Software Design 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 6, 2022

Abstract

This document is the Software Design Document (SDD) for MyUI, a web-based application that generates user-specific UIs based on the underlying dataset and user permissions. This product has been developed by a group of students at the Eindhoven University of Technology as part of their Software Engineering Project. The design decisions presented in this document comply with the requirements that have been stated in the User Requirements Document.

Contents

Document status sheet

General

Document title:	Software Design Document	
Document identifier:	SDD/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 history

Date	Authors	Changes
20-06-2022	Yemoe Thoumoung	Initial Abstract and Introduction
20-06-2022	Yemoe Thoumoung	Initial system overview
25-06-2022	Jeroen Hellenbrand	Chapter 3.4
26-06-2022	Jeroen Hellenbrand, Samuel Nijsten	Chapter 3.6
27-06-2022	Samuel Nijsten	Chapter 4 and Appendix A
28-06-2022	Nishad Guha	Chapter 3.5
30-06-2022	Jeroen Hellenbrand, Richard Farla	Chapter 3.1
30-06-2022	Samuel Nijsten, Quinten Kustermans	Chapter 3.2
01-07-2022	Samuel Nijsten, Jeroen Hellenbrand, Jeroen Lamme, Yemoe Thoumoung, Adriana Radu, Eren Aydoslu	Chapter 3.3
02-07-2022	Everyone	Revision of Chapters 1 and 2
06-07-2022	Everyone	Implement supervisor feedback

Document change records

Version	Date	Section	Authors	Reason
0.1	02-07-2022	All	Everyone	Initial version
0.2	06-07-2022	All	Everyone	Implement supervisor feedback

1 | Introduction

1.1 | Purpose

The Software Design Document (SDD) provides an overview of the design of MyUI as well as its relations to other systems. This document will also provide a high-level architectural overview of MyUI. It provides motivation and alternatives to the design decisions that were made during the project. An estimation of the resources required to run MyUI is also given. This document is intended for professionals in the software development sector.

1.2 | Scope

MyUI is developed by a group of Bachelor 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. Hasura is an open-source web application server that connects to a Postgres database to give instant real-time GraphQL APIs. Hasura supports the query language GraphQL, 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 the place and size of data representations. There currently exists software built on top of Hasura that allows users to visualize their data, called MotorAdmin [1]. MotorAdmin is used to deploy an admin panel containing graphs, tables, and other data representation elements to analyze the data of a particular database. However, this software lacks a crucial point: user-level personalization. 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
Application element	A component of the front-end of the application. This includes all elements the user sees on their screen when using the application, such as text and images.
Base tables	A set of pure tables visualized in MyUI by default. There is precisely one base table for each table in the database that the user currently logged in to MyUI has access to. Each base table is visualized on a dedicated page via a grid view.
Bearer token	The server typically generates the bearer token in response to a login request. It is a secret string. When making requests to protected resources, the client must include this token in the Authorization header.
Dashboard	A particular page of which the contents and layout are rendered based on the stored preferences for a particular user (Dashboard Configuration).
Dashboard configuration	Set of preferences constructed in MyUI used to render a dashboard.
Dashboard configuration file	A file containing the structure and parameters necessary to render a dashboard constructed using MyUI.
Database engine	A software platform that allows you to quickly build, test, and distribute apps. It delivers software in independently executable packages called containers.
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.
Introspection query	A query to fetch which resources are available in the present API schema.
Open-source software	Software that, depending on the license, is available to anyone to use, study, modify and distribute for any purpose.
Page	An abstract term that is used to refer to a collection of application elements that are visually presented to a user.
Permission profile	Set of permissions to access information in the database assigned to a specific role.
Pure table	A table whose structure is contained in one of the tables on the database and where each row can be traced back to its original record in the database. In essence, the column names of this table are a subset of the column names of one of the tables in the database, and the table includes the primary key of the database.
REST	In a Client-Server architecture, Representational State Transfer (REST) refers to a software architectural style that describes a consistent interface between disconnected Internet components.
Static element	An application element that contains static data. Examples include text fields, images, or headers/footers. These elements do not interact with the contents of the database or the application.
User configuration file	A file containing the set of all stored Dashboard Configurations of a user.
Working area	Area of the UI where all application elements are added, excluding the sidebar and header.

1.3.2 | Acronyms and abbreviations

Acronym	Definition
API	Application Programming Interface
DBMS	Database Management System
DNS	Domain Name System
ERD	Entity Relationship Diagram
SDD	Software Design Document
SLAT	Second Level Address Translation
SSE3	Streaming SIMD Extensions 3
STD	Software Transfer Document
SUM	Software User Manual
URD	User Requirements Document

1.4 | List of references

[1] "Motoradmin." [Online]. Available: https://www.getmotoradmin.com/

[2] Eindhoven University of Technology, "MyUI: Software User Manual," 2022.

[3] M. Seidl, M. Scholz, C. Huemer, and G. Kappel, *UML @ Classroom: An Introduction to Object-Oriented Modeling.* Springer, 2015.

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

[5] ——, "MyUI: Software Transfer Document," 2022.

[6] "Ip." [Online]. Available: https://www.npmjs.com/package/ip

[7] "Bearer authentication." [Online]. Available: https://swagger.io/docs/specification/authentication/bearer-authentication/#:~:text=The%20bearer%20token%20is%20a,Authorization%3A%20Bearer

[8] "Authentication using jwt." [Online]. Available: https://hasura.io/docs/latest/graphql/core/auth/authentication/jwt/

[9] G. BT, "Hasura authentication explained," Sep 2021. [Online]. Available: https://hasura.io/blog/hasura-authentication-explained/#admin-secret-setup

[10] "Javascript with syntax for types." [Online]. Available: https://www.typescriptlang.org/

[11] "React – a JavaScript library for building user interfaces." [Online]. Available: https://reactjs.org/

[12] [Online]. Available: https://angular.io/

[13] "Vue.js - The Progressive JavaScript Framework: Vue.js." [Online]. Available: https://vuejs.org/

[14] "Ant design of React - Ant Design." [Online]. Available: https://ant.design/docs/react/introduce

[15] "The React Component Library You always wanted." [Online]. Available: https://mui.com/

[16] M. Otto and J. Thornton, "Bootstrap - The most popular HTML, CSS, and JS library in the world." [Online]. Available: https://getbootstrap.com/

[17] Parvez, "Ant.design - best react UI framework / component library," Jul 2019. [Online]. Available: https://ansariparvez.medium.com/ant-design-best-react-ui-framework-component-library-2e004a33f61f

[18] "Learn once - translate everywhere." [Online]. Available: https://www.i18next.com/

[19] Node.js, "About — node.js." [Online]. Available: https://nodejs.org/en/about/

[20] "Next.js - The React Framework for Production." [Online]. Available: https://nextjs.org/

[21] "Next.js boilerplate setup: Next.js Graphql Serverless tutorial." [Online]. Available: https://hasura.io/learn/graphql/nextjs-fullstack-serverless/setup/

[22] "Comparison: React query vs SWR vs Apollo vs RTK query vs react router." [Online]. Available: https://react-query.tanstack.com/comparison

[23] "Databases." [Online]. Available: https://hasura.io/docs/latest/graphql/core/databases/index/

[24] "CITUS data: Distributed Postgres. at any scale." [Online]. Available: https://www.citusdata.com/

[25] "NPM." [Online]. Available: https://www.npmjs.com/

[26] Yarnpkg, "Use it." [Online]. Available: https://yarnpkg.com/package/bcrypt

[27] "Jest a delightful javascript testing framework." [Online]. Available: https://jestjs.io/

[28] Yarnpkg, "Yarn — home." [Online]. Available: https://yarnpkg.com/

[29] "Docker." [Online]. Available: https://www.docker.com/

[30] "Docker - market share, competitor insights in containerization." [Online]. Available: https://www.slintel.com/tech/containerization/docker-market-share

[31] "Install Docker Desktop on windows," Jun 2022. [Online]. Available: https://docs.docker.com/desktop/windows/install/

[32] "Install Docker Desktop on mac," Jun 2022. [Online]. Available: https://docs.docker.com/desktop/mac/install/

[33] "Install Docker Desktop on linux," Jun 2022. [Online]. Available: https://docs.docker.com/desktop/linux/install/

[34] "Sap help portal - system requirements for clients." [Online]. Available: https://help.sap.com/docs/SAP_ASE/244b731a316a4de0ad1dd618937b0f8e/a6e73750bc2b1014b9a180e5a9ddafa1.html?version=16.0.0.0

1.5 | Overview

In the remainder of this SDD, we compare MyUI to other software and explain its purpose and future development. The relation to current projects serves as a comparison between MyUI and similar applications, while the relation to successor projects elaborates on the possible future development of the application.

