discussed in Section 7.

6 | Software Problem Reports

No software problems were found during the acceptance test and the software transfer process.

7 | Software Change Requests

During the acceptance test of MyUI no detrimental errors presented themselves. However, the client did suggest some changes that could be made to MyUI. These changes will be listed here.

- It takes a long time to build the application. The command `RUN chown -R admin:admin /app` should not be run after the command `RUN npm install`.
- Currently both NPM [29] and Yarn [30] are used. Since they serve the same functionality as package manager, only one should be used.
- The format and positioning of the error message are inconsistent. In any situation, the error messages should be visible on the screen without the need to pan.
- The insert and delete row buttons should be fixed on top of the grid views they are part of.
- To maintain consistency throughout the UI, the text and position of buttons within prompts should be changed.
- It should only be possible to click on the OK button in the add row menu when all rows are filled.
- When a row has blank fields in required columns, the user shouldn't be able to save the row while still editing it.
- While the navigation sidebars are closed, users should still be able to see where they are within the application.
- When leaving edit mode even when no changes have been made to the dashboard, the user is occasionally requested to save the changes. This prompt should only be shown when changes were made.

8 | Software Modification Reports

This chapter elaborates on the change requests from Chapter 7 and explains which requests were implemented.

8.1 | Knowing where you are within MyUI

To help users know where they are in the application, we implemented two new features: breadcrumbs [31] and URL paths. This modification is visible in Figure 8.1 for both situations where the user is currently viewing a base table or a dashboard.

For example, for the local version of MyUI, when viewing a base table *Customer*, the URL will be changed to localhost:3000/basetables/customer and you will see a breadcrumb *Base tables / Customer*. If you view a dashboard *test-dashboard*, the URL will be changed to localhost:3000/dashboards/test-dashboard and you will see a breadcrumb *Dashboards / test-dashboard*.

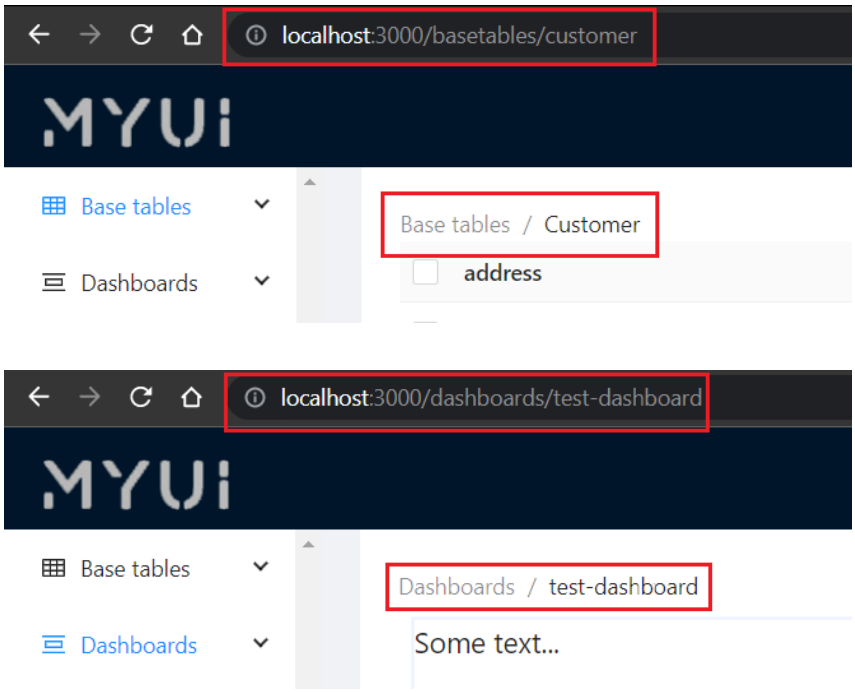

Figure 8.1: MyUI modifications

9 | Unit test report

9.1 | Unit test location and execution

This chapter describes where the unit tests for the MyUI application can be found, as well as how these unit tests can be executed.

9.1.1 | Location of tests

This section describes the location of the unit tests in the code tree of the MyUI application. The unit tests for the MyUI application can be found in the **/__tests__** subdirectory of the root folder of the project, called **myui-sep-group-8**. Each test file, **[componentName].test.tsx** contains unit tests corresponding to the react component named **[componentName]**. Here **[componentName]** should be replaced with the name of a react component defined in the **components** subdirectory of the root folder of the project. The only component file for which no corresponding test file exists is **Loader.tsx**, nonetheless, the component defined in this file is still tested as it is used by other components. Finally, there are two test files for which the name does not correspond to a component. These files are **signin.test.tsx** and **index.test.tsx**, these files contain unit tests for the sign-in page defined in **/pages/auth/signin.tsx** and the home page defined in **/pages/index.tsx** respectively, where both of the aforementioned file paths are relative to the root directory.

