Docker Web Interface

Applicazioni e Servizi Web

Razvan Florian Vasile - 000725342 {razvanflorian.vasile@studio@unibo.it}

01 October 2023

Contents

List of Figures

List of Tables

1 Introduction

1.1 Domain

What is Docker? Docker is a containerization platform that allows to package and distribute applications and their dependencies in a standardized unit called a "container".

What is a container? A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

Def. Dockerfile. A Dockerfile is used to define the image of a single container (that is the template for creating a container starts off by creating an instance of an image). It provides a set of instructions for building a Docker image, including specifying the base image, copying files into the image, installing dependencies, and configuring the container environment.

Example. It follows an example of how to create an image starting from a Dockerfile. After the image has been created, it is possible to create an arbitrary number of independent containers.

Listing 1: Sample Dockerfile

```
# base environment for our image
FROM node:7-alpine

# container exposes this port (accessible from the host)
EXPOSE 1337

# copy files from host to the container
ADD hello.js /opt/

# start a service
CMD ["node", "/opt/hello.js"]
```

Def. docker-compose file. Docker Compose is a tool that helps define and share multi-container applications. With docker-compose, it is possible to create a YAML file to define each service and with a single command, to launch all services or tear them all down.

Example. Below is an example of a docker-compose.yml file. Firstly, each docker-compose.yml file needs either an image obtained from the Docker Hub repository to instantiate containers or alternatively, a Dockerfile defined locally which serves the same purpose.

Listing 2: Sample docker-compose file

```
version: '3.3'

services:
  # defines an arbitrarily service name
  node:
    build:
      # defines locally where the Dockerfile is located
      context: app
      # alternatively, using a remote image
      image: my-remote-img
    ports:
      # ports expose in the <host>:<container> format
      - "1337:1337"
```

Relevance with the project. In this application, these files represent the building blocks for managing containers. Essentially, the application parses the *docker-compose* files stored at a predefined location (/**server-side**/**docker-projects**) and allows them to perform operations such as launching all services, tear them all down, getting logs of running containers and so on.

1.2 Benefits of using a GUI

Firstly, the purpose of this application is to serve as a graphical user interface to managing docker containers and services. There exist various benefits of using a GUI as opposed to using only the CLI.

- Ease of Use for Beginners: GUIs provide a more visually intuitive way to interact with containers, making it easier for beginners.
- Visual Representation: GUIs typically offer a visual representation of containers. This can be helpful for quickly grasping the architecture across different components.
- Monitoring and Management: GUIs include built-in monitoring and management tools that provide real-time information about container resource usage and logs.

2 Requirements

Requirements Engineering. User stories and design tasks were created during the initial phase of the project. These proved to be essential to make the transition to wireframes and prototypes using Figma. Subsequently, these mock-ups were transformed into code.

2.1 User stories

Definition. User stories are a fundamental concept in Agile software development. They are informal descriptions of a software feature or functionality from the perspective of an end user. User stories are used to capture the "who," "what," and "why" of a software requirement in a simple and understandable format.

Purpose. They are useful because they offer flexibility and simplicity. On top of this they are user centric which offers value in terms of being able to prioritize and then incrementally develop the software. By using them, the software engineers obtain continuous feedback from stakeholders and end users, which helps in refining and improving the product over time.

Below is a summary of all the user stories that were created. These have a 1 to 1 correspondence with the requirements that were crafted during the proposal of the project.

ID	Priority	Type	As a <type of user>	I want to <perform some task>	So that I can <achieve some goal>
1	Low	Home Page	First-time visitor	Quickly understand what Docker-UI offers	Decide if the application is relevant to me.
2	Low	Documentation Page	Inexperienced user	User-friendly documentation page	Easily learn what features the website offers.
3	Medium	Login/Signup Page	New member	Seamlessly load previous sessions' content	Be satisfied and make decisions based on previous data.
4.1	High	Dashboard Page	Registered user	Centralized dashboard to interact with a container's actions	Manage created containers during their execution.
4.2	High	Dashboard Functionality	Docker user	Customize projects maintained by DockerUI	Manipulate docker containers via docker-compose files.
5	High	Logger's Page	Experienced user	See logging messages within the browser	Easily debug running containers.
6	Medium	Docker Hub	Product owner	Simply distribute Docker UI	Dispense the application quickly to interested users.

Table 1: User stories

2.2 Design tasks

Definition. Design tasks are specific, actionable items that are part of the software development process. These tasks are focused on creating the visual and interactive design elements of the software via Figma.

ID	Type	Design Steps
1	Home Page	✓ 1.1. Display a clear and catchy headline that describes our value proposition. ✓ 1.2. Include a captivating hero image or illustration that resonates with the brand. ✓ 1.3. Add a login button that guides users to explore further.
2	Documentation Page	✗ 2.1. Setup and host documentation pages via Github Pages (see 2.2). ✓ 2.2. Setup README.md on Github with information about the project. TODO ✓ 2.3. Organize the documentation in an easily understandable format. ✓ 2.4. Provide information on how the website is organized.
3	Login/Signup Page	✓ 3.1. Offer a clear login form with fields for id and password. ✓ 3.2. Offer a clear registration page with fields for id, password and email. ✗ 3.3. Allow users to sign up using social media accounts for convenience (not initial requirement). ✗ 3.4. Provide password recovery options and ensure a secure login process (not initial requirement).
4.1	Dashboard design	✓ 4.1.1. Design a clean and organized dashboard layout using Figma [a]. ✓ 4.1.2. Display account information and settings through intuitive and easy to use icons and widgets. ✓ 4.1.3. Ensure easy navigation between different sections or features.
4.2	Dashboard Functionaity	✓ 4.2.1. buildAll(path) - Used to build all services given a Docker Compose file. ✓ 4.2.2. logs(services, path) uses configServices(path) - Show logs of service(s). ✓ 4.2.3. kill(path) - Force stop service containers. ✓ 4.2.4. stop(path) - Stop running containers without removing them. ✓ 4.2.5. start or restartAll(path) - Restart all services. ✓ 4.2.6. upAll(path) - Build, (re)creates, starts, and attaches to containers for all services. ✓ 4.2.7. down(path) - Stops containers and removes containers, networks, volumes, and images created by up.
5	Logger's Page	✓ 5.1. Redirect container logging messages to the browser to easily read them. ✓ 5.2. Create an intuitive interface that allows users to easily navigate across the available features. ✓ 5.3. Decide on the technologies necessary to setup the real time interaction between the client and server.
6	Docker Hub	✓ 6.1. Gain the technical expertise needed to setup a Docker Hub release through existing examples. ✓ 6.2. Setup dependent containers that need to be installed along-side Docker-UI. ✓ 6.3. Exhaustive tests on multiple machines to ensure that the release works consistently across different OSes. TODO

Table 2: Design tasks

[a]Figma Design Page: here

Purpose. Design tasks involved the specific activities of creating wireframes and mockups via Figma. They defined the user interface (UI) and user experience (UX) aspects. They ensure that the user stories are translated into tangible design elements.

Comparison of the two. **User stories** provide the high-level requirements and goals, guiding what needs to be achieved. They give the 'what' and 'why'. **Design tasks**, on the other hand, provide the 'how' and 'what it looks like'. They take the user stories and break them down into design elements that fulfill the user's needs.

3 Design

3.1 MVC

The MVC design pattern was used alongside the MEAN technological stack. Below is a quick overview of both and how they are organized.

- Model (MongoDB). The model layer is responsible for the application's data logic and storing and retrieving data from back-end data stores. In this project the data schema and models were defined using Mongoose (a ODM - Object Document Mapper).
- View (Angular). The view layer provides the UI necessary to interact with the application. In this project, it is handled by Angular to create a frontend UI. The Angular services are used to interact with the Express.js controllers via HTTP requests.
- Controller (Express.js). The controller layer contains the application logic necessary to facilitate communication across the application, acting as an interface between the view and model layers. Routes were defined as endpoints in the **/roots/server-side/routes** folder.