We describe the system architecture and elaborate on the logical modal description of the application. This logical modal includes a class diagram of the application's components. To further explain the logic of the application, the state dynamics are demonstrated with sequence diagrams. In addition, the data model used by the application, external interfaces, and the design rationale of the application is clarified.

To aid future developers of MyUI, the feasibility and resource estimations are discussed. This includes the minimum requirements to run both the development and client systems. A performance measure is given based on MyUI's performance needs.

2 | System overview

In this chapter, the place of MyUI in the current software landscape, its future development, and its purpose will be discussed.

2.1 | Relation to current projects

The ability to create a custom UI for database management through Hasura is already available. Our customer, Connected.Football, has used existing applications in order to generate user interfaces. However, these existing applications have shortcomings. For example, the tool Motor Admin [1]. This is one of the tools Connected.Football has used in the past. It is an open-source project that serves as a no-code admin panel using Hasura. While MyUI uses Motor Admin as a source of inspiration, it offers two important additional functionalities that are not present in Motor Admin.

Firstly, the ability to have different permission profiles for the same database is not supported by Motor Admin in the free version. MyUI implements permissions as a base feature, thereby allowing centralized databases to be used. In these databases, different users are allowed to access different data while it is still possible to manage the database centrally. Moreover, the ability to restrict users from deleting, inserting or editing data can be very useful, making data available while securing its correctness.

Secondly, MyUI couples the customization of the UI to every user individually. In Motor Admin every user that has access to a certain database can edit the UI, which is global for that database. Either a user needs to configure the UI to their personal preference every time another user edits the UI, or they would use a UI that is not suited for their personal preferences. The use of a personal UI would solve the aforementioned problems. In MyUI users have full control over their own UI, regardless of the permissions, they have on the database.

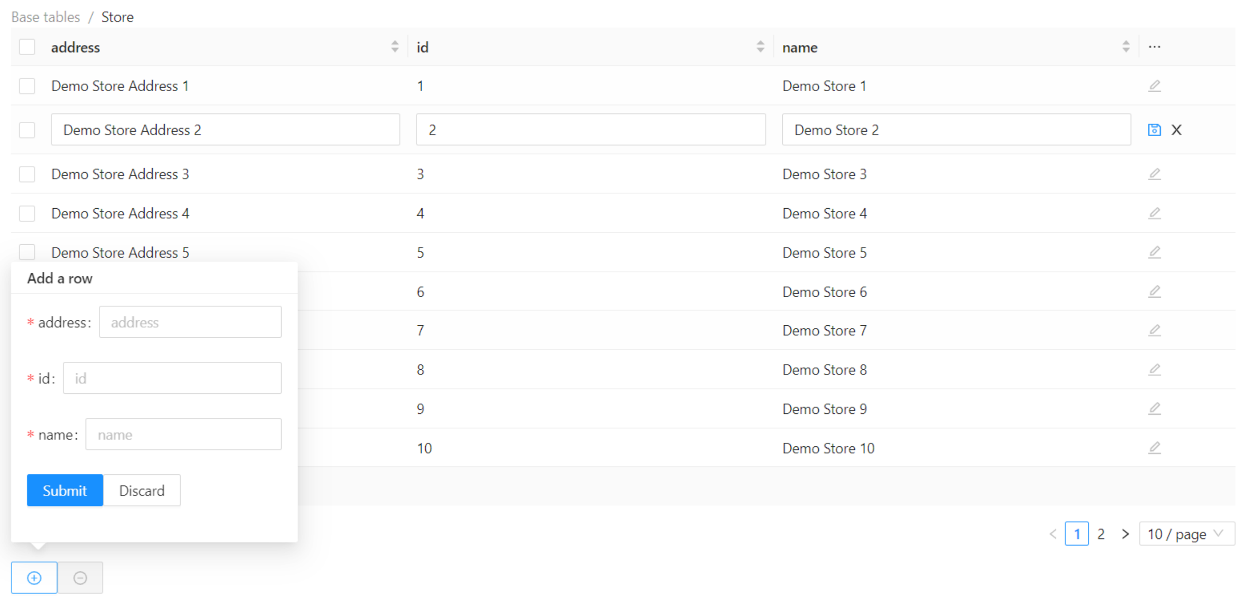
2.2 | Relation to predecessor and successor projects

There are no predecessor projects of MyUI. By the nature of being an open-source project, it is possible for many successor projects to be built upon the basis of MyUI. Our customer has stated that they intend to continue development of the MyUI application. A concrete example of this development is the addition of a user management system within MyUI, and to replace the current user authentication system.

2.3 | Function and purpose

The purpose of MyUI is to give users the ability to construct a UI for database management using only UI interactions. This created UI is stored for every user individually. The data shown in the UI is in accordance with the user's permission profile. The actions can be divided into the following functionalities.

- Viewing data: Users can view data from the database in the base tables or in tables they have queried and placed into their own custom dashboards. Only data they have select access to will be shown. Ordering on a single column is also possible (Figure 2.1).
- Editing data: With the right permissions users can insert, update and delete entries in pure tables. This can be done without writing code, only UI actions are needed (Figure 2.1).
- Customization: Users can customize how tables are viewed by changing the ordering of data within a column. A column can have no order, or be ordered ascendingly or descendingly. Users can create and delete custom dashboards that can contain tables filled with data from a GraphQL query entered by the user. The users are free to add more tables and order those tables on a dashboard (Figure 2.2).