9.1.2 | Running the tests

This section describes the procedure for running the unit tests for the MyUI application. This procedure assumes that the user is using a Linux device. Furthermore, it is required that the user has already installed the required modules by executing the commands `npm i --force` and `yarn install --ignore-engines`. The procedure consists of the following steps:

1. Open a terminal.
2. Navigate to the root directory of the MyUI application.
3. Execute: `yarn test`

After following these steps, the results of the execution of the unit tests will be displayed in the terminal. These results include a listing of all test suites, indicating for each test suite whether it passed or failed, as well as a coverage report.

9.2 | Test report

This section lists the results and the coverage of the unit tests for the MyUI application. Furthermore, the reasons for low coverage for certain components and pages are explained.

9.2.1 | Unit test results

This section lists the individual tests for each react component or page, as well as whether this test passed (✔) or failed (✗).

AddDeleteRowMenu

Test suite	Result
Buttons only render when a user has permission to perform the corresponding action	
Buttons render succesfully when insertable and deletable	✔
Buttons don't render when not insertable and not deletable	✔
Only add button renders when insertable but not deletable	✔
Only delete button renders when deletable but not insertable	✔
Add menu works correctly	
Add row menu renders	✔
Query is sent when add confirm button is clicked	✔
Delete menu works correctly	
Delete row menu renders	✔
Delete row menu does not query when cancel button is clicked	✔
Query is sent when delete confirm button is clicked	✔
Alert is shown when delete confirm button is clicked	✔
Add and delete for users table	
Encrypt is called when new passwords are inserted into the users table	✔
Correct query is sent when deleting entry from users table	✔

AppHeader

Test suite	Result
Render Header	
Renders AppHeader correctly when viewing a dashboard	✔
Renders AppHeader correctly when editing a dashboard	✔
Renders AppHeader correctly when viewing a basetable	✔
Renders AppHeader correctly when userConfig is undefined	✔
Interact with Header	
Clicking on the gear toggles editmode	✔
Clicking on the user icon opens a menu	✔
Clicking on the logout option in the menu calls signOut	✔
Clicking on the english option in the menu changes the language to english	✔
Clicking on the dutch option in the menu changes the language to dutch	✔
Logs the error when changing the language throws an error	✔
Does not show global settings menu if user does not have admin role	✔
Does show global settings menu if user does have admin role	✔
Calls setGlobalSettingsModalState when an admin clicks on the global settings	✔

BaseQueries

Test suite	Result
Tests for configurationQuery	
Configuration query executed succesfully with default configuration	✔
Configuration query executed succesfully with new configuration	✔
Configuration query logs an error	✔
Tests for UpdateUserConfiguration	
User configuration is updated	✔
User configuration is updated with undefined userConfig	✔
Tests for tableQuery	
Correctly performs an introspection query	✔

BaseTable

Test suite	Result
It loads a basetable	
Renders basetable successfully	✔

Dashboard

Test suite	Result
Dashboard renders correctly	
Renders	✔
Dashboard renders correctly with gridview element	✔
Calls functions onDrop	✔
Testcases for the opening of the editModal by double clicking an element	
Opens editElementModal when an element is doubleclicked	✔
Doesn't open editElementModal when an element is doubleclicked in non-editmode	✔
Testcases for deleting / adding elements form / to the dashboard	
Deletes the dashboardElement from the dashboard configuration when the deletebutton is clicked	✔

EditElementModal

Test suite	Result
Render editElementModal	
Renders editElementModal when visible is true	✔
Does not render editElementModal when visible is false	✔
Query edit modal tests	
Input and submit a legal query	✔
Input and fail to submit a string that is not a query	✔
Input and fail to submit an query containing illegal characters	✔
Input and fail to submit a query consisting of an empty string	✔
Cancel editing a query	✔
Static edit modal tests	
Input and submit text	✔
Input and fail to submit illegal text	✔
Cancel editing text	✔

