Query My Library! Question-Answering with Open-Source LLMs and Local Books

Razvan Florian Vasile
University of Bologna
Department of Computer Science
razvanflorian.vasile@studio.unibo.it

Abstract

Docker is a tool that allows to create and run virtual machines. It is used by many companies to deploy their applications in production environments. In this paper, I present a web interface for the management of Docker containers. The proposed application makes use of Docker-Compose files to orchestrate many Docker containers, the main idea behind the approach being to access the Docker daemon for managing different container environments from within the browser. By doing this, I expect that the steep learning curve, traditionally involved in managing multiple Docker environments, will be reduced. The application is meant for beginners in order to get started more easily with Docker, while at the same time facilitate learning. There are many advantages to learning Docker, but some of the main benefits for beginners include: 1) simplified environment setup which makes it easier to quickly test and remove applications without polluting the host environment, 2) simplify application isolation, which makes it possible to run multiple versions of specific applications without any dependency conflicts and 3) Docker being an industry-standard tool, learning Docker can give valuable skills for future job prospects. The application is available at https://github.com/raz-m12/docker-ui/.

1. Introduction

Problem definition. Suppose a skilled software developer regularly works with complex systems and diverse coding environments. The engineer might have a virtual toolkit filled with various programming languages, frameworks, and libraries. While deeply passionate about her work, she may sometimes find it challenging to maintain consistency across different environments or to ensure that her applications run smoothly regardless of where they are deployed. To streamline this process, she turns to Docker, a powerful containerization platform that allows her to encapsulate her applications along with all their dependencies.

Needless to say, all the flexibility that Docker brings, comes with the disadvantage that it has a steep learning curve. As a result, a graphical user interface meant to integrate the docker instructions while also streamlining and simplifying the Docker API would be very useful.

Key ideas. In this work, I explore the task of integrating the Docker API into a web interface in order to ease the transition for beginners to use Docker. This application may be useful to novice users but experienced ones as well, that may want to avoid the hustle of interacting with Docker through the command line and prefer using a web interface for managing multiple application environments. As a result, I propose an application that is meant to help orchestrate docker images through an intuitive web interface, effectively simplifying the complex workflows required to interact with multiple containers.

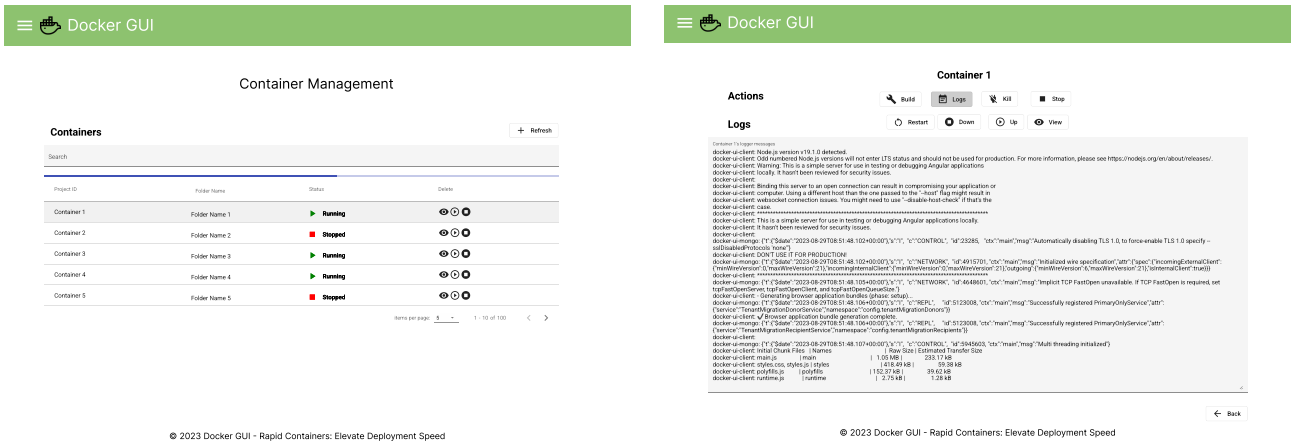
The appeal of a Web UI. Our hard-working software developer can spot some clear advantages to this approach; She may have more time to focus on solving concrete tasks as opposed to spending this time on setting up the application environment. By spending less time on configuring containers, she will eventually become accustomed to how containers work by passively learning to interact with docker images and manage different use cases. As a result, she will learn more easily complex container workflows by using the command line only when absolutely required to do so.

2. Requirements Engineering

2.1. User Stories

User stories were initially sketched in order to facilitate development and develop a holistic view of the most basic use cases involved in the project. These user stories are shown in Table 1.

Each task was assigned a priority and different personas were created for each individual potentially using the application.



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

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

Figure 1. The User Interface design for interacting with various containers. The image on the left displays a unified view of the status of all running containers, while on the right it is possible to perform more fine-grained instructions on a specific container.

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. The User Stories used for bootstrapping development. Potential categories, so called personas, were identified in order to accommodate the needs of as many users as possible.

2.2. Personas and A Story Board

It is possible to identify different personas in Table 1. These user groups were identified by acknowledging how the user would use the application and also what he or she would prefer to see. The approach is a tool that attempts to model a good balance between the technical knowledge required to perform different tasks, while at the same time not overwhelming the user with too much information.

This approach was useful in recognizing that developing a minimalistic interface for the application is going to play an important role in its design. Moreover, by keeping the design simple we comply with some of the heuristics described by Nielsen in [3]. I made use of Figma prototypes to create initial sketches, some of which are going to be shown in Section 3.1.

2.3. Nielsen's Usability Heuristics

Special attention was payed to the following heuristics:

- **Visibility of system status.** Figure 1a displays the global status of all running and non-running containers, with the ability to start or stop specific docker images on demand.

  The panel in Figure 1b displays additional visibility information about a container's state, effectively making available to the user any logging information that is being generated.

- **Error prevention.** Overlaying the mouse on any button provides specific information about its function, effectively reducing the chance of error.

- **Design and minimalistic UI.** Attention was paid to provide the user with enough information to effectively perform designated tasks, while at the same time keep the UI as minimalistic as possible, as shown for Figure 2a.

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	✓ 2.2. Setup README.md on Github with information about the project. ✓ 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.
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.

Table 2. The User Stories which were developed during the initial phases of the project.