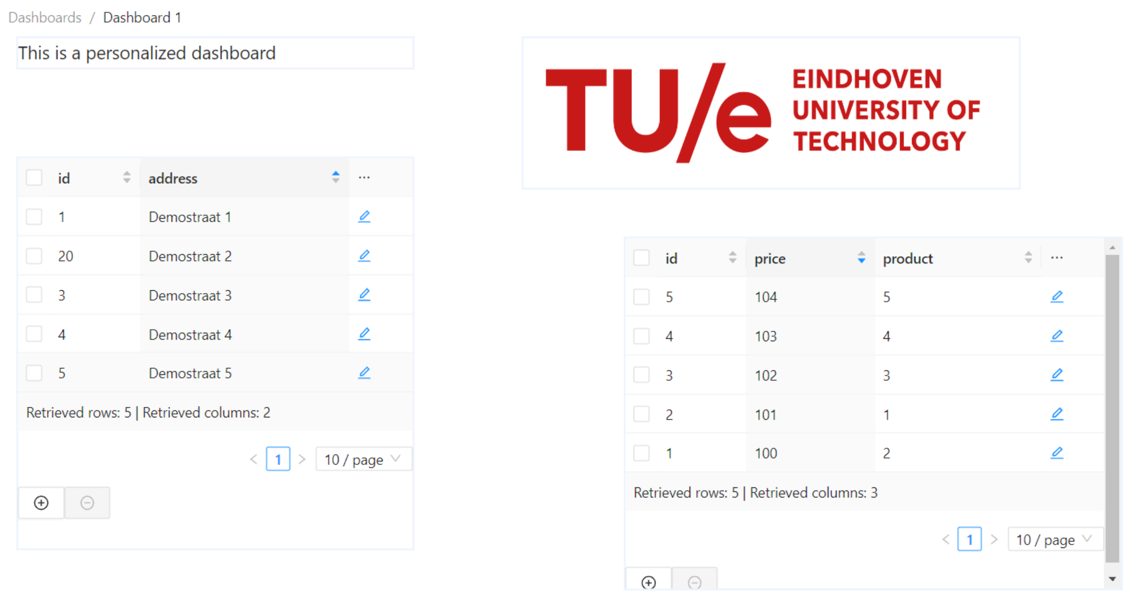
Base tables / Store

	address	id	name	···
	Demo Store Address 1	1	Demo Store 1	
	Demo Store Address 2	2	Demo Store 2	
	Demo Store Address 3	3	Demo Store 3	
	Demo Store Address 4	4	Demo Store 4	
	Demo Store Address 5	5	Demo Store 5	
		6	Demo Store 6	
		7	Demo Store 7	
		8	Demo Store 8	
		9	Demo Store 9	
		10	Demo Store 10	

Add a row

* address:	address
* id:	id
* name:	name

Figure 2.1: The viewing and editing of data



Dashboards / Dashboard 1

This is a personalized dashboard

	id	address	···
	1	Demostraat 1	
	20	Demostraat 2	
	3	Demostraat 3	
	4	Demostraat 4	
	5	Demostraat 5	

Retrieved rows: 5 | Retrieved columns: 2

	id	price	product	···
	5	104	5	
	4	103	4	
	3	102	3	
	2	101	1	
	1	100	2	

Retrieved rows: 5 | Retrieved columns: 3

Figure 2.2: Displaying customized data representations on a dashboard

2.4 | Environment

MyUI is a web-based application, developed and tested on Chrome version 101, Mozilla Firefox version 99, Microsoft Edge version 100, and Safari version 15.

MyUI has two main types of users, namely admins and end-users. To make use of the functionalities of MyUI, a user is required to log in with valid credentials. These credentials can be found in the users table in Hasura. It is not possible to create an account from within MyUI. The existence of an account depends solely on the users table. MyUI admins can modify the users table to modify who can use MyUI. These main types of users are characterized as follows:

- MyUI admins[1] are allowed to see all tables that are present in the database. Due to their role, they can perform insertion, deletion, and updates on any row of any table. Furthermore, they can change the global system configuration of the MyUI application.
- End-users[1] use the MyUI web application to generate data representations in order to gain an overview of and modify the data stored on the Postgres server. End-users can add text, media, and gridviews to their dashboard. Gridviews are a tabular representation of the data retrieved by a user-submitted GraphQL query. The data of these tables can then be modified by end-users, and those modifications will be saved in the database. Each end-user has a role assigned to them, which corresponds to a permission profile. Based on this permission profile, an end-user is capable of:
  - Viewing a specific table,
  - Inserting an entry into a specific table,
  - Updating an entry of a specific table,
  - Deleting an entry from a specific table.

Moreover, there is an administrator of the PostgreSQL and Hasura instance[1] who manages and maintains the Postgres server as well as Hasura. In this Hasura instance, they can modify the database on the Postgres server. These administrators define the roles and the corresponding permission profiles within Hasura.

2.5 | Relation to other systems

MyUI is built upon the permission system and the GraphQL API that Hasura provides. In the next section, in Figure 3.1, it will be illustrated where MyUI stands in relation to Hasura and the database. MyUI is mainly focused on the graphical representation of data and who is allowed to access certain data. Furthermore, the handling of queries is done by Hasura. Changing user permissions can only be done by administrators within Hasura, MyUI does not support this feature itself. While data handling for MyUI is done by Hasura, a Postgres server is still needed to store the database from which Hasura fetches the data. Changes to the database structure need to be done either within Hasura or in the Postgres database itself. The relation between the systems is explained futher in Section 3.

[1] Note that these roles are not mutually exclusive, one can assume multiple roles.

3 | System architecture

The architectural layout of MyUI is described in this chapter. We provide an abstract, logical model of the application and a representation of the state dynamics using sequence diagrams to provide extra context for the design of MyUI. MyUI's data model and the external interfaces it uses are both described. Finally, the justification for each design choice made throughout the applications' development will be addressed.

3.1 | Architectural design

MyUI is a front-end-driven application that communicates with a database through Hasura. The major components of MyUI's front-end, their relationships, and how they interact are described in this section.

3.1.1 | Global architecture design

MyUI is a front-end application that runs in the context of a PostgreSQL database that is accessed via the GraphQL API made available by Hasura. The figure below describes the global architecture design of MyUI and its context. The rest of Section 3.1 will discuss the individual components that comprise MyUI. Hasura and the PostgreSQL database form the context in which MyUI can be plugged in. A further elaboration of this context can be found in Section 3.5.1.

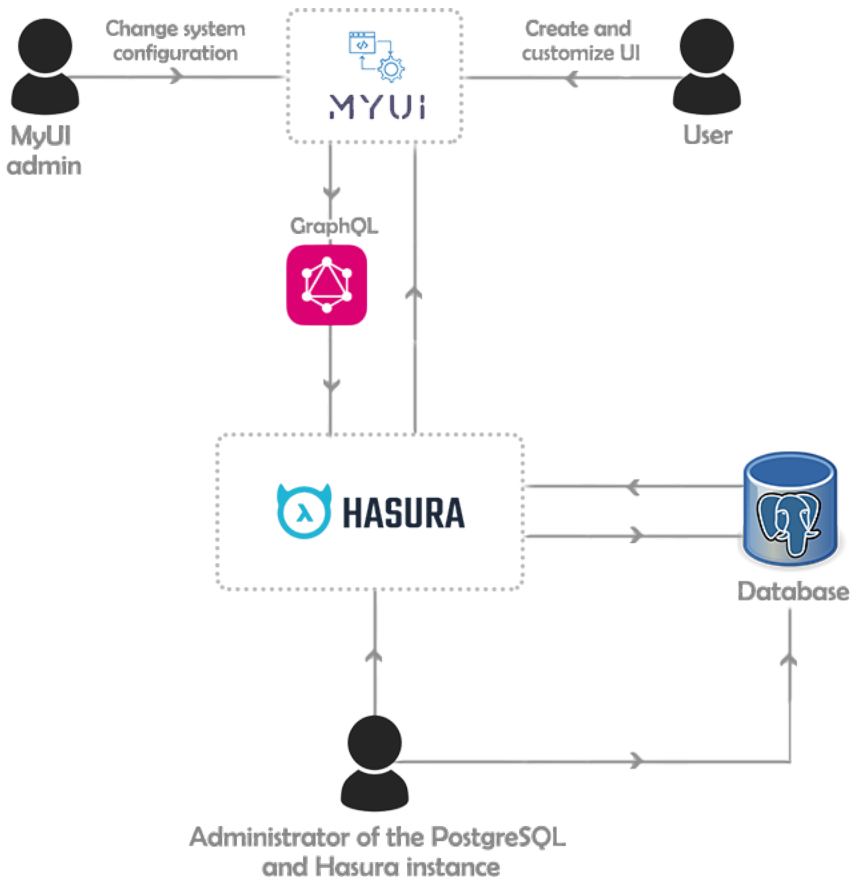

Change system configuration
MyUI admin
MYUI
Create and customize UI
User
GraphQL
HASURA
Database
Administrator of the PostgreSQL and Hasura instance

Figure 3.1: The environment of the application

Users of MyUI, as defined in the previous section, interact with the database through Hasura via GraphQL queries written in MyUI. The administrator of the PostgreSQL and Hasura instances manages the database and Hasura instances directly.