EditModeSider

Test suite	Result
EditModeSider functions correctly	
Renders	✔
Renders static and gridview buttons	✔
Calls saveDashboardChanges when clicked on save button	✔
Downloads the dashboard configuration file when clicked on download dashboard button	✔
Sets the dataTransfer properties when a gridview element is dragged from the sidebar	✔
Sets the dataTransfer properties when a static element is dragged from the sidebar	✔

EditRowsQueries

Test suite	Result
Tests for checkPermissions	
Does not set permissions when query returns no data	✔
Sets permissions correctly when only allowed to edit a basetable	✔
Sets permissions correctly when allowed to do everything, viewing a basetable	✔
Sets permissions correctly when allowed to do everything, viewing a non-editmode dashboard	✔
Sets permissions correctly when allowed to do everything, viewing a dashboard in editmode	✔
Tests for updateRowQuery	
Handles undefined query input error	✔
Handles edit permission error	✔
Handles foreign key violation error	✔
Handles errors returned by hasura (not an edit or foreign key error)	✔
Works when no error occurs	✔

EditableCell

Test suite	Result
Test cases for EditableCell component	
Renders succesfully with inputType === number	✔
Renders succesfully with inputType !== number	✔
Renders succesfully with undefined title	✔
Renders succesfully with editing === false	✔

GlobalSettings

Test suite	Result
Global Settings Modal	
Pressing OK closes the modal	✔
Pressing CANCEL closes the modal	✔
Changing the display setting to Media calls updateUserConfig with MEDIA	✔
Changing the display setting to URL calls updateUserConfig with URL	✔

GridView

Test suite	Result
GridView functions as expected	
Renders	✔

index

Test suite	Result
Tests for viewing of basetables	
Navigate to basetable	✔
Tests for Edit mode	
Navigate to dashboard and enter and exit editmode without changes	✔
Navigate to dashboard and enter and exit editmode with changes	✔
Navigate to dashboard and enter and exit editmode with changes	✔
Navigate to dashboard, enter editmode, and cancel exiting editmode	✔
Navigate to dashboard, enter editmode, and cancel exiting editmode	✔
Tests for Adding and removing dashboards	
Add a new dashboard and delete it again	✔
Cancel adding a new dashboard	✔
Cancel deleting a dashboard	✔

ManageDashboardsModal

Test suite	Result
Renders ManageDashboardsModal correctly	
Renders when isVisible=true	✔
Does not render when isVisible=false	✔
SetVisible is called on cancel when adding	✔
SetVisible is called on cancel when removing	✔
Functions are called on OK when adding a dashboard	✔
No functions are called on OK when dashboardtype !== ADD and dashboardtype !== DELETE	✔
Adds a dashboard name to the dashboardNames and userConfig	✔
setUploadState is called when using file upload	✔
Removes a dashboard name from dashboardNames and userConfig	✔
Removes a dashboard name from dashboardNames and userConfig and calls setWorkspaceState when deleting the dashboard currently displayed in the workspace	✔
Checks name validity when adding dashboard	✔
Does not remove dashboards when clicked on OK	✔
Checks name validity when removing a dashboard	✔

NavigationSider

Test suite	Result
Testcases for NavigationSider	
Renders NavigationSider correctly	✔
Renders BaseTable names	✔
Renders add/remove dashboard icons and dashboard names	✔
Calls baseTableOnClick when clicked on a basetable name	✔
Calls dashboardOnClick when clicked on a dashboard menu item	✔
Still renders successfully when no basetables are present	✔

signin

Test suite	Result
Signin	
Renders a signin form	✔
Signs in to application	✔
Does not sign in with empty credentials	✔

StaticElement

Test suite	Result
Static element adapts to content	
Renders text correctly	✔
Renders png image correctly	✔
Renders jpg image correctly	✔
Renders jpeg image correctly	✔
Renders gif correctly	✔
Renders mp3 correctly	✔
Renders mp4 correctly	✔
Snapshot tests for static element adapts to content	
Renders text correctly	✔
Renders png image correctly	✔
Renders jpg image correctly	✔
Renders jpeg image correctly	✔
Renders gif correctly	✔
Renders mp3 correctly	✔
Renders mp4 correctly	✔