[a]Figma Design Page: here

- **Documentation.** Attention was paid to build an interface as simple as possible without the need to go through documentation, however some documented instructions are nevertheless provided as shown in Figure 2b.

- **Accessability.** The Material Design theme was used to assure accessability. A related discussion can be found at Section 4.2.4.

2.4. Design tasks

After the initial phase on understanding various scenarios which the user might find himself/herself in, an analysis was made to develop subsequent tasks for each individual user story. This approach was helpful in developing an additional perspective for designing the system. It offered valuable insights without overwhelming details. The outcomes are presented in Table 2.

User Stories and Design tasks. The user stories provide high-level requirements and goals, while the design tasks are meant to break down the user stories into tangible design elements that fullfil users' needs, providing the bricks of what is meant to be developed.

Purpose. The design tasks are used to reduce the gap between specific activities, as identified by the user stories, to creating wire-frames and mockups which are concrete tools for developing the user interface more easily. These mockups are going to be evaluated by a group of users, details which are described in Section 3.2. By using this methodology based on the user-centric evaluation, we expect to identify problems related to the user experience early in the development process, which in the end helps develop software which is aligned as closely as possible to user's expectations.

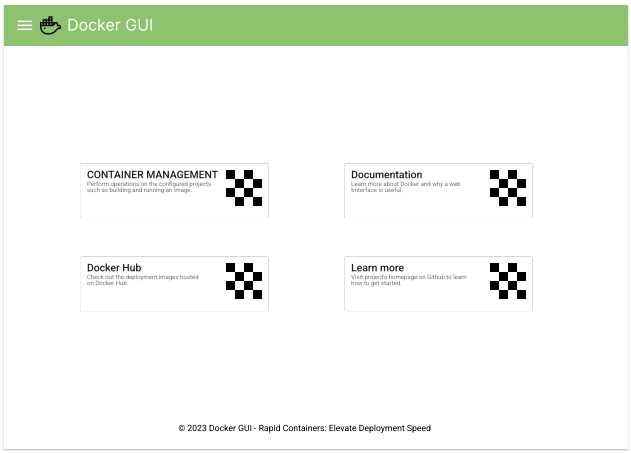
The software used to build the mockups is based on the Figma platform [1]. The design tasks define the user interface (UI) and provide useful information in understanding concerns regarding the user experience. For example, the design tasks are translated into tangible design elements in the next section, and these sketches are evaluated by a group of users.

3. Design

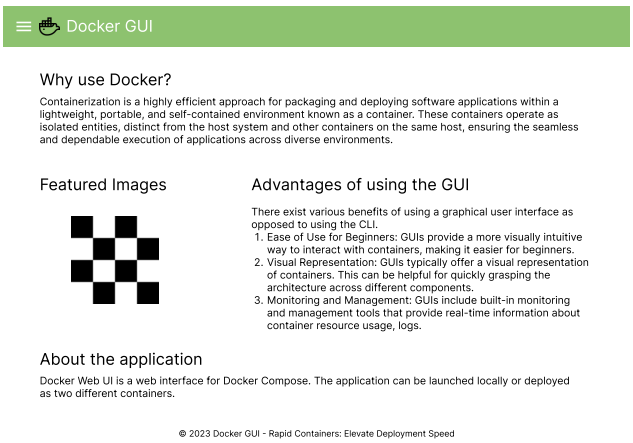
3.1. Figma Mockups

What is Figma? It is a collaborative web application for interface design. Its features focus on user interface and user experience design, utilizing a variety of vector graphics editor and prototyping tools [8]. These tools are useful for creating early design mockups which are relatively low-cost.

Some of its key features that were used in the project were *Vector Editing*, *Prototyping*, *Component Libraries* and



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



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

Figure 2. A map of how the website is organized is shown in the left image. The right image shows documenting information including features of the website.

Plug-ins.

Examples. Some Figma diagrams were already displayed in the previous sections, which include Figure 1 and 2. For the sake of completeness, the mockup for the login page is shown in Figure 3. All of them can be found online at [5].

3.2. User Centric Design

Assessing User's Experience. User Experience Questionnaires (UEQ) were used to assess and measure users' feedback upon interacting with the application [6]. The results are displayed in Figure 4 and Figure 5.

On the right-hand-side of Figure 4, it is shown the number of people that took part in the survey and within each line there is the choice each person made against each particular evaluated metric.

The second diagram, shown in Figure 5, 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.

3.2.1 Benefits of using the UEQs

Outdated components. As a result of users' feedback, some components were modified. In particular, the initial design of the container management and login pages are shown in Appendix B.3. They were considered a bit cluttered and the brightness as well as the contrast were particularly high. Also, the size of the widgets was also considered a bit odd with respect to the other parts of the application. As a result some changes were made and the final results are shown in Figures 1b and 3a.

Most notably, the color and size of the login/registration widgets were changed and the logging display area was completely reworked by moving the logging information into a page of its own.

3.3. MVC

Together with the MVC design pattern, I used the MEAN technological stack to develop the application. Below is an overview of how the architecture is 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. I used Angular to create the 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 different components, acting as an interface between the view and model. Routes were defined as endpoints in the **/server-side/routes** folder.

Architectural 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 business logic and the controller handles user input and manages