3.1.2 | Front-end

When a user tries to access the MyUI application, they are first presented with a login page. After logging in, the user has access to a single-page application, which will be referred to as *Index* from here onwards. *Index* consists of three main components, as seen in Figure 3.2. These components are always displayed in *Index*, but their contents depend on the state of the application. In order to gain more insight into the components given in this diagram and the relations between them, a logical model is given in Section 3.2.

The login page, the separate components, and the coherence between them will be explained further in this section. To find detailed explanations of how each separate component can be used, MyUI's SUM [2] should be consulted.

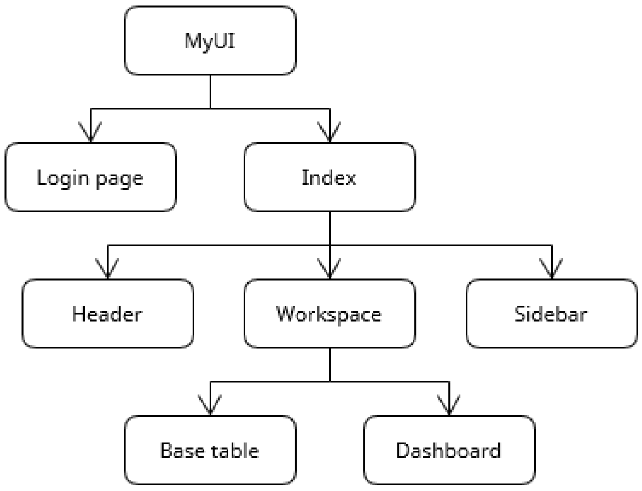


Figure 3.2: The front-end architecture

Login page

The login page (Figure 3.3) provides the user with two input fields to enter their credentials and a submit button. When the user presses this button, a call to the back-end is made to attempt logging in the user with the provided username and password. The back-end will verify the credentials, and only give access to the application if the credentials were valid.

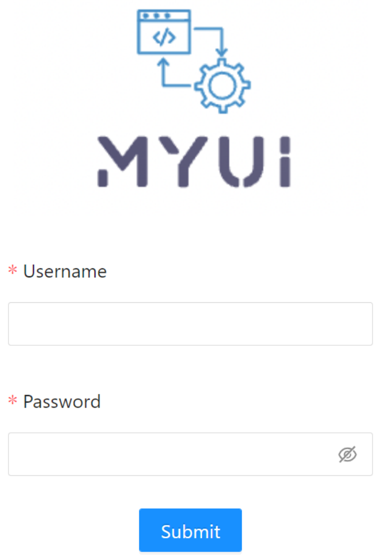


Figure 3.3: MyUI's login page

Header

At any time, the header of the application (Figure 3.4) contains MyUI's logo and a user icon. When the user clicks on this icon, a menu is unfolded for the user to change the application's language and log out of the application. Whenever the user has a dashboard loaded in their workspace, the header also displays a gear icon. Clicking on this icon toggles edit mode for said dashboard.

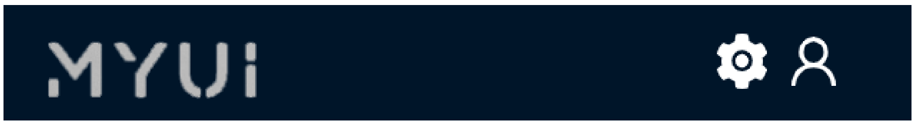


Figure 3.4: MyUI's header

Sidebar

MyUI has two types of sidebars, the navigation and edit mode sidebar (Figure 3.5). Outside of edit mode, the navigation sidebar is always visible. It contains two dropdown menus; one for base tables and one for dashboards. When an item from one of these dropdown menus is clicked, the respective base table or dashboard is loaded in MyUI's working area. In addition, the dashboards dropdown menu contains an add and delete button to create a new dashboard or delete an existing one.

The edit mode sidebar is only shown when the user is in edit mode for a dashboard. This sidebar contains two draggable elements, the static element, and the grid view element. When the user drags and drops one of these in their workspace, the respective element is added to the dashboard they are currently editing. Furthermore, a save button to save the dashboard changes and a download button to download the dashboard configuration are present.

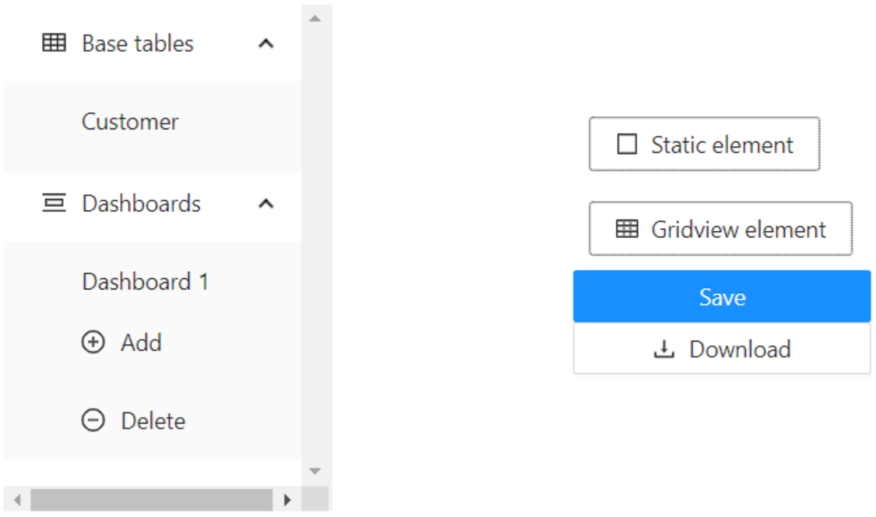
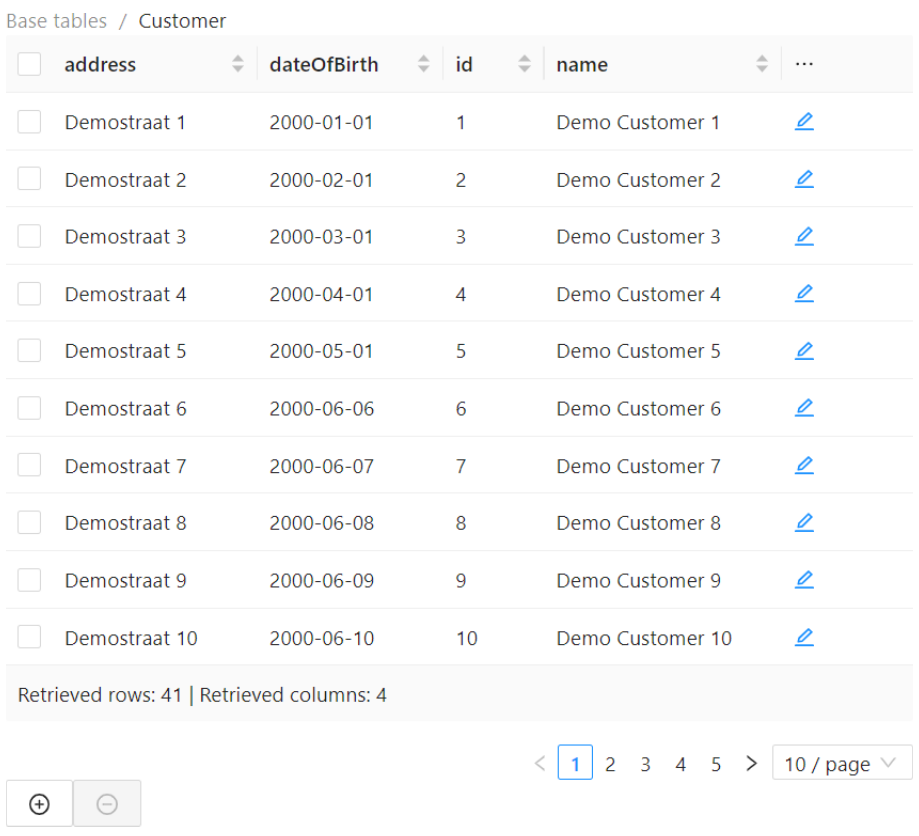