Benefits. Some of the benefits of using MVC include organizing the code to enforce a clear separation of concerns. The view is responsible for presenting the data to the user, the model represents the application's data and business logic and the controller handles user input and manages the flow of data between the model and view. A visual representation of how MVC works is shown in Figure 1. Moreover, how the project's code is organized under MVC is shown in Figure 2.

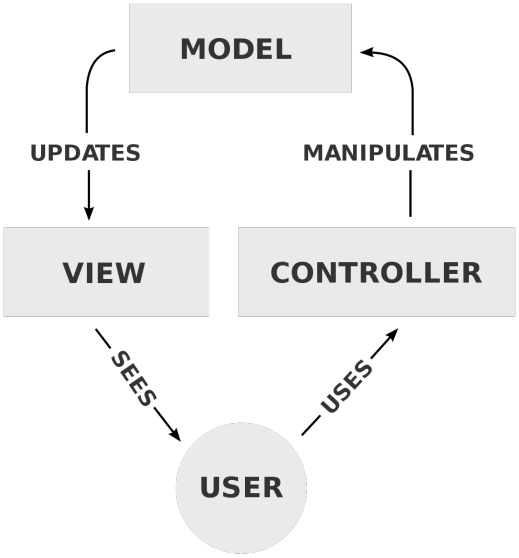


Figure 1: MVC Pattern

3.2 Responsive design using Bootstrap

Bootstrap is used to create responsive and visually appealing interface design. This is done using Bootstrap's grid system which allows to create responsive layouts by dividing the page into 12 columns.

A snippet of code that uses this design can be seen in *table.component.html* or *docs.component.html*.

Listing 3: Responsive Design HTML Code

```
<div class="container">
  <div class="row">
    <div class="col-sm-6">
      <!-- Content for small screens (>=576px) -->
    </div>
    <div class="col-md-6">
      <!-- Content for medium screens (>=768px) -->
    </div>
  </div>
  <div class="row">
    <div class="col-lg-12">
      <!-- Content for large screens (>=1200px) -->
    </div>
  </div>
</div>
```

Generally, this layout works by creating a parent *container* element that stores the entire layout of the page. Then, initiating with a new row, the page is separated into a grid of 12 columns. The designer must choose how many such columns must be used by each element of the page row-wise, that is for each specific row. When the elements of a row have been decided, it is possible to create a new row for which the designer has control once again of new elements by indicating them across the 12 available columns and the process repeats itself.

4 Technologies

In terms of dependencies, the requirements that were considered *non negotiable* are MongoDB, ExpressJS, Angular and NodeJS - that is the MEAN stack. The extent to which each technology was used will be detailed in the following section.

Apart from the general requirements imposed by the project guidelines, there were also used some other third party libraries. These had the role of improving the UI, providing security and making the website responsive.

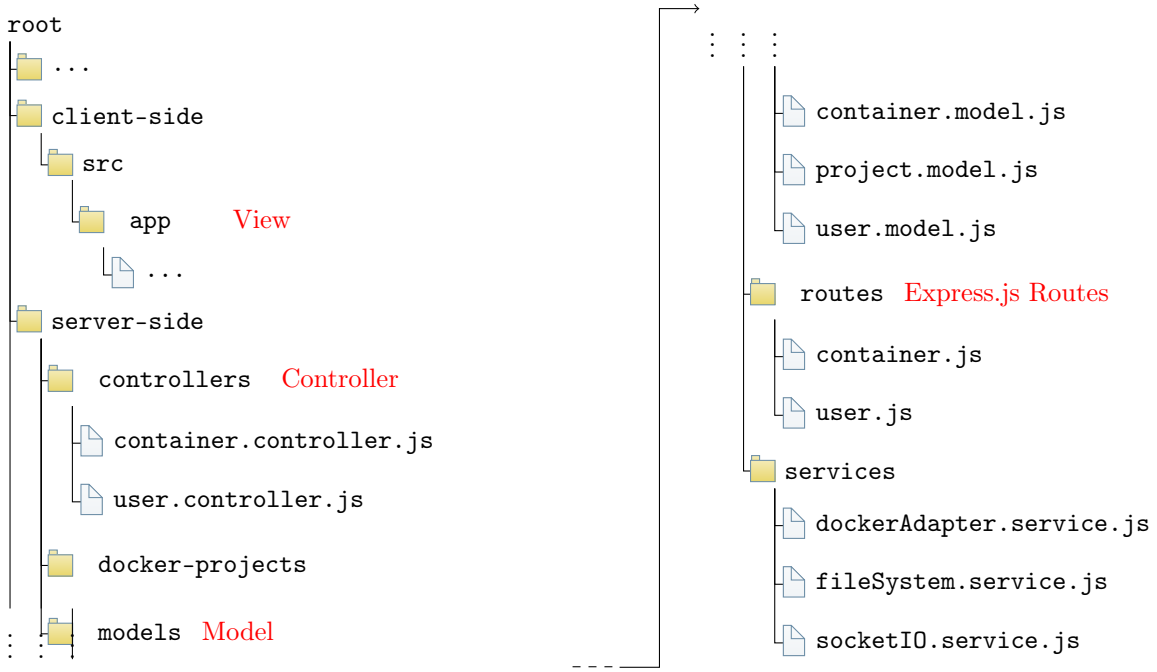


Figure 2: Code organization under MVC

4.1 The MEAN stack

The MEAN stack is a popular technology stack for building web applications. MEAN is an acronym that stands for MongoDB, Express.js, AngularJS (or Angular) and Node.js. Each component of the stack serves a specific purpose in web application development.

1. MongoDB is a NoSQL database that stores data in a flexible, JSON-like format called BSON (Binary JSON). It's designed for scalability and can handle large amounts of data.

2. Express.js is a web application framework for Node.js. It simplifies the process of building robust, scalable web applications by providing a set of features and tools for handling HTTP requests, routing, middleware. Express.js represents the backbone of our application, responsible for serving data to the frontend represented by the Angular framework.

3. Angular: is a front-end JavaScript framework developed by Google. It's used for building dynamic and interactive user interfaces. It provides a structured way to organize front-end code, offers features like two-way data binding, dependency injection, and routing, and makes it easier to create single-page applications (SPAs).

4. Node.js is a server-side runtime environment that allows you to run JavaScript on the server. It's known for its non-blocking, event-driven architecture, making it well-suited for building real-time applications and APIs. Node.js is used as the server runtime in a MEAN stack application.

4.2 Libraries

There were used various third party libraries in order to accomodate all the requirements detailed in Table 2. On the one hand, some of them relate to updating the web UI in real-time, attained via *SocketIO*. On the other hand, some of them relate to security, either of the individual passwords stored in the database, via *bcrypt*, or of securely exchanging and verifying (authentication) claims exchanged between the client and server - in the form of *JsonWebToken*. Moreover, I'll make remarks on the subject of accessibility made possible through functionality provided by *Angular Material* and the *Toastr* library.

4.2.1 SocketIO

Socket.io is a JavaScript library that simplifies real-time, bidirectional communication between clients (browsers) and servers. It was used for displaying information related to a container's log information. This means that the user gets real-time information about a container's status which facilitates debugging of any potential problems that may arise. Moreover, aside the logger's information, the messages regarding the execution of any command is shown in the project's information page.

4.2.2 bcrypt

BCrypt is a password-hashing function based on the Blowfish cipher. Besides incorporating a salt to protect against rainbow table attacks [3], bcrypt is an adaptive function: over time, the iteration count can be increased to make it slower, so it remains resistant to brute-force search attacks even with increasing computation power [1].

Crucially, bcrypt is used to encrypt user's login password prior to storing it in the database. It works as follows:

- To load a new password into the system, the user selects a password. This password is combined with a fixed-length salt value.
- Salt value is a pseudorandom or random number. The password and salt serve as inputs to a hashing algorithm to produce a fixed-length hash code. The hash algorithm is designed to be slow to execute in order to thwart attacks. The hashed password is then stored in the database with the corresponding user ID.
- Bcrypt uses the ID to index the password and retrieve the plaintext salt and the encrypted password. The salt and user-supplied password are used as input to the encryption routine. If the result matches the stored value, the password is accepted.