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

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

Figure 3. Login and Registration pages

	1		2		3		4		5		6		7			ø	
annoying	0.00%	0	0.00%	0	0.00%	0	0.00%	0	20.00%	2	10.00%	1	70.00%	7	enjoyable	6.50	10
not understandable	0.00%	0	0.00%	0	0.00%	0	0.00%	0	10.00%	1	20.00%	2	70.00%	7	understandable	6.60	10
creative	50.00%	5	30.00%	3	20.00%	2	0.00%	0	0.00%	0	0.00%	0	0.00%	0	dull	1.70	10
easy to learn	80.00%	8	10.00%	1	10.00%	1	0.00%	0	0.00%	0	0.00%	0	0.00%	0	difficult to learn	1.30	10
valuable	90.00%	9	10.00%	1	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	inferior	1.10	10
boring	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	40.00%	4	60.00%	6	exciting	6.60	10
not interesting	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	40.00%	4	60.00%	6	interesting	6.60	10
unpredictable	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	20.00%	2	80.00%	8	predictable	6.80	10
fast	90.00%	9	10.00%	1	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	slow	1.10	10
inventive	70.00%	7	0.00%	0	0.00%	0	0.00%	0	0.00%	0	20.00%	2	10.00%	1	conventional	2.60	10
obstructive	0.00%	0	0.00%	0	0.00%	0	0.00%	0	10.00%	1	40.00%	4	50.00%	5	supportive	6.40	10
good	90.00%	9	10.00%	1	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	bad	1.10	10
complicated	0.00%	0	0.00%	0	10.00%	1	0.00%	0	20.00%	2	40.00%	4	30.00%	3	easy	5.80	10
unlikable	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	10.00%	1	90.00%	9	pleasing	6.90	10
usual	20.00%	2	10.00%	1	0.00%	0	0.00%	0	0.00%	0	30.00%	3	40.00%	4	leading edge	5.00	10
unpleasant	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	50.00%	5	50.00%	5	pleasant	6.50	10
secure	80.00%	8	20.00%	2	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	not secure	1.20	10
motivating	50.00%	5	30.00%	3	20.00%	2	0.00%	0	0.00%	0	0.00%	0	0.00%	0	demotivating	1.70	10
meets expectations	60.00%	6	40.00%	4	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	does not meet expectations	1.40	10
inefficient	0.00%	0	0.00%	0	0.00%	0	0.00%	0	10.00%	1	40.00%	4	50.00%	5	efficient	6.40	10
clear	100.00%	10	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	confusing	1.00	10
impractical	0.00%	0	0.00%	0	0.00%	0	0.00%	0	20.00%	2	50.00%	5	30.00%	3	practical	6.10	10
organized	100.00%	10	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	cluttered	1.00	10
attractive	60.00%	6	40.00%	4	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	unattractive	1.40	10
friendly	70.00%	7	20.00%	2	10.00%	1	0.00%	0	0.00%	0	0.00%	0	0.00%	0	unfriendly	1.40	10
conservative	0.00%	0	10.00%	1	0.00%	0	0.00%	0	0.00%	0	30.00%	3	60.00%	6	innovative	6.20	10

Figure 4. UEQ: Visual Representation results

the flow of data between the model and view. A visual representation of how MVC works is shown in Appendix A. Alongside the MVC pattern, the code organization is displayed in Figure 10.

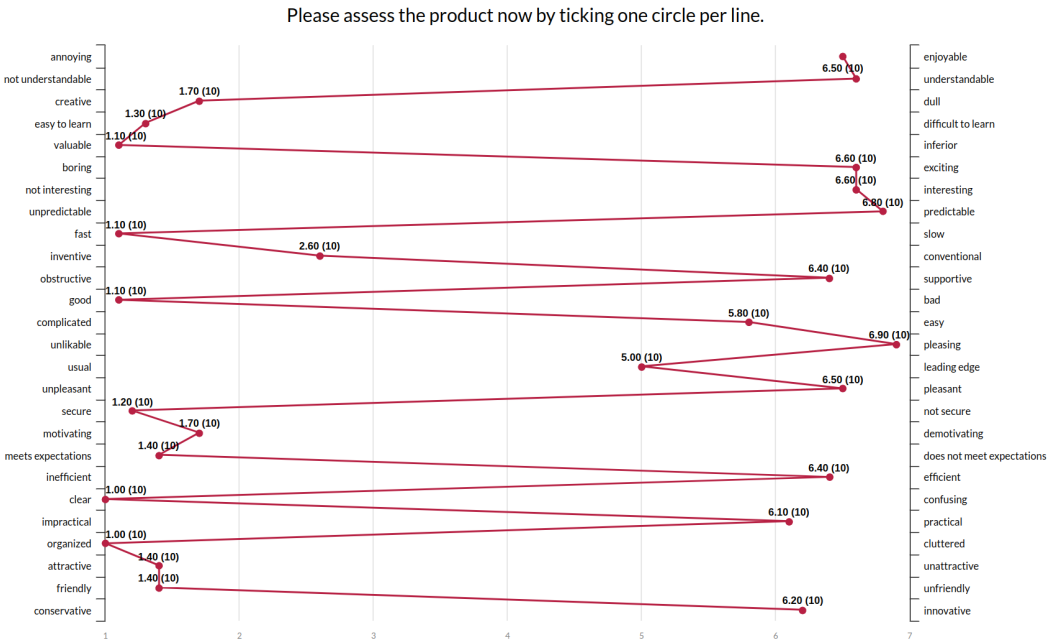


Figure 5. UEQ: Average of the results

3.4. Responsive design

I used both Bootstrap as well as the Flexbox framework to design the web pages. This approach allowed to make use of the flexibility offered by Flex containers while also having available the simplicity provided by Bootstrap containers required in simple pages. Some other benefits are described below:

Rapid Development. Bootstrap provides a set of predefined components and grid system that significantly simplify and speed up the development process. This is useful for quickly setting up pages that do not require the advanced flexibility offered by Flexbox. On the other hand, Flexbox offers more flexibility and control over the layout, allowing to create more complex and responsive designs. Leveraging their strengths and weaknesses, it was possible to employ each framework on page-by-page basis. Generally, this improved development speed, while at the same time allowed to develop fluid mobile-first web pages leveraging what each individual framework excels at.

Mobile-first Response Design. Below are some specific areas which used Bootstrap with its powerful grid system and general ease of use. Moreover, some other web pages required more control and flexibility over the organization of each individual element, features provided by the powerful Flexbox framework.

Firstly, the general layout from the documentation page shown in Figure 2b was created using the Bootstrap layout grid. The design was quite simple that did not require much flexibility.

Secondly, more complex pages that required finer control over individual elements are displayed in the dashboard, as illustrated in the Figures 1a and 1b. Using Bootstrap to build these pages was not ideal because the button positioning needed more precise control.

4. Architectural Design and Technologies

Alongside the MEAN technological stack, there were used a few other libraries which helped create a responsive design.