Figure 3.5: MyUI's navigation sidebar (left) and edit mode sidebar (right)

Workspace

After logging in, the workspace is empty. Either a base table or a dashboard can be loaded in the workspace. When a user loads a base table (Figure 3.6), the workspace renders the corresponding table containing all the columns that the user is allowed to see. All table operations supported by MyUI can be performed here.

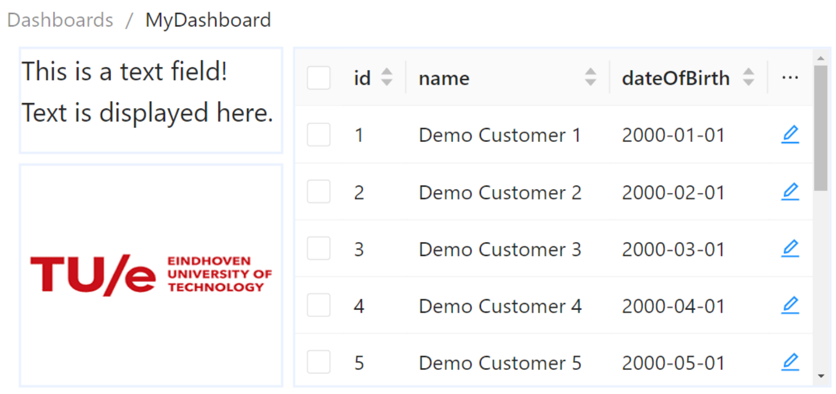
When a user loads a dashboard (Figure 3.7), the dashboard will be displayed based on the dashboard configuration saved for the user. If the user toggles edit mode, they will be able to edit the current dashboard by adding, deleting, resizing, and editing the dashboard's elements.


Base tables / Customer

address	dateOfBirth	id	name
Demostraat 1	2000-01-01	1	Demo Customer 1
Demostraat 2	2000-02-01	2	Demo Customer 2
Demostraat 3	2000-03-01	3	Demo Customer 3
Demostraat 4	2000-04-01	4	Demo Customer 4
Demostraat 5	2000-05-01	5	Demo Customer 5
Demostraat 6	2000-06-06	6	Demo Customer 6
Demostraat 7	2000-06-07	7	Demo Customer 7
Demostraat 8	2000-06-08	8	Demo Customer 8
Demostraat 9	2000-06-09	9	Demo Customer 9
Demostraat 10	2000-06-10	10	Demo Customer 10

Retrieved rows: 41 | Retrieved columns: 4

Figure 3.6: MyUI's workspace containing a base table


Dashboards / MyDashboard

This is a text field!
Text is displayed here.

id	name	dateOfBirth
1	Demo Customer 1	2000-01-01
2	Demo Customer 2	2000-02-01
3	Demo Customer 3	2000-03-01
4	Demo Customer 4	2000-04-01
5	Demo Customer 5	2000-05-01

Figure 3.7: MyUI's workspace containing a dashboard

3.2 | Logical model description

This section contains the logical model description of MyUI. This includes the class diagram and the associated components. Note that the class diagram and its associated components are a logical abstraction from the actual implementation.

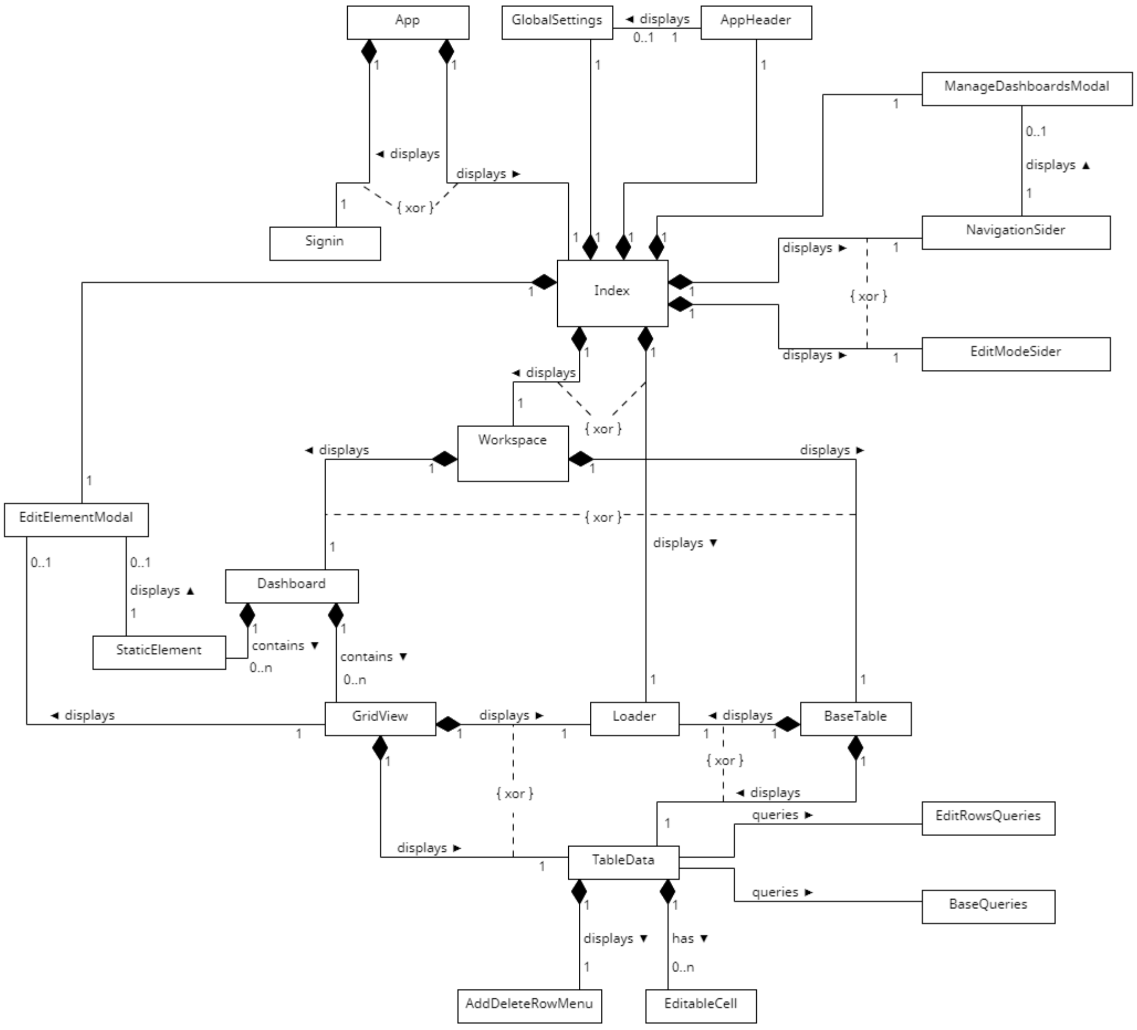


Figure 3.8: The Class Diagram of MyUI

The class diagram of MyUI is given in Figure 3.8. It serves as a concise overview of the static structure of MyUI. For each component in this diagram, an explanation of the associated parts of the class diagram will be given, as well as an elaboration of its relations and the relevant methods.