Salt serves three purposes:

1. It prevents duplicate passwords from being visible in the database.
2. It greatly increases the difficulty of offline dictionary attacks.
3. It becomes nearly impossible to find out whether a person with passwords on two or more systems has used the same password on all of them.

The relevant code which implements the approach outlined above can be found in *user.controller.js* and is reported below. This shows how to create and store a user in MongoDB.

Listing 4: User registration

```
const newUser = new User();
const salt = bcrypt.genSaltSync(10);
const hash = bcrypt.hashSync(req.body.password, salt);
newUser.username = req.body.username;
newUser.password = hash;
newUser.email = req.body.username;

newUser.save().then((result) => {
  requestResult.json({
    message: 'Successful registration.',
    data: result,
  });
})
```

4.2.3 JsonWebToken

Authentication. It is a means of representing information between two parties in a way that can be easily verified and trusted. These tokens were used for user's authentication which is required initially when the application is run. Moreover, it is also used for maintaining user authentication status across browser refreshes.

Listing 5: JWT guard example

```
canActivate(route: ActivatedRouteSnapshot, state:
    RouterStateSnapshot) {

  // Check whether code runs in web browser
  if (typeof window !== "undefined") {
    const token = localStorage.getItem("currentUser");
    if (token !== null) {
      return !this.jwtHelper.isTokenExpired(token);
    }
  }
  // not logged in so redirect to login page with the return url
  this.toastr.info("Please log in to proceed");
  this.router.navigate(["/login"], { queryParams: { returnUrl:
      state.url } });
  return false;
}
```

This checks if the current user is authenticated, in which case it verifies that the token has has not expired. In case it expired, it redirects to the login page, otherwise allows to see the desired resource.

4.2.4 Angular Material libraries

Angular Material is a UI component framework for Angular applications. It provides a set of pre-built, customizable UI components that can be used to create responsive and visually appealing web applications. This library was used extensively across the project.

The library offers a wide range of UI components such as buttons, forms, navigation elements, and dialogs. Some of the components which were used are summarized below:

- **MatButtonModule**: For creating buttons with various styles.
- **MatInputModule**: To create input fields and form controls.
- **MatToolbarModule**: For building app headers and toolbars.
- **MatIconModule**: To use Material Design icons.
- **MatDialogModule**: To create dialogs and modals.
- **MatTableModule**: Mat table module.

Theme. The library allows for easy theming and styling of the application, which helps maintain a visually appealing and consistent design across all components. The application uses the default theme called *deeppurple-amber.css* found in the *material* package.

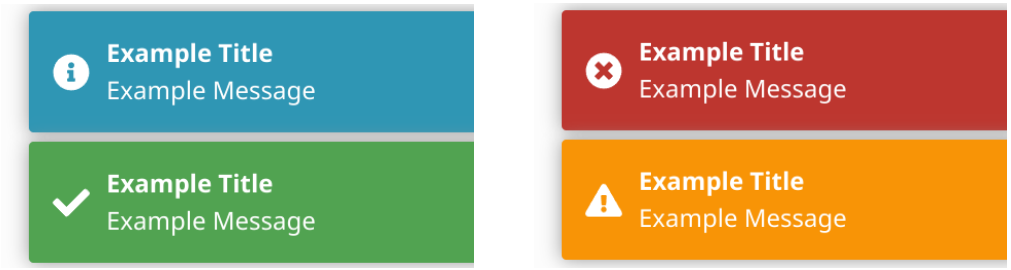
Accessibility. Angular Material places a strong emphasis on accessibility. The components are designed to be accessible by default, ensuring that the application is usable by people with disabilities. Some examples in which accessibility was kept in mind include:

- Descriptive Alt Text. For images, the **alt** attribute conveys the content or function of an image. This was used across the application images.
- HTML elements designed for accessibility. Angular encourages the use of semantic HTML elements, such as <button>, <input>, and <a>. These elements have built-in accessibility features, making it easier for screen readers and other assistive technologies to interpret the content.
- Keyboard Navigation. Angular applications should be navigable using only a keyboard. It was ensured that focus is managed correctly, and users can interact with all interactive elements using keyboard keys like Tab, Enter, and Space.
- Error Messaging. Clear and concise error messages are displayed in case of errors. Furthermore, some **input** fields become red when they are mandatory, giving visual feedback against content which must be supplied.

4.2.5 Toastr Service

Ngx-Toastr is a JavaScript library used for displaying non-blocking notifications to users in the form of pop-up toasts or messages. These toasts was used to provide feedback, alerts, success messages, or any other kind of notification to the user.

Below are a few examples of how these messages look like:



(a) Info/Success diagrams (b) Error/Warning diagrams

Figure 3: Toastr diagrams

5 Code

5.1 User login and registration

5.1.1 Sequence Diagrams

Login. It follows a discussion of the sequence of steps required for user authentication. This is depicted in Diagram 4.

1. User. The user inserts the credentials into the browser.
2. Frontend (Angular). Makes an http authentication request. The http service used to perform the request is provided by the Angular framework.
3. Backend (ExpressJS). The server queries MongoDB for user's credentials and verifies them against the provided ones and as a result returns success or failure.
4. Database (MongoDB). Used to store user authentication information. The data is encrypted using bcrypt as explained in Section 4.2.2. This library is used in order to provide additional security against Rainbow Dictionary Attacks [3].

Registration. Moreover, Figure 5 depicts the registration process, which is very similar to the diagram shown in Figure 4. The entities described in the previous list apply here as well.

These sequence diagrams illustrate a simplified login/registration process, however the actual implementation varies and will also involve error handling, encryption of data, generation of the JSON Web Token, as it was discussed in Section 4.2.2 and 4.2.3 respectively.

5.1.2 Class Diagrams

Figures 6 and 7 show two class diagrams which depict the relationship between the different modules and classes across the two most important modules of the application: the User and the Dashboard respectively.

There are similarities between the two modules. Both of them make use of a base module which encompasses information that is reusable around the entire application. This module is extensively shared across and enables information such as the dashboard, guards, models, toolbar or services to exist in only one place. This kind of modularity provides advantages in terms of flexibility, scalability and maintainability of existing code as it was shown by the two diagrams that were recently mentioned.

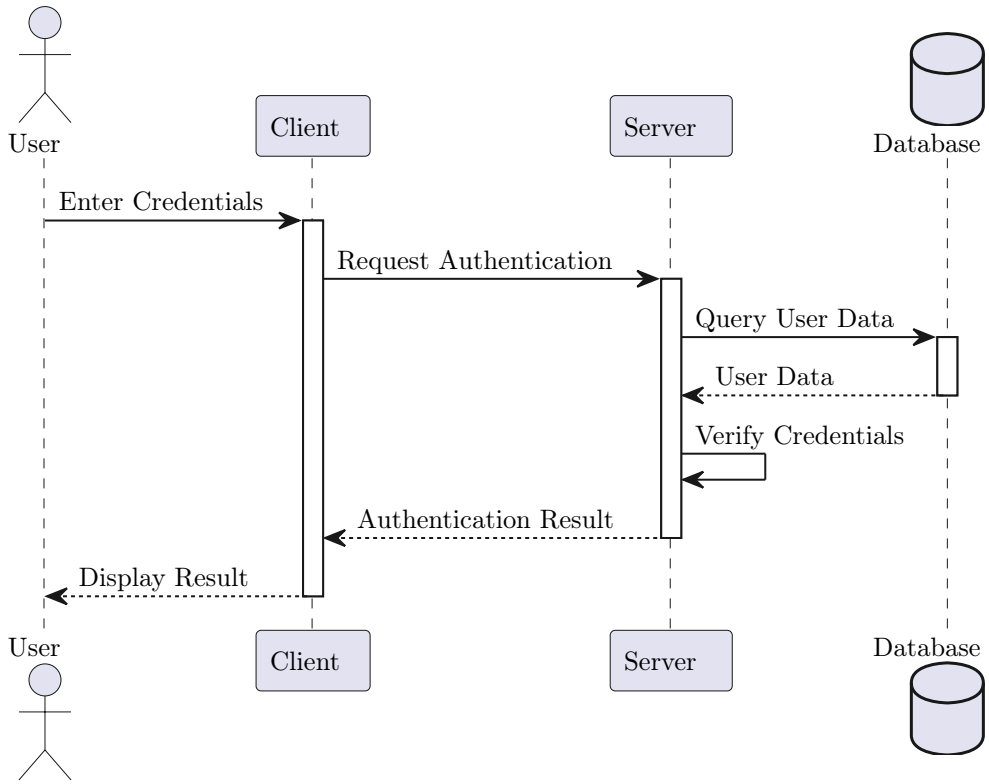


Figure 4: Login Sequence Diagram

User Module In Figure 6, the structure of the project becomes clearer. Firstly, there is the *UserModule* which spawns the entire chain of dependencies. There are two high level components which spring from it (*RegistrationComponent* and *LoginComponent*), which in turn make use of a shared *UserService*, residing in the *BaseModule*. Moreover, these components make use of DTOs[1] to manipulate data around, making use of Typescript's strong typed system.