4.1. The MEAN Stack

Architecture. MEAN is a popular technology stack for building web applications. The architectural design is shown in Figure 6. MEAN is an acronym that stands for MongoDB, Express.js, 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,

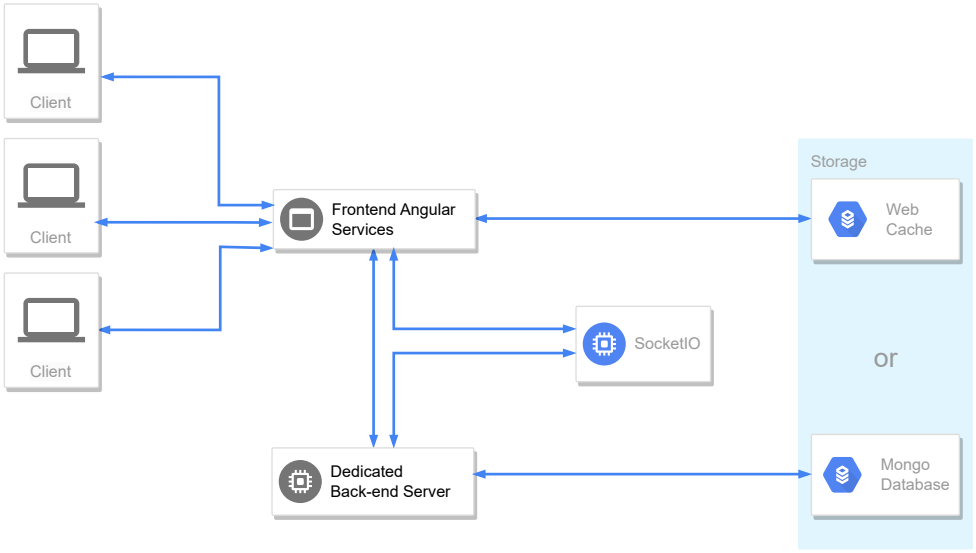


Figure 6. Docker Architecture

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 Typescript 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 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

The used libraries serve the following purposes:

- **Real-time communication.** *SocketIO* was used to ensure timely communicate between the client and server.

- **Security.** Passwords were securely encrypted and stored in the database using *bcrypt*. Moreover, in order to ensure secure exchanges and verified communication between client and server *JsonWebToken* was used.

- **Feedback, Accessability and UI.** The *Toastr* package was used to provide notifications about the completed operations and *Angular Material* was used to provide a standardized theme across the entire application.

4.2.1 SocketIO

Bidirectional Communication. Socket.io is a JavaScript library that simplifies real-time, bidirectional communication between browsers and servers. It was used for displaying data related to a container's log information. This means that the user is updated in real-time with information about a container's status which help resolve some of the potential problems that may arise and guarantees that the user is kept informed about the status of the container.

4.2.2 BCrypt

Secure Password Storage. BCrypt is a password-hashing function based on the Blowfish cipher. Besides incorporating a salt to protect against rainbow table attacks [9], 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 [7]. An example on how bcrypt is used can be found in Appendix B.1.

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

- **User's password.** To load a new password into the system, the user selects a password. This password is combined with a fixed-length salt value.

- **Salt purpose.** Salt value is a pseudorandom 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.

- **Retrieval.** 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 algorithm. If the result matches the stored hash value, the password is accepted.

Salt purposes. Some of the main advantages to using BCrypt can be described as follows. Firstly, duplicate passwords are not visible in the database. Moreover, due to the fact that BCrypt is a slow algorithm, it greatly increases the difficult of offline dictionary attacks. Lastly, 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.

4.2.3 JsonWebToken

Authentication Session. To ensure that information sent between two parties can be easily verified and trusted, a JsonWebToken is used. These tokens authenticate a user's request during the login phase and maintains the user authentication status across browser refreshes. Sample code is shown in Appendix B.2.

4.2.4 Angular Material

Reusable Components. 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.

The library offers a wide range of UI components such as buttons, forms, tables, toolbars, navigation elements and dialogs.

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

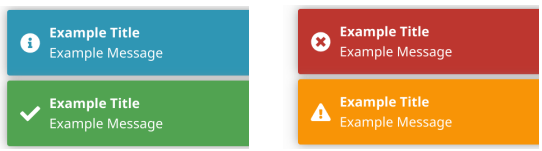
Conformity to Google's design standards. By using the predefined Angular Material theme, we assure that the interface design conforms to the Google UI design standards. The theming system is based on Google's Material Design 3 specification which is the latest iteration of Google's open-source design system, Material Design [2].

Accessibility. The components are designed to be accessible by default, ensuring that the application is usable by a wide range of people. Some of the accessability concerns which were kept in consideration are the following:

- **Descriptive Alt Text.** For images, the **alt** attribute displays the content of an image, in case its content fails to be correctly displayed.
- **Accessible HTML elements.** By using Angular the website 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 are navigable using only a keyboard. Users can interact with all interactive elements using keyboard keys like Tab, Enter, and Space.
- **Error Messaging.** Short error messages are displayed in case of errors and some input fields become red when they are mandatory, giving visual feedback for content which must be supplied. Examples can be found on the Registration page.

4.2.5 Notification Service: Toastr

Visual Notifications. Ngx-Toastr is a JavaScript library used for displaying non-blocking notifications to users in the form of pop-up messages. These messages are used to provide feedback, alerts and success notifications. Some examples are shown in Figure 7.



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

Figure 7. Toastr diagrams

5. Code

5.1. Registration and Login

Login. Below are described the sequence of steps required to complete the authentication process, which are also displayed using a sequence diagram in Figure 8.

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.

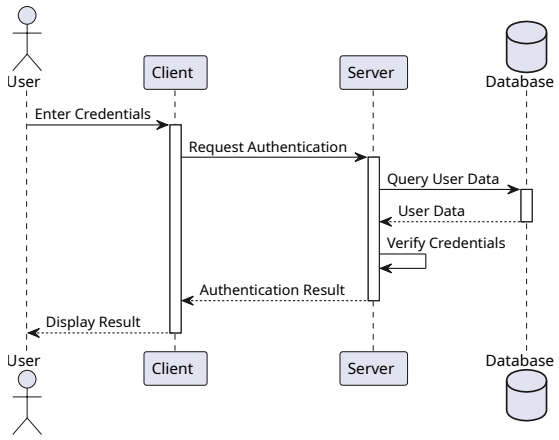


Figure 8. Login Sequence Diagram

Registration. On the other hand, the registration logic is very similar to the login process and for brevity I will display the diagram in Appendix C.1.

The encryption and token authentication processes are described in Figures B.1 and B.2.

5.1.1 The Login and Registration Modules

Dashboard and User modules. Figures 13 and 14 show two class diagrams which represent the relationship between the two main modules: the User login and registration components and the Dashboard component meant to manage different containers.