In order to provide a logical overview as opposed to an implementation-focused class description, a number of abstractions were made. The most important of these is the following: MyUI makes use of states, which are built-in React objects that are used to contain data or information about the component. These states are used for a multitude of reasons, but mainly to exchange information between components. To provide a clearer overview, all individual states have been accumulated into a composite global variable called STATE. Moreover, the return statement of each component, which executes the rendering of said component, has been modeled as a function named display with a parameter STATE. Lastly, the methods and descriptions given in the following subsections are a general abstraction from the functionality of their component. They do not correspond to actual methods.

App

The App page is the root of MyUI. As can be seen in Figure 3.8, the App page participates in a *displays* relation with the Signin page and the Index page, as it renders either of these pages depending on whether the credentials have been validated, which in turn is modeled by the *XOR* constraint on this relation.

Table 3.1: App: Functionality

Name	Description
isAuthenticated	Checks if the user has already been authenticated.
display	Renders either the *Signin* page or the *Index* page, depending on whether the user has been authenticated.

Signin

The Signin page is responsible for rendering the sign-in page and handling the sign-in functionality of MyUI. This page is a straightforward login page with a username field and a password field, accompanied with a submit button. The page is part of a *displays* relation with App, which renders the page when an unauthorized user tries to access MyUI.

Table 3.2: Signin: Functionality

Name	Description
authenticate	Checks if the submitted credentials are correct.
display	Renders the log in page of MyUI. This includes MyUI's logo and input fields for both the username and password.

Index

Index is the main page of MyUI, and is responsible for rendering most components. As can be seen in the Class Diagram (Figure 3.8), a multitude of components are dependent or even included in Index. Moreover, every component except for App and Signin has index as an ancestor.

Index is involved in a multitude of relations, listed below

- A *displays* relation with the NavigationSider and the EditModeSider components. The associated *XOR* constraint indicates that either the navigation sidebar or the edit mode sidebar is rendered, based on the STATE.
- A *display* relation with the AppHeader component. This indicates that the Index page renders the app header.
- A *display* relation with the Workspace and Loader component. The associated *XOR* constraint indicates that the Index page either renders the throbber or the workspace, depending on the STATE.

Besides these relations, both the EditElementModal and ManageDashboardModal are contained in the Index page.

Table 3.3: Index: Functionality

Name	Description
displayBaseTable	Updates the STATE of the application, particularly it sets the display mode to displaying a base table.
displayDashboard	Updates the STATE of the application, particularly it sets the display mode to displaying a dashboard.
toggleEditMode	Updates the STATE to toggle edit mode.
saveDashboardChanges	Writes the current state of the dashboard to the user configuration file.
displaySider	Returns either the sidebar for edit mode or the sidebar for navigating through MyUI based on the STATE, particularly whether the user is in edit mode.
display	Renders the main application UI, based on the STATE. There are a number of factors within this state that play an important role: ■ The status of the user configuration file. If and only if this is successfully loaded according to the STATE, the application will display the UI of MyUI. ■ Whether the STATE indicates that the application is in edit mode. The application renders a sidebar based on this. ■ Whether the STATE indicates that the data of the sidebar is loaded and ready, if not, the application will show a loader in the sidebar's place.

AppHeader

The AppHeader component is responsible for rendering, and handling, the functionality of MyUI's header. It is rendered as an application header that is visible at all times after a user has logged in. Contained in this header are the change-mode button, MyUI's logo, and the user menu. The latter may update the STATE to also show the Global settings menu, such that the appropriate modal for this is displayed. Lastly, the language switching and the logging out functionalities are also handled by the AppHeader component.

As can be seen in Figure 3.8, the AppHeader component participates in a *displays* relation with the Index page, which renders this header at all times. Furthermore, it participates in a *displays* relation with GlobalSettings, as Appheader is responsible for rendering the associated modal when an administrator tries to configure the global settings of MyUI.

Table 3.4: AppHeader: Functionality

Name	Description
updateLanguage	Updates the STATE and the application text to adhere to the selected language.
rotateGear	Rotates the gear in the header of MyUI, depending on the STATE, particularly whether the application is in edit mode.
display	Renders the header based on the current STATE, particularly whether the application is in edit mode.

GlobalSettings

The GlobalSettings component is responsible for rendering the global setting drop-down menu. In this menu, an administrator can change the global media setting. As can be seen in Figure 3.8, the GlobalSettings component participates in a *displays* relation with the AppHeader component. This indicates that AppHeader renders this component when an admin wants to edit the global settings of MyUI.

Table 3.5: GlobalSettings: Functionality

Name	Description
changeSettings	Updates the STATE and the global settings modal according to the option selected by the user.
display	Renders the global settings modal with the global setting dropdown menu, containing settings that can be edited by the admin user.

ManageDashboardsModal

The ManageDashboardsModal component for rendering and handling the functionality of the modal menus shown to either add or remove a user's dashboards. The contents of the modal depend on the STATE, particularly whether the modal is shown to add or delete a dashboard. When deleting a dashboard, the modal includes a text input field, a confirm, and a cancel button. When adding a dashboard, it also includes these components, with the addition of an upload file input field.

ManageDashboardsModal participates in a *displays* relation with Index, indicating that Index is responsible for showing the user this modal with the appropriate contents when they wish to either add or delete a dashboard.

Table 3.6: ManageDashboardsModal: Functionality

Name	Description
isValidDashboardName	Checks if the provided name is a valid dashboard name.
addDashboard	Updates STATE to include the newly added dashboard.
removeDashboard	Updates STATE to remove the recently deleted dashboard.
display	■ Handles the functionality of the dashboard modal. This includes adding a dashboard, deleting a dashboard, and the cancellation of both addition and deletion. ■ Renders the manage dashboard modal. This includes the deletion prompt or addition prompt, depending on the STATE.

NavigationSider

The NavigationSider component is responsible for rendering and handling the functionality of the navigation sidebar of MyUI. The navigation sidebar offers tools to the user to switch between various parts of MyUI: the user can either fold out the base table menu or the dashboard menu and interact with their respective contents. Upon such interactions, the user is navigated to the associated part of MyUI, which can be either a base table, a dashboard, or the landing page.

As can be seen in Figure 3.8, the NavigationSider component participates in a *displays* relation with the Index page. This indicates that the Index page is responsible for rendering the navigation sidebar, which is done based on the STATE, particularly whether the user is in edit mode. If they are not, this sidebar will be shown, otherwise, the edit mode sidebar is rendered, as modeled by the *XOR* constraint.

Table 3.7: NavigationSider: Functionality

Name	Description
display	Renders the sidebar used for navigation based on the current STATE, particularly whether the application is in edit mode. The method only renders the component if this is not the case.

EditModeSider

The EditModeSider component is responsible for rendering and handling the functionality of the edit mode sidebar of MyUI. In the edit mode sidebar, the user can drag both static elements and grid view elements onto the displayed dashboard. Moreover, the user can save the changes made to the dashboard and download the dashboard configuration file using two dedicated buttons in this menu.

As can be seen in Figure 3.8, the EditModeSider component participates in a *displays* relation with the Index page. This indicates that the Index page is responsible for rendering the edit mode sidebar, which is done based on the STATE, particularly whether the user is in edit mode. If they are in edit mode, this sidebar will be shown, otherwise, the navigation sidebar is rendered, as modeled by the *XOR* constraint.