Dashboard Module On the other hand, it follows of description of Figure 7. The dashboard module is slightly more complex, since the processes which it must describe are more involved. However, the general layout described in the previous paragraph still holds. The module can be described through three components:

1. ManagementComponent. It contains the functionality related to managing a selected projects. This includes commands such as building, starting or closing all services within a project.
2. TableComponent. It displays the projects that were detected inside the projects' root directory (*server-side/docker-projects*) as a Material table. This page makes use of the modal defined in TableDialogComponent.
3. TableDialogComponent. Used to display information about a specific project. It aids usability as the application provides a quick way of inspecting a specific project's build instructions.

5.2 Back-end Code organization

As it is shown in Figure 8, the back-end is organized around the MVC pattern and some of its features and advantages were described in Section 3.1.

- Project's root contains configuration files such as **package.json** to determine project dependencies and the **.gitignore** file.
- The root directory contains a Dockerfile which is used to create an image which is subsequently deployed to Docker Hub.

[1]DTO - Data Transfer Object

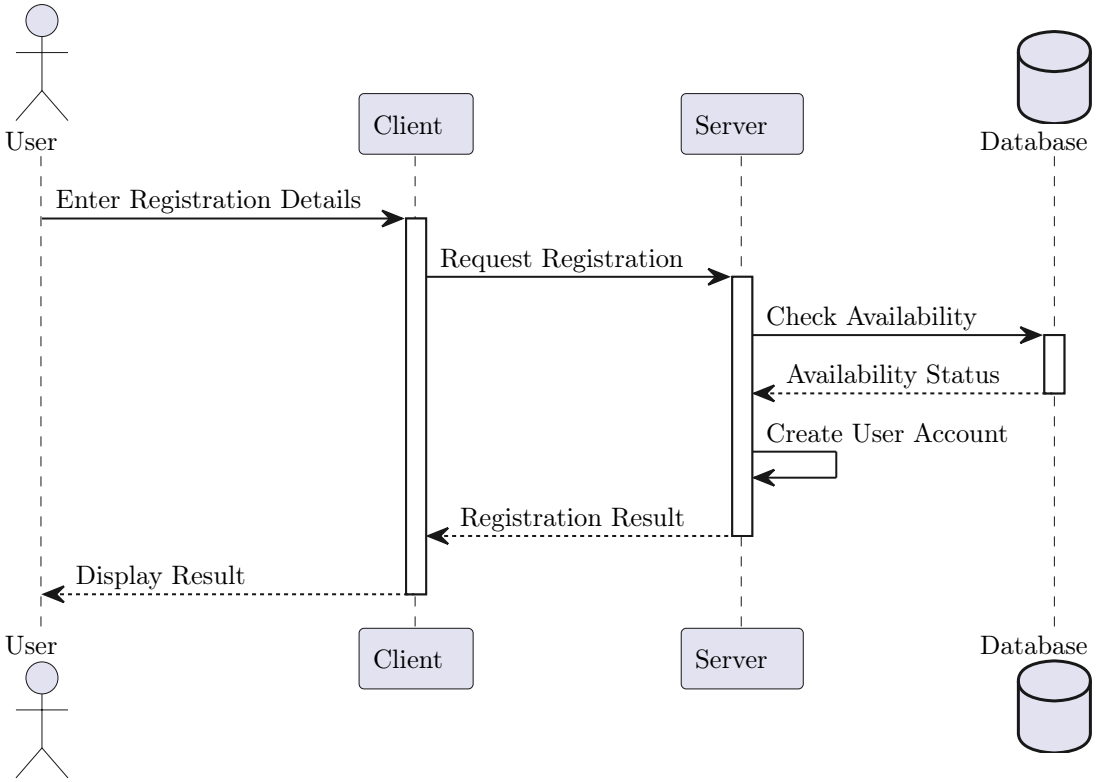


Figure 5: Registration Sequence Diagram

- The code is organized under the MVC pattern and is organized as follows:
  - **controllers**: define logic for handling HTTP requests and responses.
  - **models**: define the data structures and schemas for MongoDB documents.
  - **routes**: define API routes and link them to corresponding controllers.
  - **services**: reusable code across the different files.

5.3 Frontend code organization

The folder **frontend** holds all client side Angular code, is shown in Figure 9 and is organized as follows:

- **Root** directory contains:
  - **angular.json** configuration file for Angular
  - **Dockerfile** for creating an image which is used for deployment via Docker Hub
  - **package.json** used to define NodeJS dependencies.
- **config** folder contains the environment variables used for MongoDB, encryption, server and socket IO endpoints, etc.
- **src** directory contains the component that bootstraps the application, all other components spawn from here.
- **src/base** contains components, services, models and guards that are shared across the whole application. This folder is composed of entities that do not partake to some specific feature but are shared across different areas of the application. This could be the registration or login features or the management of containers.
  - **Dashboard**: contains left hand side menu which can be toggled on and off.