A discussion about each diagram is available in the Appendix, together with a brief description of each component's main reusable services.

5.2. Code Organization

Back-end. The back-end code was organized around the MVC pattern. Its organization is fairly simple and is shown in Appendix D.1. It is important to note that the views are located in the Angular directory.

Front-end. Similarly, the front-end code organization based on the MVVM model is shown in D.2. The components were organized in terms of services and modules, which are key architectural tools provided by Angular. The models are located in the shared directory, while the controllers and view-models are grouped up based on their corresponding feature. The views are represented by the ".component.html" and ".component.css" files, while the model-views correspond to the component typescript files.

6. Deployment

Diagram 6 shows the high level architectural view of the application and relevant deployment code is available in Appendix E. This deployment process uses a number of services which are described below:

Frontend: a variable number of clients connect using the browser to the Angular frontend application, which is represented using its own Docker container. The process is described below:

- Interaction with the UI leads to requests being made to the Express backend.

- 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.

Backend operates as a service within its own container. It interacts with the Mongo database using Mongoose and sends messages to the client through SocketIO.

MongoDB is deployed as a standalone service in its own container. It primarily stores registration and user information, which is accessed during login requests.

The Docker images were uploaded to the Docker Hub [4].

7. Conclusion

Containerization. The most interesting part of the project was related to containerization. I learnt about how to package an application into smaller images, each deployed independently through the use of containers. As a result I am now more familiar with the mechanisms used by Docker to deploy a non trivial applications organized around multiple services.

Docker. The project was hosted and deployed to the Docker Hub. Gaining experience with the deployment process is exciting because this significantly simplifies application distribution, which is a valuable skill in the tech industry.

Docker Compose. The docker-compose API streamlines and simplifies container management process. This tool proved to be useful for building, running and removing containers, which greatly reduces the steep learning curve required for getting started with Docker.

Angular. Finally, regarding the Angular framework, I was glad to be able to write modular code in Typescript. This design allows to reuse services across multiple parts of the application, effectively reutilizing code. Also, the dependency injection mechanism was something I was already accustomed to. It allows to cleanly instantiate services as well as pipes without any boilerplate code. As an example of more complex workflows, I made use of guards to protect pages and redirect in case the user was not signed in. Also, the JsonWebToken was used to keep the user authenticated across page navigation.

Future work. Potential future work may involve providing support for Docker files. Currently, the project works by identifying docker-compose files, giving the ability to run each file individually. This is convenient because docker-compose files provide better flexibility compared to simple Dockerfiles. However, more flexibility by providing the ability to manage Dockerfiles as well, would be a useful extension to the project.

References

[1] Figma. Figma: The Collaborative Interface Design Tool — figma.com. https://figma.com/. [Accessed 19-08-2024].

[2] Google. Material Design — m3.material.io. https://m3.material.io/. [Accessed 19-08-2024].

[3] Nielsen. 10 Usability Heuristics for User Interface Design — nngroup.com. https://www.nngroup.com/articles/ten-usability-heuristics/. [Accessed 17-08-2024].

[4] Razvan Florian Vasile. https://hub.docker.com/r/razvanfv/docker-ui. [Accessed 19-08-2024].

[5] Razvan Florian Vasile. Figma — figma.com. https://www.figma.com/file/TGgkRNt5faxYmILkp60JdN/Docker-UI?type=design&node-id=0-1&mode=design&t=rZRXwLEom6yfP9AU-0. [Accessed 19-08-2024].

[6] Razvan Florian Vasile. User experience questionnaires — easy-feedback.com. https://easy-feedback.com/docker-ui/1721729/sV3Xny-270e61ca6f27407fb95806c4f3e446b0. [Accessed 19-08-2024].

[7] Wikipedia. Bcrypt — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1175136618, 2023. [Accessed 19-08-2024].

[8] Wikipedia. Figma (software) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Figma%20(software)&oldid=1173484278, 2023. [Accessed 19-08-2024].

[9] Wikipedia. Rainbow table — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Rainbow%20table&oldid=1171785273, 2023. [Accessed 19-08-2024].

A. Appendix: MVC Pattern

The user views the user interface using the View, which is represented by the Angular frontend. Then he or she interacts with the controller to change the state of the model, which in turn changes the view and the user is notified.

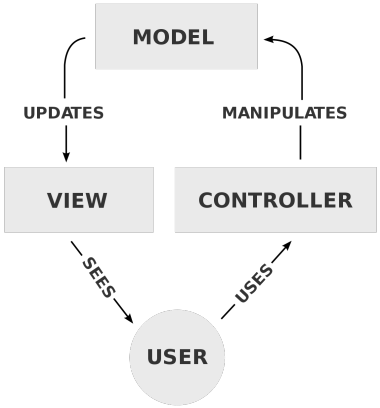


Figure 9. MVC Pattern

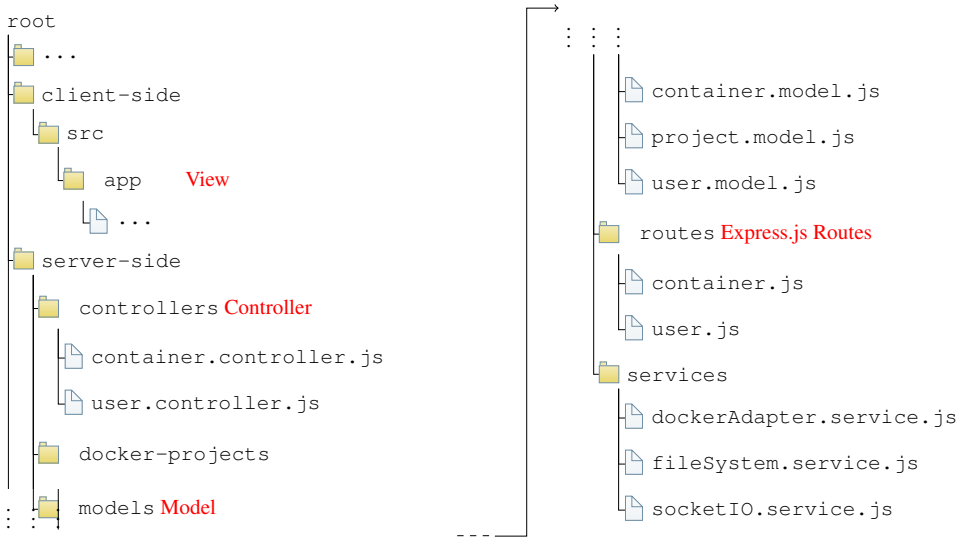


Figure 10. Code organization under MVC

B. Appendix: Design

B.1. BCrypt

The following code shows how to create and store a user in MongoDB.

```
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,
  });
})
```