Table 3.8: EditModeSider: Functionality

Name	Description
downloadDashboard	Prompts the user to select a location in their file system. Once the user has done so, downloads the current dashboard configuration file to the selected location.
display	Renders the sidebar used in edit mode to add components to a dashboard or to save a current dashboard layout. The method only renders the component if the STATE indicates that the application is in edit mode.

Loader

The Loader component is responsible for rendering a throbber. This component partakes in a *displays* relation with GridView, BaseTable, and Index, as can be seen in the Class Diagram (Figure 3.8). This indicates that the Loader is rendered conditionally based on whether the STATE indicates that some required data is not yet (completely) available for either of these three components. When this happens, the Loader is rendered as a placeholder component for this particular component until said data is available. When the STATE indicates the data is available, the appropriate component will be rendered instead. This behavior is modeled by the *XOR* constraint on these relations.

Table 3.9: Loader: Functionality

Name	Description
display	Renders the throbber used to represent loading in the UI.

Workspace

The Workspace component is responsible for rendering the workspace of MyUI. As can be seen in Figure 3.8, the Workspace component participates in three *displays* relations.

Two of these, the relation with the Dashboard and BaseTable components, are bounded by an *XOR* constraint. This illustrates that the Workspace renders either a dashboard or a base table. The final relation is with the Index page, indicating that the Index page renders the workspace.

Table 3.10: Workspace: Functionality

Name	Description
display	Renders the workspace in the UI. Based on the STATE, this workspace shows either a base table, a dashboard or an empty space with a welcome message.

Dashboard

The Dashboard component is responsible for rendering and handling the functionality of a dashboard. As can be seen in Figure 3.8, the BaseTable component participates in one *displays* relation with the Workspace component, which implies that the Workspace component renders the Dashboard, this is done based on the STATE.

The Dashboard component also participates in a *contains* relation with the StaticElement and GridView components. These relations are one-to-many, implying that a Dashboard can contain 0 to n static elements and gridviews. The *contains* relation indicates that the Dashboard component is responsible for rendering the elements with which it has such a relation, as well as providing the editing, moving, resizing, and deletion functionalities of said elements.

Table 3.11: Dashboard: Functionality

Name	Description
editElement	Prompts the user to provide content in order override the content of the element.
deleteElement	Deletes the element from the dashboard.
showDeleteConfirm	Prompts the user to confirm the deletion of the element from the dashboard.
saveChanges	Saves the changes made to the dashboard.
addElement	Adds a new element to the dashboard, and calls *editElement* in order to allow the user to edit it.
display	Renders a constructed dashboard in the working area.

EditElementModal

The EditElementModal component is responsible for rendering and handling the functionality of the edit element modal. This modal includes an input field, where an element's desired content is to be entered, and a cancel and confirm button. As can be seen in Figure 3.8, the EditElementModal component participates in a *display* relation with the StaticElement component and the Gridview component, indicating that these components render the edit element modal when a user wishes to edit their contents.

Table 3.12: EditElementModal: Functionality

Name	Description
defineContent	Defines the content of the element.
defineType	Defines the type of data being held by the element, either static or gridview.
checkFormat	Validates the format of the provided input.
display	Renders the edit element mode, containing a text input field, a cancel and a confirm button.

StaticElement

The StaticElement component is responsible for rendering a static element. A static element can contain both plain text and a URL. Depending on the STATE, it either renders a URL as text or media. StaticElement is part of two relations:

Firstly, it participates in a *displays* relation with EditElementModal, which describes the responsibility of StaticElement to show the user the edit element modal when they wish to edit the static element's contents.

Secondly, it has a *contains* relation with Dashboard. This indicates that a dashboard can contain a number of static element, the rendering of which is delegated to this Dashboard component.

Table 3.13: StaticElement: Functionality

Name	Description
convertURL	(Optionally) Converts a URL representing media into an HTML tag displaying said media, depending on the media display setting.
display	Renders an HTML tag containing the static element.

GridView

The GridView component is responsible for rendering the grid view element on a dashboard. As can be seen in Figure 3.8, the GridView component participates in a *displays* relation with the Loader and TableData component. This indicates that the GridView component is responsible for rendering either the Loader or the TableData component, this is done based on the STATE. As indicated by the *XOR* constraint, this is mutually exclusive. Furthermore, GridView has a *displays* relation with EditElementModal, describing the responsibility of GridView to render an appropriate edit modal for editing the gridview's contents when a user tries to do so.

Besides the *displays* relations, GridView also participates in a *contains* relation with the Dashboard component. This illustrates that a grid view is contained in a dashboard.

Table 3.14: GridView: Functionality

Name	Description
display	Renders either the TableData or the Loader component on the UI. Depending on the STATE, this may include the EditElementModal component.

TableData

The TableData component is responsible for rendering and handling the functionality of a table. This includes rendering the actual data and the number of rows and columns fetched. Furthermore, TableData renders UI elements for the following supported table operations:

- Sorting by attribute
- Pagination
- Selecting/deselecting a row
- Selecting/deselecting all rows on the page

Lastly, TableData respectively provides UI elements for the edit cell, add row and delete row functionality if the user permission profile allows these actions.

As mentioned above, TableData renders tables. This is in line with the Class Diagram (Figure 3.8), where TableData has a *displays* relation with GridView and BaseTable, indicating that these components delegate the actual displaying of the data, along with the aforementioned functionalities, to the TableData component.

Furthermore, the TableData component shares a *queries* relation with BaseQueries and EditRowsQueries. This relation represents the fact that these two components are responsible for sending their respective queries to the applicable endpoints on request of TableData. The received result is then to be processed by TableData into the tables with the functionalities as described above.

Moreover, TableData participates in a *has* with EditableCell, implying that a table rendered by TableData may render 0 to n editable cells, depending on the state, particularly whether the user is editing a row.

Lastly, TableData has a *displays* relation with AddDeleteRowMenu, which corresponds to the responsibility of TableData to show the add row menu or delete row menu when a user tries to perform the respective action.

Table 3.15: TableData: Functionality

Name	**Description**
saveEdit	Checks the content of the form for editing rows, and constructs a query to send a mutation to the database to execute any specified changes.
setEditDefaults	Sets the default values for the editing in a particular row.
displayTableInfo	Displays the amount of retrieved rows and columns in the table's footer.
display	Handles errors by Hasura and renders a table's content: ▪ Conditionally renders an additional error message with the respective content if the STATE indicates a Hasura error was received. ▪ Displays a warning if the STATE indicates the requested data could not be fetched. ▪ Displays a table if the data was fetched successfully.

BaseTable

The BaseTable component is responsible for rendering the base tables. As can be seen in Figure 3.8, the BaseTable component participates in a *displays* relation with the Workspace component, the TableData component, and the Loader component.

The first relation straightforwardly implies that the Workspace component renders the BaseTable, this is done based on the STATE. The latter two relations concern the actual rendering of the table and are bounded by an *XOR* constraint. This indicates that a BaseTable component renders the respective TableData component or a Loader depending on the STATE. If the data is not completely retrieved, a throbber is shown, otherwise the TableData component is rendered along with the UI elements for the available table operations.

Table 3.16: BaseTable: Functionality

Name	**Description**
display	Renders the base table in the workspace. Based on the STATE, either the base table or a throbber is displayed

EditableCell

The EditableCell component is responsible for rendering a cell that can be edited by the user. As can be seen in Figure 3.8, the EditableCell participates in a *has* relation with the TableData component, which renders this component in a cell's place when the user tries to edit a cell.