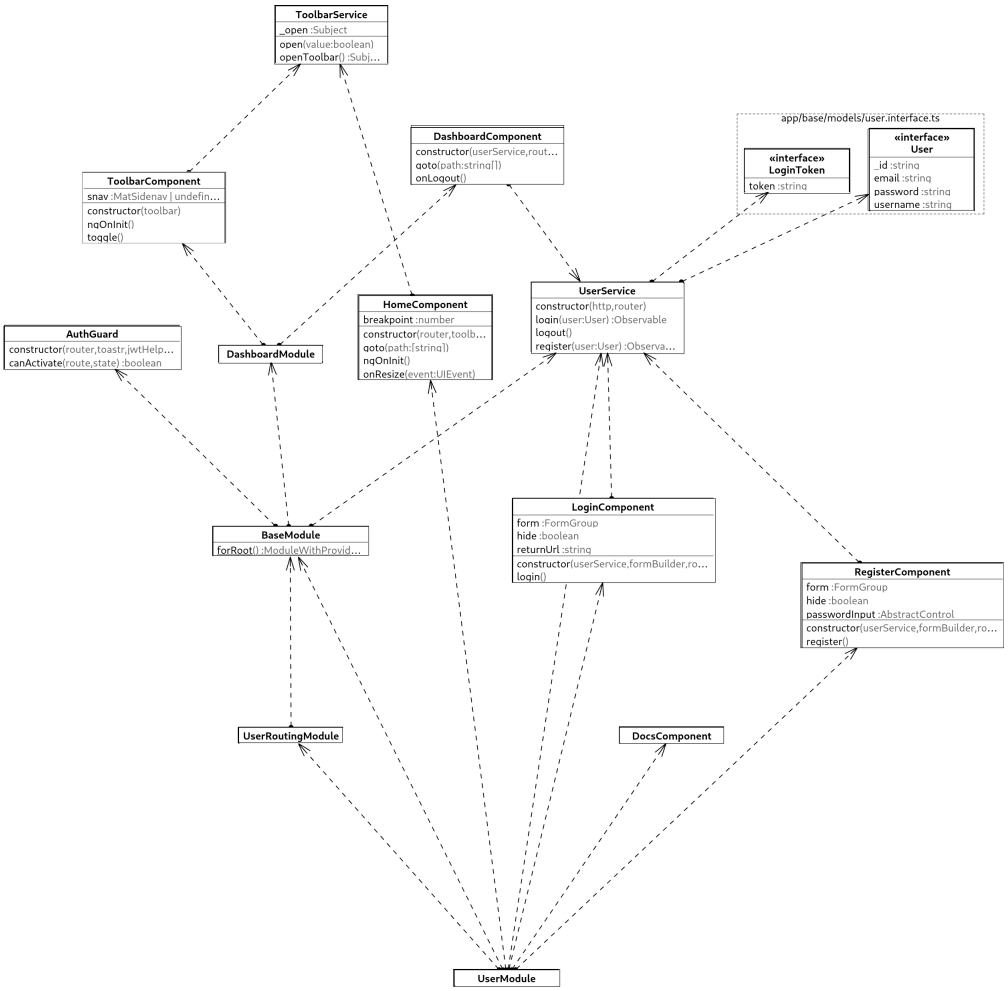


Figure 6: Angular User module class diagram

- **Authentication Guard**: ensures that the user is logged in prior to accessing specific resources.
- **Models**: models used for hard typing data that arrives from the server.
- **Services**: needed to share data across components.
- **Toolbar**: this is the bar from the top area of the screen. Used to toggle the dashboard navigation list.

- **src/components**: define the main features: login/registration, management of the containers, home page and documentation page.

6 Test

As far as testing goes, there were used two techniques. First one entails the creation of prototypes using Figma at the beginning of the project and the other one involved using questionnaires to gauge the robustness of the final result after the project has been developed.

6.1 Figma

What is Figma? It is a collaborative web application for interface design. The feature set of Figma focuses on user interface and user experience design, with an emphasis on real-time collaboration, utilising a variety of vector graphics editor and prototyping tools [2].

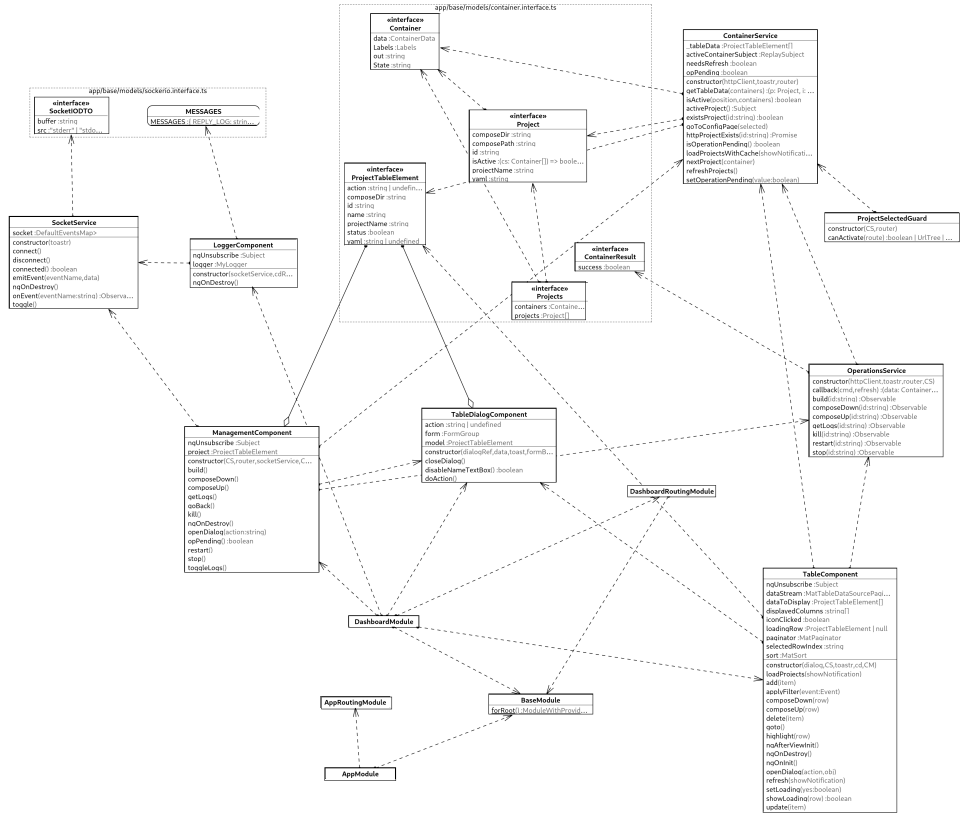


Figure 7: Angular Dashboard module class diagram

Some of its key features include **Vector Editing**, **Prototyping**, **Component Libraries** and **Plug-ins**, all of which were used to some degree.

Figma as a design tool. Figma was the main technology used for prototyping the system. As a consequence of creating user stories and design tasks which were discussed in Sections 2.1 and 2.2, there were made multiple Figma diagrams, which drove the development of the application. There were times when initial sketches did not make it into the final version (i.e. design decisions which were subsequently abandoned) and as a result Figma proved to be a useful tool for planning the application. Mock-ups were used for assessing the user experience, proving to have a significant impact into how the final result was tailored.

The Figma mock-ups can be found online[2]. However, I will display the main diagrams in this document to illustrate and comment upon some of the relevant design decisions. The diagrams can be seen at Figures 10, 11 and 12.

6.1.1 Benefits of using Figma

As it has been suggested, developing the prototypes helped highlight subpar design decisions which were improved timely, in essence gauging relevant areas of improvement by creating a tangible high level description of the application. In essence, it helped guide important decisions before losing sight of the bigger picture by delving into the particularities of how application details shall be developed.

In particular, initially the login/registration pages were designed as it is shown in Figure 13 but the final implementation is displayed in Figure 10. Essentially, Figma helped individuate cluttered areas, which represented areas of improvement (i.e. the brightness and contrast of the registration page were particularly high) and were taken care of by making modifications such as changing the background color or adjusting the size of the widgets used for login/registration.

[2]The Figma project together with all sketches used to devise the mock-ups can be found here.

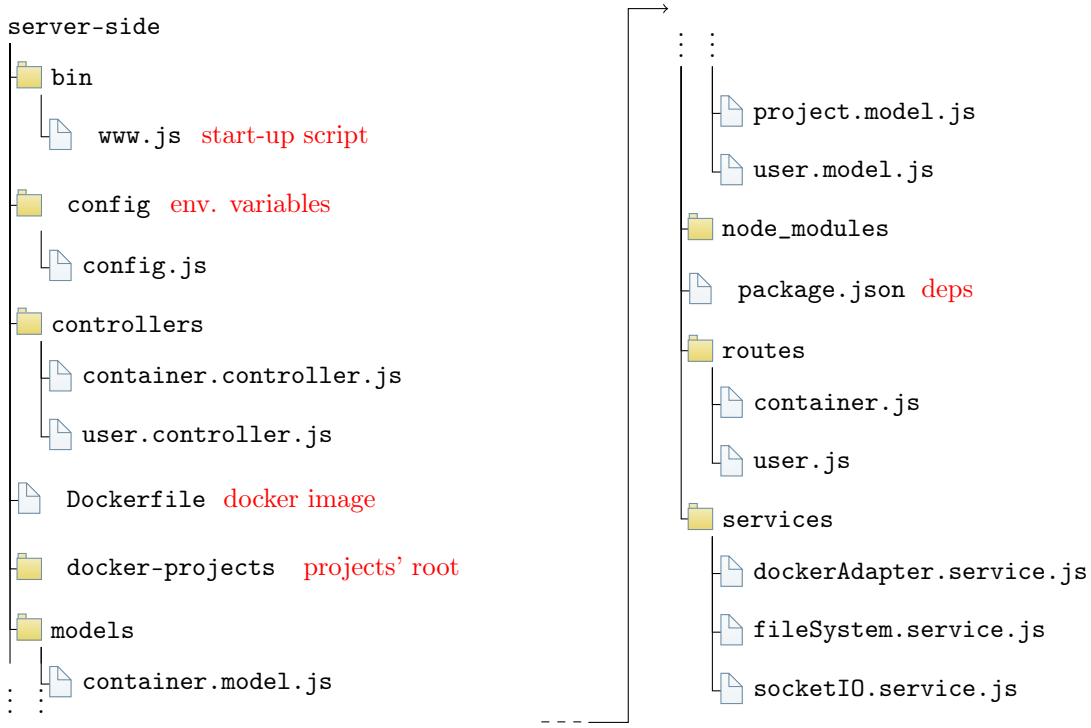


Figure 8: Server side code organization

Similarly, the logger's display area (Figure b of 13), was changed as it was initially too cluttered and as a result was moved to its own page. The separation enhanced and hopefully made for a more robust and intuitive browsing experience.