B.2. JsonWebToken

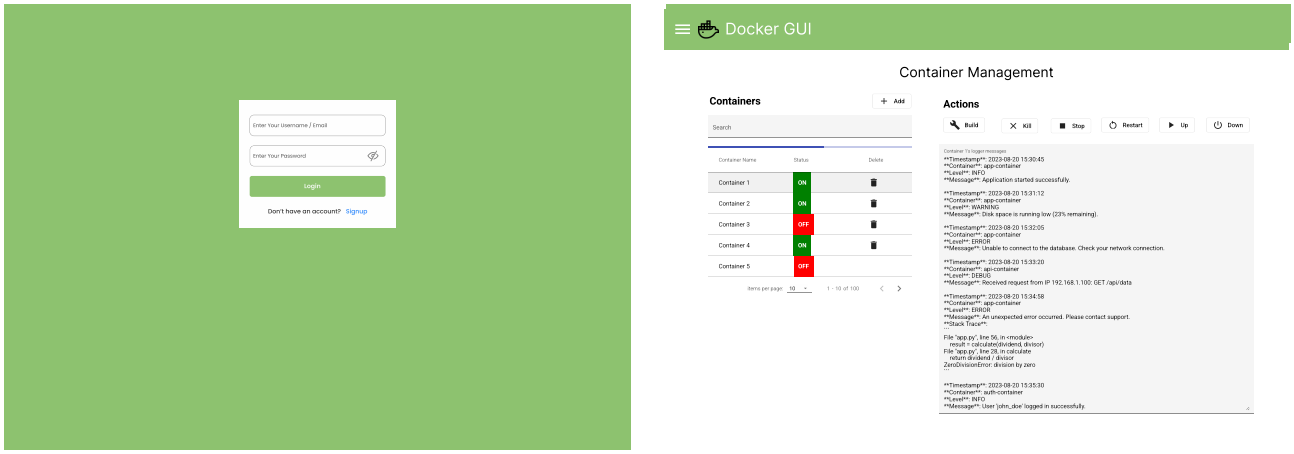
The following code checks if the current user is authenticated, in which case it verifies that the token has has not expired. In case it expired, it redirects to the login page, otherwise allows to see the desired resource.

```
function canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
  // Check whether code runs in web browser
  if (typeof window !== "undefined") {
    // retrieve the token and check whether expired
    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;
}
```

B.3. Outdated Design Samples

These are some initial sketches of the UI. As a result of the UEQs these mockups were reworked and their final version is shown in Figures 3a and 3b.



(a) Login Page

(b) Container Management Page

Figure 11. Non final versions, revised as a consequence of the UEQs.

C. Code

C.1. Register Sequence Diagram

Below is a sequence diagram that represents the register process for a user.

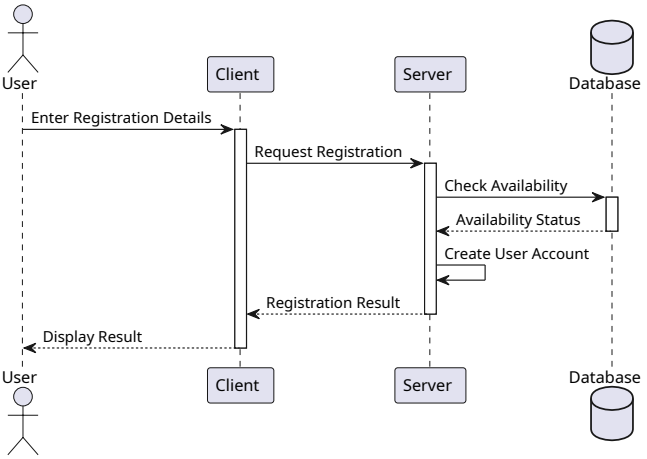


Figure 12. Registration Sequence Diagram

C.2. User Class Diagram

User Module First of all, 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, using of Typescript's strong typing system.