TableData

Test suite	Result
TableData functions as expected in BaseTable form	
Renders basetable when dataState is ready	✔
Displays row and column count correctly	✔
Renders loader when data is loading	✔
Renders save button, cancel button, and editable cells when clicked on edit button	✔
Hides editable cells when clicked on cancel and confirmed cancel	✔
Updates cell values when clicked on save after editing	✔
Values remain unchanged when saving after not changing anything	✔
Fails to updates cell values when clicked on save after editing with empty input field	✔
Shows an alert when setAlert is called by AddDeleteRowMenu	✔
Calls setUserConfigQueryInput when clicked on sort	✔
Table data function as expected in users table form	
Encrypt is called when password is edited	✔
Encrypt is not called when a value that is not a password is edited	✔
Table data function as expected in dashboard form	
Renders dashboard table when dataState is ready	✔
Updates selected rows accordingly when checkbox is clicked	✔
Informs the user when dataState is ready and no data was found	✔
AddDeleteRowMenu is not rendered when in editmode	✔
Calls setUserConfigQueryInput when clicked on sort	✔

TableDataQuery

Test suite	Result
Tests for TableDataQuery	
When error is returned	✔
Query returns an empty table	✔
BaseTable query returns a non-empty table	✔
Basetable query not yet defined in userconfig	✔
Non-baseTable Query returns a non-empty table, ordering not yet defined	✔
Non-baseTable Query returns a non-empty table, ordering already defined	✔

Workspace

Test suite	Result
Renders Workspace correctly	
Renders default workspace	✔
Renders empty workspace	✔
Renders base table	✔
Renders dashboard in editmode	✔
Renders dashboard in displaymode	✔

9.2.2 | Coverage report

In this section the coverage for all files, as reported by Jest, is listed. Furthermore, it is explained why not all files have a coverage of 100%.

Coverage report

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	94.02	88.12	93.25	94.98	
components	95.62	88.88	97.16	96.12	
AddDeleteRowMenu.tsx	100	100	100	100	
AppHeader.tsx	100	100	100	100	
BaseQueries.tsx	100	100	100	100	
BaseTable.tsx	100	100	100	100	
Dashboard.tsx	80.59	90	91.66	81.81	207-214,237-256
EditElementModal.tsx	100	100	100	100	
EditModeSider.tsx	100	100	100	100	
EditRowsQueries.tsx	100	100	100	100	
EditableCell.tsx	100	100	100	100	
GlobalSettings.tsx	100	100	100	100	
GridView.tsx	100	100	100	100	
Loader.tsx	100	100	100	100	
ManageDashboardsModal.tsx	100	100	100	100	
NavigationSider.tsx	100	100	100	100	
StaticElement.tsx	100	100	100	100	
TableData.tsx	100	100	100	100	
TableDataQuery.tsx	73.78	51.06	81.25	77.89	26-34,196-262
Workspace.tsx	100	100	100	100	
consts	100	100	100	100	
Workspace.tsx	100	100	100	100	
defaultConfiguration.ts	100	100	100	100	
demoDashboardConfig.ts	100	100	100	100	
demoUserConfig.ts	100	100	100	100	
enum.tsx	100	100	100	100	
hasuraProps.ts	100	100	100	100	
inputSanitizer.tsx	100	100	100	100	
pages	96.79	92	95	96.09	
index.tsx	96.79	92	95	96.09	461-475
pages/auth	50	0	25	57.89	
signin.tsx	50	0	25	57.89	35,133-146
utils	25	0	0	25	
encryption.ts	25	0	0	25	10-19
updateUserConfig.ts	25	0	0	25	8-19

As can be read from the coverage reports, the files that do not have 100% coverage are **Dashboard.tsx**, **TableDataQuery.tsx**, **index.tsx**, **signin.tsx**, and the files contained in the **utils** directory. The reasons for this below 100% coverage are as follows:

- **Dashboard.tsx**: The uncovered lines in this file are related to the drag and drop events from dashboard elements. For this, we used the React-Grid-Layout module [32], which has a custom `onDrop` function. We could not find a way to test all of our code related to this `onDrop` function using Jest.
- **TableDataQuery.tsx**: The uncovered lines in consist of functions that are passed on as properties to the `Table` component from Ant Design [33].
- **index.tsx**: The uncovered lines in index.tsx consists of an export of a function named `getServerSideProps` which is used for server-side rendering by the Next.js framework [34], as such it does not need to be tested.

- **signIn.tsx**: The lines 133-146 consist of an export of a function named `getServerSideProps` which is used for server-side rendering by the Next.js framework [34], as such it does not need to be tested. Line 35 is a call to the `signIn` from the NextAuth.js module. This line is not covered since `signIn` is mocked during testing.
- **utils/**: The files in this directory contain API requests to the back-end. These files have low coverage since the back-end is mocked during testing.