6.2 Questionnaires: UEQ

I utilized an User Experience Questionnaire (UEQ) [3] which assessed and measured users' enjoyment of using the application. In the next section, I'll describe these results which can be summarized in two diagrams.

6.2.1 Results

On the right-hand-side of Figure 14, it is shown the number of people that took part in the survey and across each line there is the choice each person made against each particular evaluated characteristic.

There were two areas that were considered unclear or cluttered and as a result changes were made, pre-modification illustrations may be seen in Figure 13.

- Login/Registration pages were considered ambiguous.
- Container Management page was considered cluttered.

The second diagram, shown in Figure 15, shows the average of each evaluation metric, that is the sum across all surveys for a specific characteristic divided by the number of total completed questionnaires. These questions were arranged in the same order as shown in the official documentation [4].

7 Deployment

7.1 Docker Architecture

Diagram 16 shows the high level architectural view of the application. This can be described as follows:

[3] The link of the UEQ is available here.
[4] UEQ homepage: https://www.ueq-online.org/.

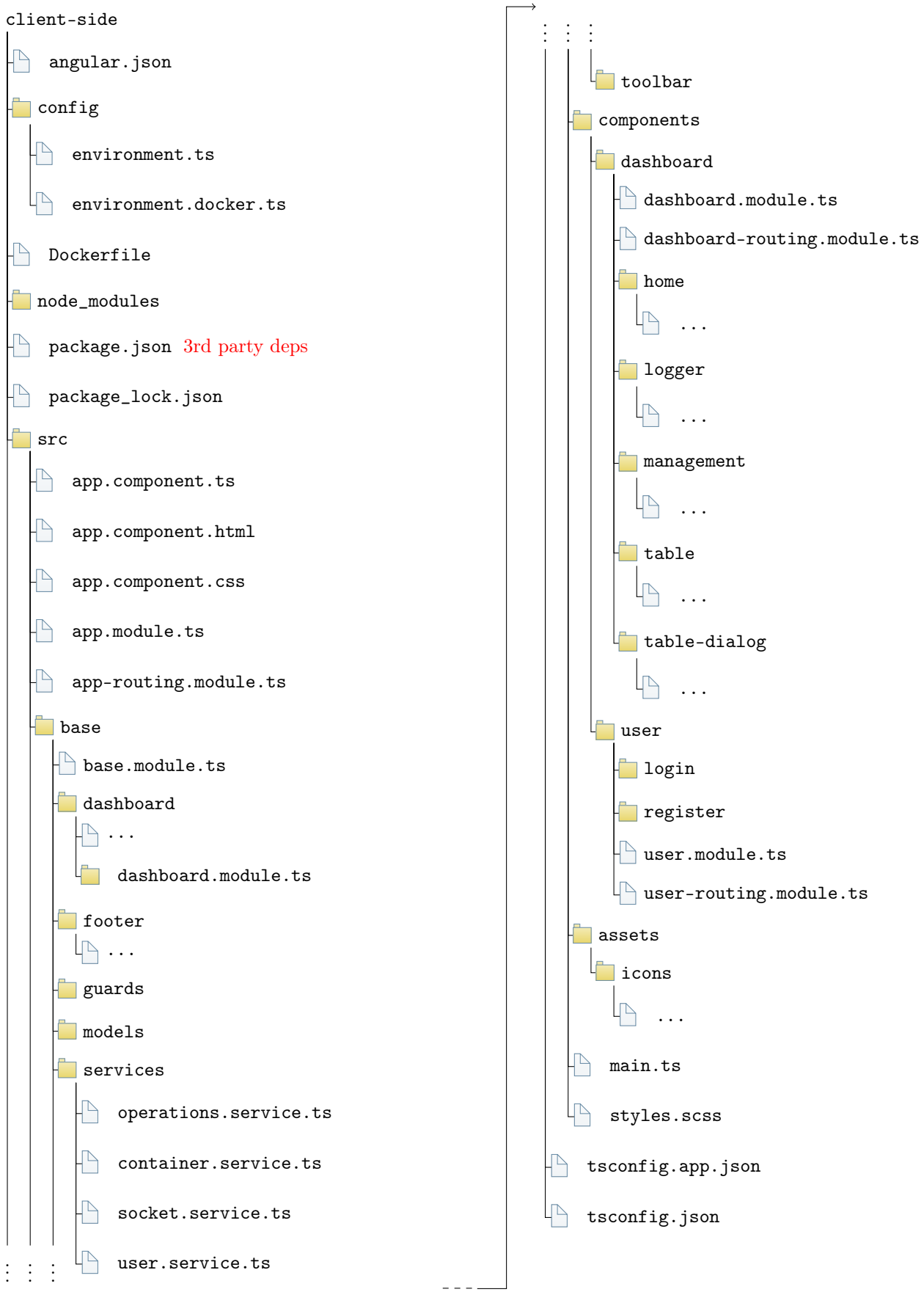
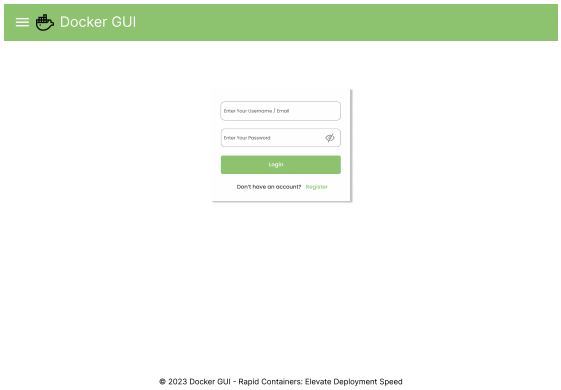
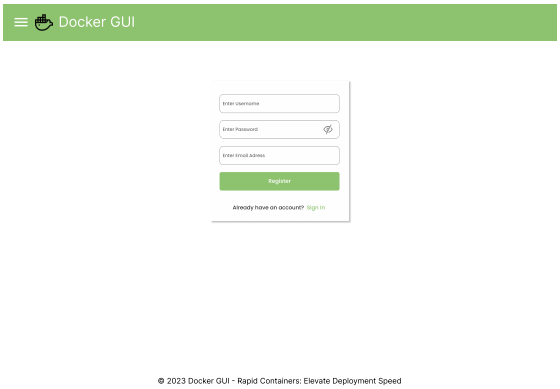