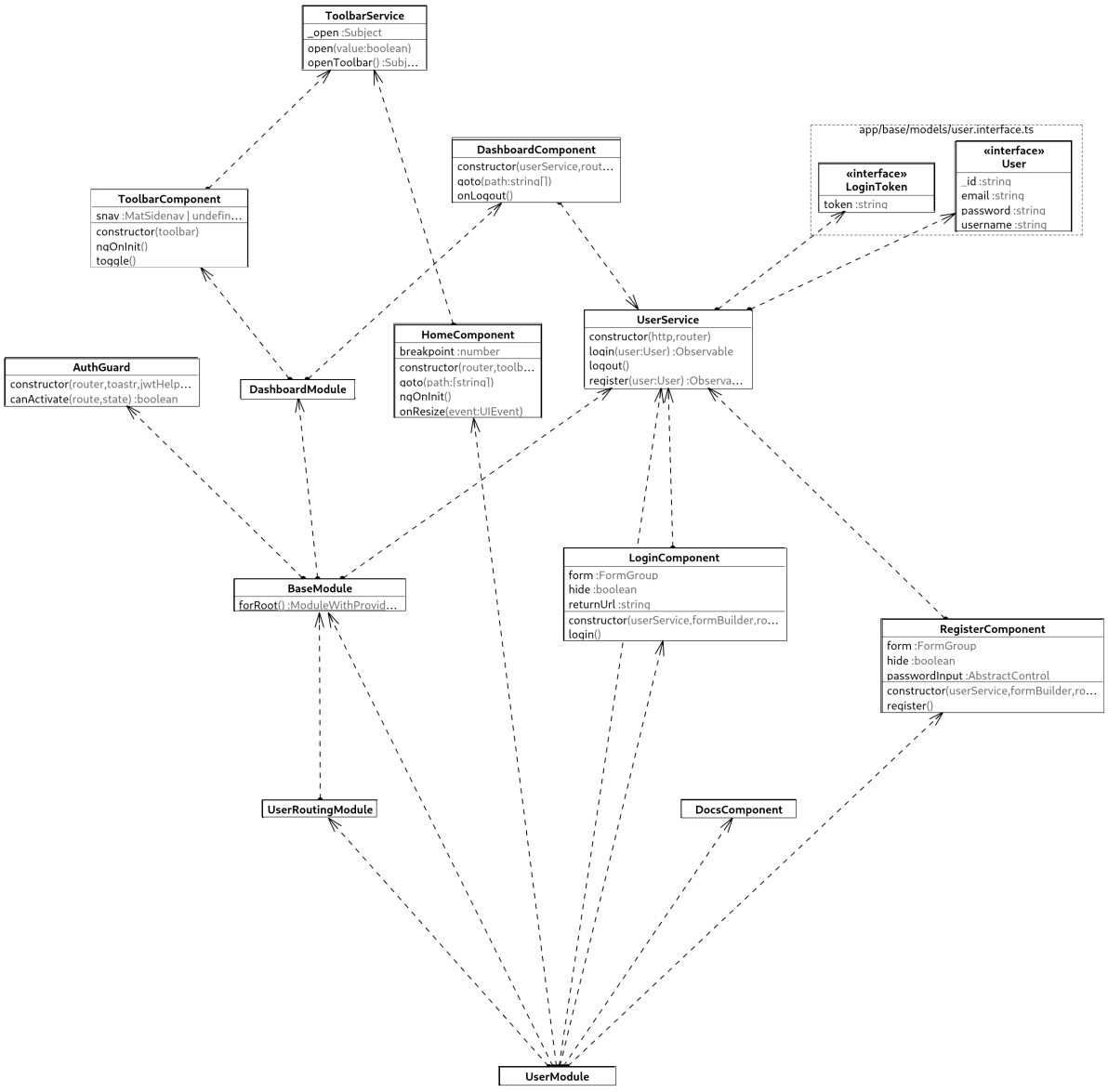


Figure 13. Angular User Registration/Login Module

[1] DTO - Data Transfer Object

Dashboard Module The main components are the following:

1. **ManagementComponent.** It contains the functionality related to managing 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.

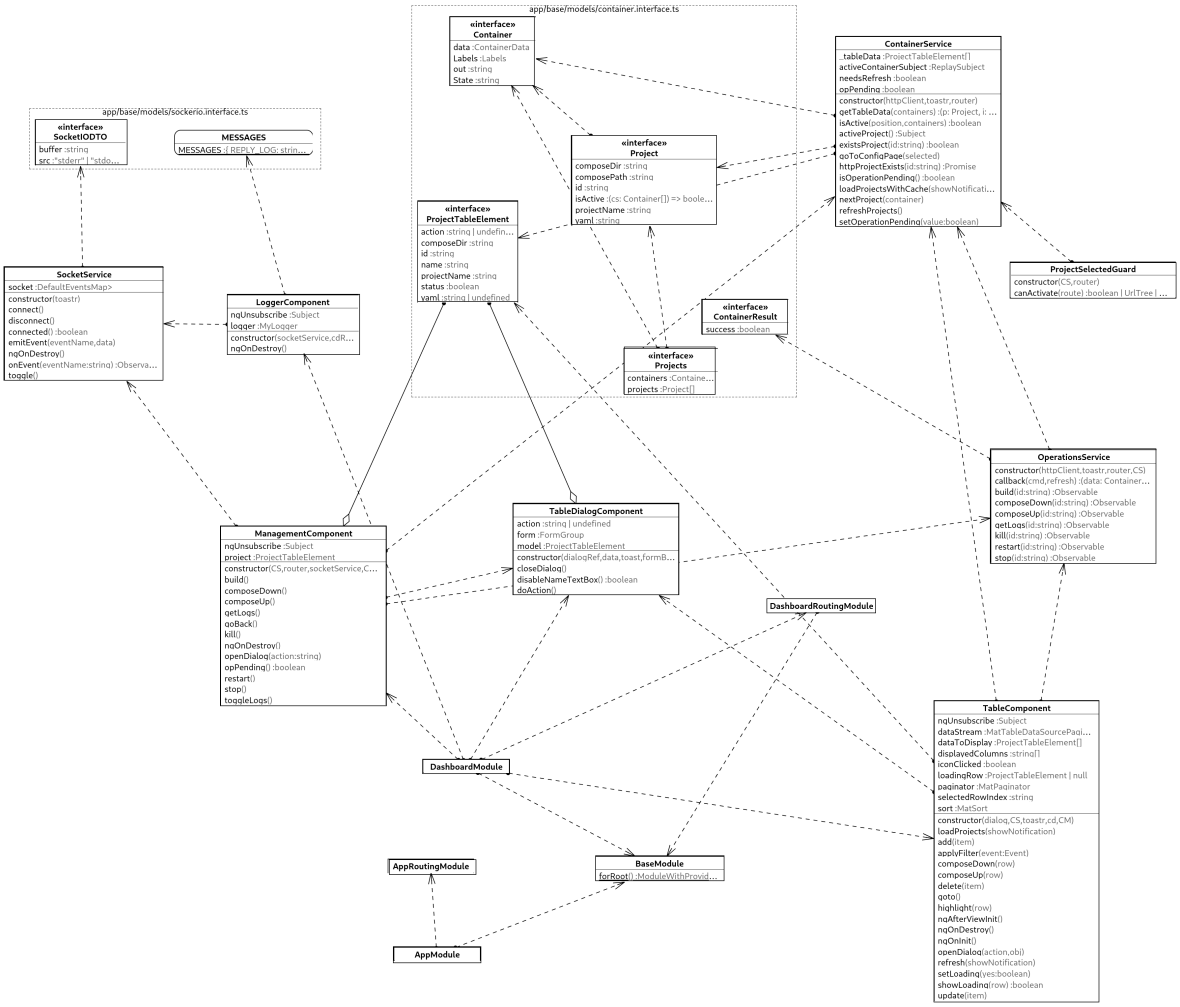


Figure 14. Angular Container Management Module

D. Appendix: Code organization

D.1. Back-end Code organization

As it is shown in Figure 15, the backend is organized around the MVC pattern and some of its features and advantages were described in Section 3.3.

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

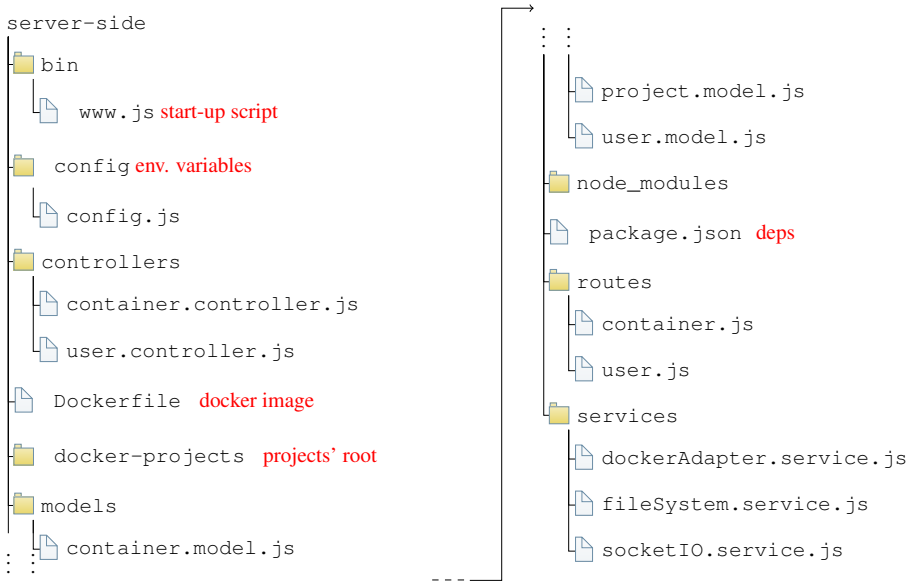


Figure 15. Server side code organization

D.2. Frontend code organization

The folder **frontend** holds all client side Angular code, is shown in Figure 16 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/base** contains the following reusable sub-components:
  - **Dashboard**: contains left hand side menu which can be toggled on and off.
  - **Authentication Guard**: ensures that the user is logged in prior to accessing specific resources.

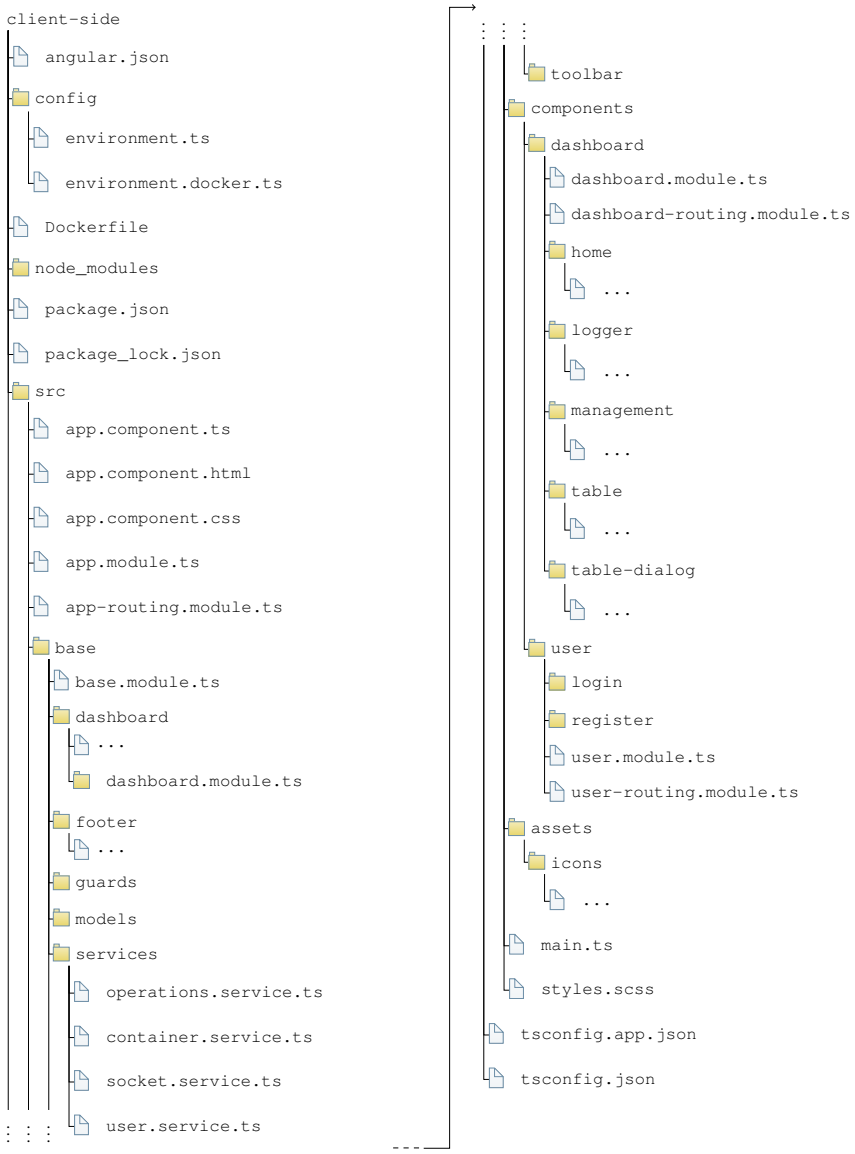


Figure 16. Client side code organization

  - **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**: defines the main features: login and registration, management of containers, home and documentation pages.

E. Appendix: Deployment

Listing 1. 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
      - MONGO_DB_DATABASE=docker-ui
      - USING_DOCKER=true
    links:
      - database

  database: # name of the third service
    image: mongo # specify image to build container from
    container_name: docker-ui-mongo
    environment:
      - MONGO_INITDB_ROOT_USERNAME=admin-user
      - MONGO_INITDB_ROOT_PASSWORD=admin-password
      - MONGO_DB_USERNAME=admin-user1
      - MONGO_DB_PASSWORD=admin-password1
      - MONGO_DB=docker-ui
    volumes:
      - ./mongo:/home/mongodb
      - ./mongo/init-db.d/:/docker-ui-entrypoint-init-db.d/
      - ./mongo/db:/data/db
    ports:
      - "27017:27017" # specify port forewarding


  client: # name of the first service
    image: razvanfv/docker-ui:client
    container_name: docker-ui-client
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    ports:
      - "4200:4200" # <host>:<container>
    links:
      - server
```

The deployment process creates three services: the **server**, the **database** and the **client**. Some of the more interesting settings are described below:

- **image**: the context used as an image. Can be an image from Docker Hub or a local *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. 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.