Figure 9: Client side code organization

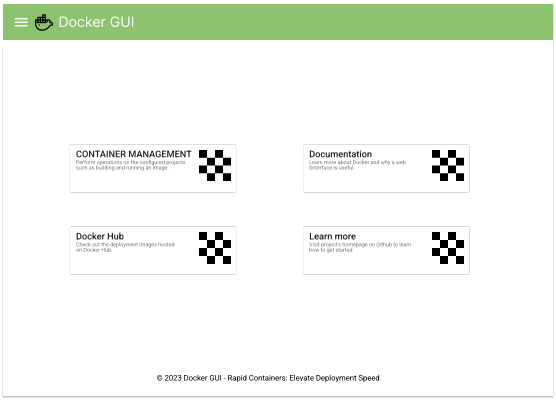


(a) http://localhost:4200/login

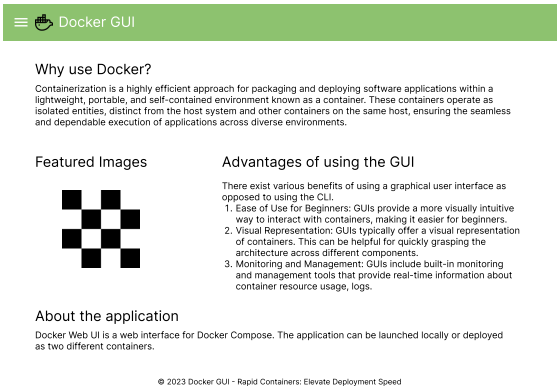


(b) http://localhost:4200/register

Figure 10: Login and Registration pages



(a) http://localhost:4200/dashboard



(b) http://localhost:4200/dashboard/docs

Figure 11: Home and Documentation pages

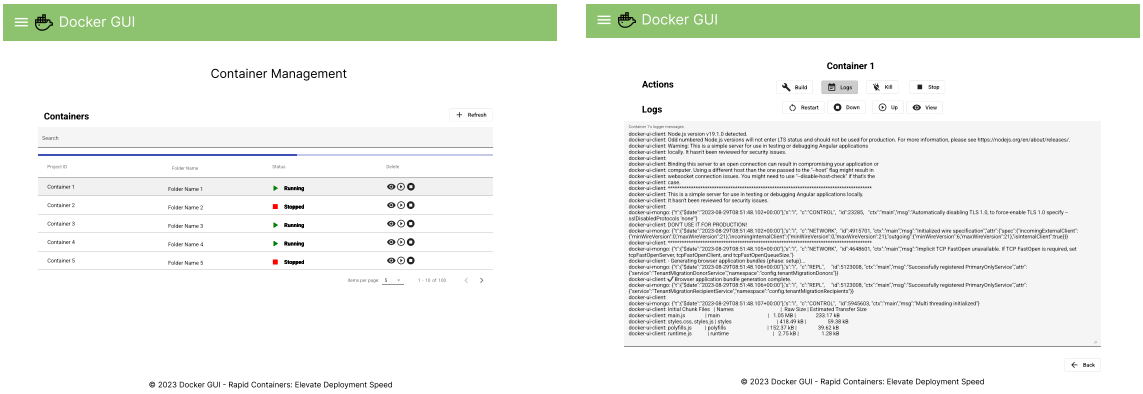
- **Frontend**: a variable number of clients connect using the browser to the Angular Frontend application, which exists in its own self contained Docker container.
  - interaction with the UI leads to requests being made to the ExpressJS back-end.
  - There exists an intermediary layer which manages data in real-time using SocketIO. This is used for updating the user interface dynamically as messages are generated by the containers.
  - the frontend has access to local storage using the Web Cache which allows the user to remain logged in across browser refreshes.
- **Back-end**: using its own container. It speaks with the Mongo database via Mongoose. It also generates messages which are sent to the client via SocketIO.
- **MongoDB**: deployed separately as its own container. It is primarily used for storing registration and user information which is used when an user logs in.

7.2 Docker deployment

The two Docker images (*docker-ui-client* and *docker-ui-server*), are created and uploaded to the Docker Hub [5].

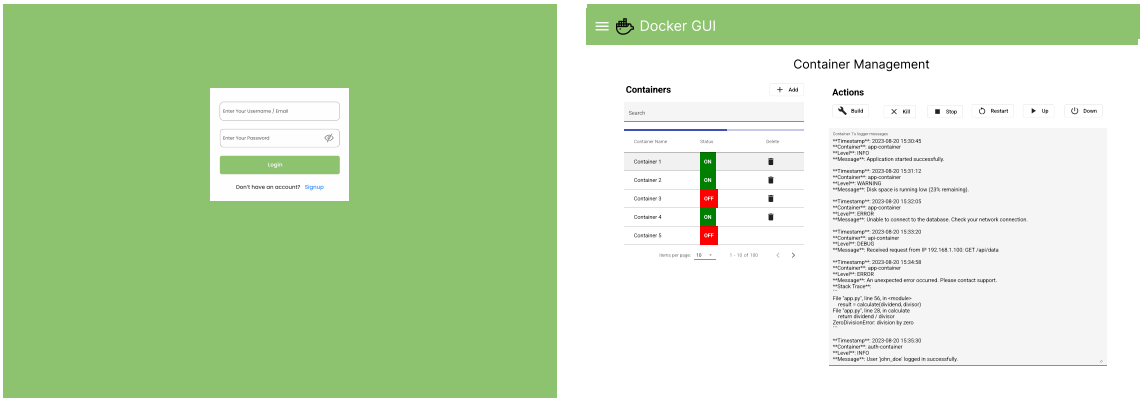
Below is a review of how these images are generated:

[5] Uploaded to https://hub.docker.com/r/razvanfv/docker-ui



(a) http://localhost:4200/dashboard/containers (b) http://localhost:4200/dashboard/containers/<id>

Figure 12: Container and Container Management pages



(a) Login Page (b) Container Management Page

Figure 13: Non final versions, revised via Figma

Listing 6: Deployment to Docker Hub

```yaml
version: "3.8"

# Define the services
services:
  server:
    image: razvanfv/docker-ui:server # based on upstream image
    container_name: docker-ui-server
    hostname: docker_compose
    working_dir: /opt/docker-projects/ # where sample projects are
        located
    ports:
      - "3000:3000" # port listening to
    volumes:
      - ./server-side/docker-projects:/opt/docker-projects
      - /var/run/docker.sock:/var/run/docker.sock
    environment:
      - SECRET=Thisismysecret
      - NODE_ENV=development
      - MONGO_DB_USERNAME=admin-user
      - MONGO_DB_PASSWORD=admin-password
      - MONGO_DB_HOST=database
      - MONGO_DB_PORT=
      - MONGO_DB_PARAMETERS=?authSource=admin
```

	1	2	3	4	5	6	7		ø	
annoying	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	enjoyable	7.00	1
not understandable	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	understandable	6.00	1
creative	0.00% 0	0.00% 0	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	dull	3.00	1
easy to learn	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	difficult to learn	1.00	1
valuable	0.00% 0	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	inferior	2.00	1
boring	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	exciting	6.00	1
not interesting	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	interesting	6.00	1
unpredictable	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	predictable	7.00	1
fast	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	slow	1.00	1
inventive	0.00% 0	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	conventional	2.00	1
obstructive	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	supportive	6.00	1
good	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	bad	1.00	1
complicated	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	easy	7.00	1
unlikable	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	pleasing	6.00	1
usual	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	0.00% 0	leading edge	5.00	1
unpleasant	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	pleasant	6.00	1
secure	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	not secure	1.00	1
motivating	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	demotivating	1.00	1
meets expectations	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	does not meet expectations	1.00	1
inefficient	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	efficient	6.00	1
clear	0.00% 0	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	confusing	2.00	1
impractical	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	100.00% 1	0.00% 0	practical	6.00	1
organized	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	cluttered	1.00	1
attractive	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	unattractive	1.00	1
friendly	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	unfriendly	1.00	1
conservative	0.00% 0	100.00% 1	0.00% 0	0.00% 0	0.00% 0	0.00% 0	0.00% 0	innovative	2.00	1

Figure 14: UEQ: Visual Representation results

```
23        - MONGO_DB_DATABASE=docker-ui
24        - USING_DOCKER=true
25      links:
26        - database
27
28    database: # name of the third service
29      image: mongo # specify image to build container from
30      container_name: docker-ui-mongo
31      environment:
32        - MONGO_INITDB_ROOT_USERNAME=admin-user
33        - MONGO_INITDB_ROOT_PASSWORD=admin-password
34        - MONGO_DB_USERNAME=admin-user1
35        - MONGO_DB_PASSWORD=admin-password1
36        - MONGO_DB=docker-ui
37      volumes:
38        - ./mongo:/home/mongodb
39        - ./mongo/init-db.d/:/docker-ui-entrypoint-init-db.d/
40        - ./mongo/db:/data/db
41      ports:
42        - "27017:27017" # specify port forewarding
43
44
45    client: # name of the first service
```

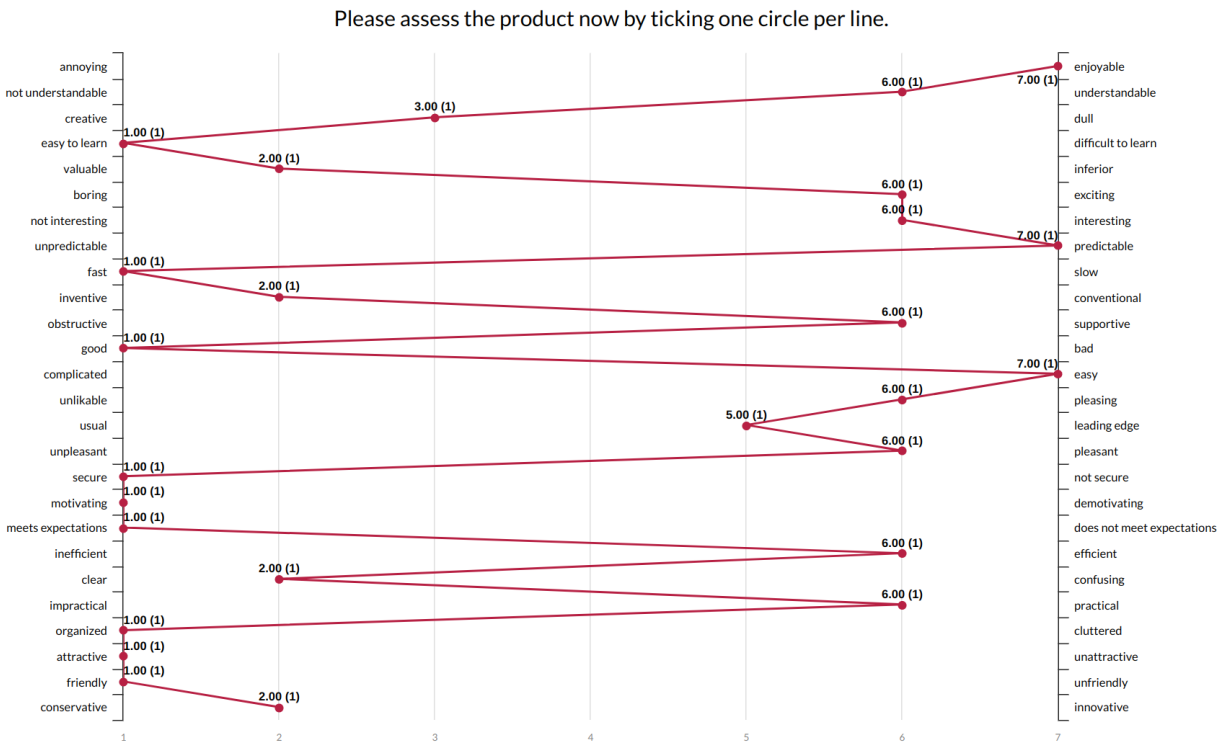


Figure 15: UEQ: Average of the results

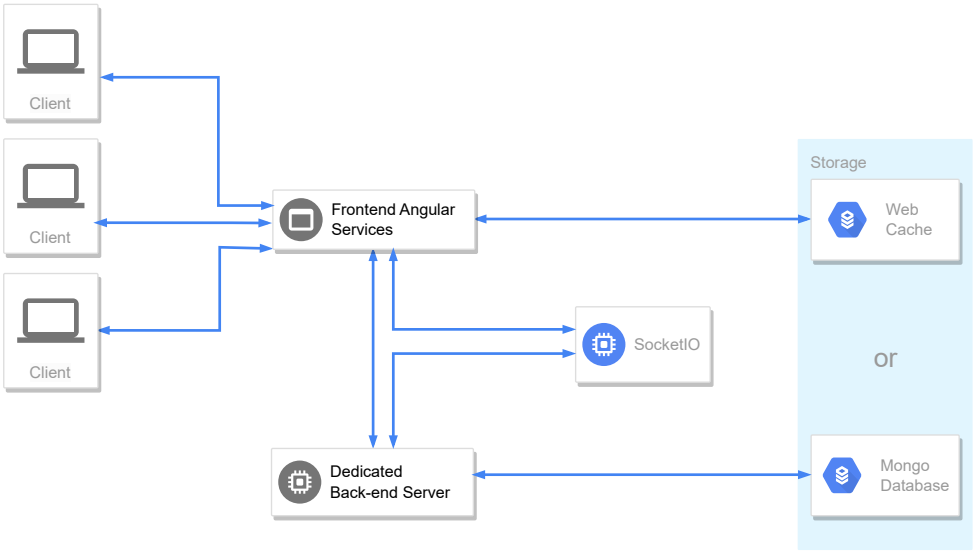


Figure 16: Docker Architecture

```
46      image: razvanfv/docker-ui:client
47      container_name: docker-ui-client
48      volumes:
49        - /var/run/docker.sock:/var/run/docker.sock
50      ports:
51        - "4200:4200" # <host>:<container>
52      links:
53        - server
```

Listing 6 is a *docker-compose.yml* file which creates and deploys three services: the **server**, the **database** and the **client**. Below is a description of the file:

- **image**: the context used as an image. Can be remote, on Docker Hub or local to a *Dockerfile*.
- **working_dir**: sets the working directory of the container that is created.
- **environment**: variables needed to configure the service. Could be for instance configuration parameters needed by MongoDB to create the database.
- **volumes**: are a mechanism for persisting data generated by and used by Docker containers. It should be noted that the following line allows Docker inside the container to communicate with the host.

  $$/var/run/docker.sock : /var/run/docker.sock$$
- **ports**: links ports between the host and the container in such a way to allow inter-communication.
- **links**: There exists a dependency between the server and the database, as well as one between the client and the server which is defined via the **links** keyword. This make the corresponding services reachable across the network.

8 Conclusion

To conclude, it follows a list of learning outcomes that were attained as a result of having completed this project.

Containerization. The concepts related to **containerization** were particularly interesting. I learnt about how to package an application into smaller images each deployed independently through the use of containers. As an example, I was able to obtain familiarity with the mechanisms used by Docker to deploy a somewhat non trivial application around a variable number of services.

Docker. As it was already mentioned, I hosted and deployed the application to Docker Hub. As a consequence, I learnt more thorough knowledge of images as the prime building block, created directly through the use of *Dockerfile*s. These images were instantiated to create containers. These containers communicate with one another and run the application. Having implemented functionality leveraging these concepts through the use of docker-compose files was particularly exciting.

Docker Compose. The project examples which were provided enabled the user to perform common operations on the corresponding containers. These operations leveraged the use of a Javascript library known as *docker-compose*. The implemented functionality involved tasks such as building, running or removing a container.

Future work. Keeping in mind the importance and inter-dependence of services with the docker-compose files, perhaps further work might involve highlighting better the status of each service. This would support the user to understand more thoroughly and easily the status of each executed docker-compose file. Currently, a project is considered *up* when all the services which compose it have been started successfully. In some cases this is inconvenient as it requires the user to carefully check the project's logging information in order to fully individuate the issue.

Angular. On the other side, having worked with Angular allowed me to learn relatively more advanced concepts. For example, I made use of guards to protect pages and redirect to the login page in case the user would not be signed in. Furthermore, the JsonWebToken was used to keep the user authenticated across page navigation. To be more precise, during refreshes of the website, user authentication information would normally get lost but by using these tokens the information is securely stored and restored whenever needed.

Conclusion. As a whole, the project was particularly interesting and having developed more thorough knowledge of both Docker and Angular, I am excited to see when these skills will be of use in the future.

References

[1] Wikipedia. *Bcrypt — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1175136618. [Online; accessed 18-September-2023]. 2023.

[2] Wikipedia. *Figma (software) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Figma%20(software)&oldid=1173484278. [Online; accessed 19-September-2023]. 2023.

[3] Wikipedia. *Rainbow table — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Rainbow%20table&oldid=1171785273. [Online; accessed 21-September-2023]. 2023.

[4] Wikipedia. *Socket.IO — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Socket.IO&oldid=1173333199. [Online; accessed 19-September-2023]. 2023.