Table 3.17: EditableCell: Functionality

Name	**Description**
display	Renders the columns of a row as a group of editable cells, depending on whether the STATE indicates that this row is being edited.

AddDeleteRowsMenu

The AddDeleteRowsMenu component is responsible for rendering and handling the functionality of the add row and delete row menu for tables. The add row menu contains an input field for each column in the respective table, and a confirm and cancel button. The delete row menu contains only a confirm and cancel button.

The rendering of the respective menus is conditional on the user permission profile. In terms of functionality, the AddDeleteRowsMenu component constructs mutations used to either delete or add rows to the database.

As can be seen in Figure 3.8, the AddDeleteRowMenu participates in a *displays* relation with the TableData component, which renders this menu when the user respectively tries to add or delete a row.

Table 3.18: AddDeleteRowsMenu: Functionality

Name	**Description**
addRow	Adds a row to the current table and updates the STATE as well as the database accordingly.
deleteRow	Deletes a row in the current table and updates the STATE as well as the database accordingly.
display	Renders the add row menu or the delete row menu, depending on the STATE, particularly whether the user is adding or deleting a row.

BaseQueries

The BaseQueries component is responsible for constructing the basic queries for MyUI. This includes constructing queries to retrieve and update the user configuration file, constructing queries to retrieve the table names of the tables a user has access to according to their permission profile and retrieving the data of said tables.

As can be seen in Figure 3.8, the BaseQueries component participates in a *queries* relation with the TableData component. This indicates that BaseQueries is responsible for sending the queries it offers to the applicable endpoints on request of TableData and that the query result is then processed by TableData such that it can be displayed as a table.

Table 3.19: BaseQueries: Functionality

Name	Description
configurationQuery	Retrieves the user configuration file of the user.
updateUserConfiguration	Updates the configuration file of the user.
retrieveTables	Retrieves the names of the base tables for which the user has select permissions, and retrieves the corresponding data once this is done.

EditRowsQueries

The EditRowsQueries component is responsible for constructing the queries needed to edit a row. This includes checking if the user has the correct user permission profile, as well as constructing a query to send a mutation to the database to execute any specified changes. As can be seen in Figure 3.8, the EditRowsQueries component participates in a *queries* relation with the TableData component. This indicates that EditRowsQueries is responsible for sending its queries to the applicable endpoints on request of TableData, which then processes the result such that it can be displayed as a table.

Table 3.20: EditRowsQueries: Functionality

Name	Description
checkPermissions	Checks the user's permission for a table.
updateRow	Constructs the query to edit the respective row in the database after a user's edit.

3.3 | State dynamics

In this section, the state dynamics will be discussed. The use cases of MyUI will be described with the use of UML sequence diagrams, as described in UML@classroom [3]. In order to give a clear and concise overview, the sequence diagrams are based on the logical model as presented in Section 3.2. Therefore the sequence diagrams also adhere to the abstractions used in the class diagram.

Each sequence diagram pertains to one of the possible interactions between the user and MyUI. The combination of all use cases covers the main functionalities of MyUI. Note that in this section, the assumption has been made that the user is logged into MyUI as an admin unless specified otherwise.

User login

In order to log in, the user must provide his/her credentials. Clicking the submit button will trigger MyUI to check said credentials. Depending on their validity, the user will either be redirected to the homepage of MyUI, or an error message is displayed.

Goal: A user wants to log in to MyUI in order to use the application.
Preconditions: The user is not logged in to MyUI.
Postcondition: The user is logged in to MyUI, in accordance with the user permission profile.

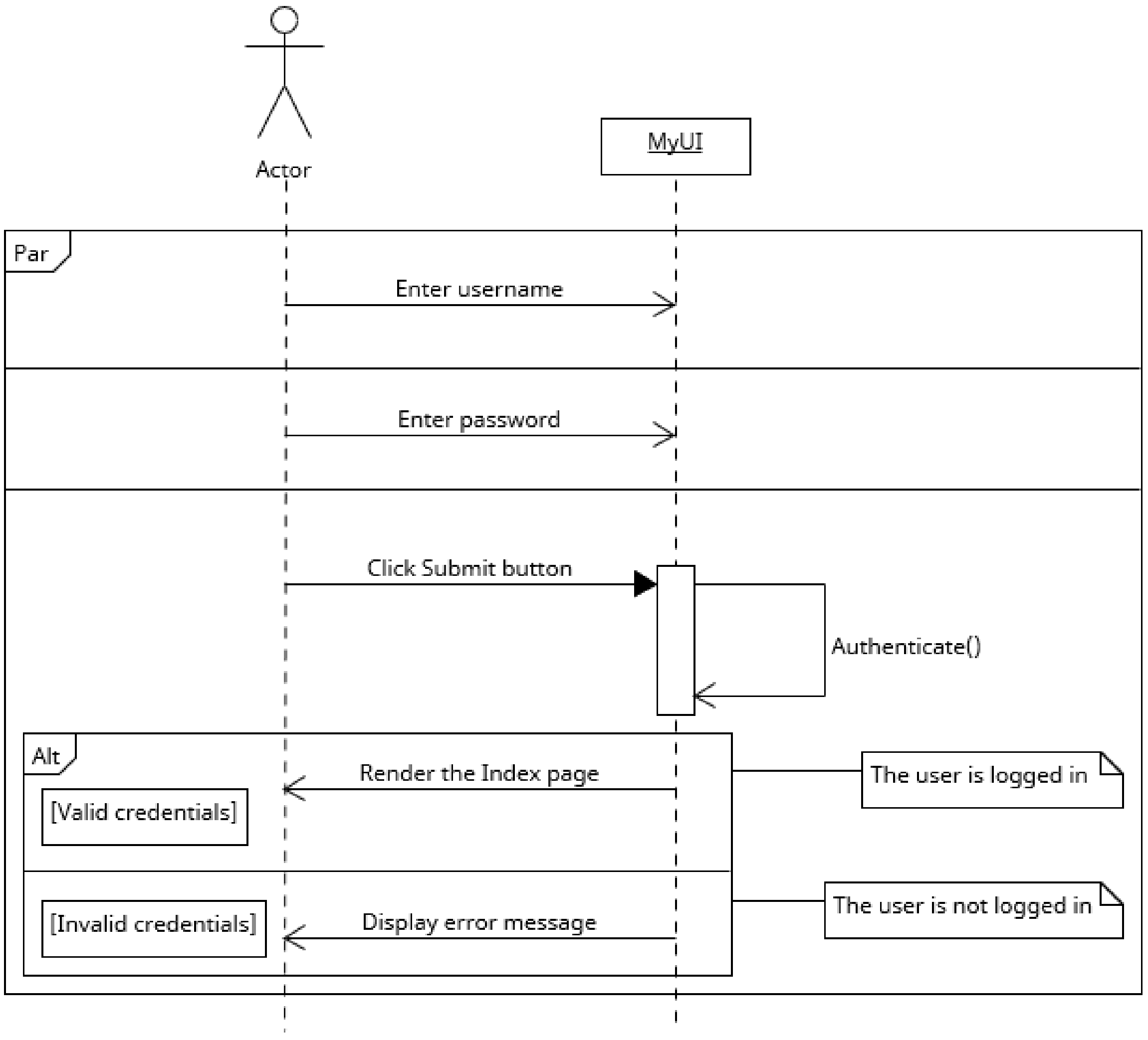


Figure 3.9: User login UML sequence diagram

Viewing user menu

In order to view the user menu, a user can click on the user menu icon in the header of MyUI. The user menu will then be unfolded.

Goal: A user wants to open the user menu.
Preconditions: The user menu is folded.
Postcondition: The user menu is unfolded.

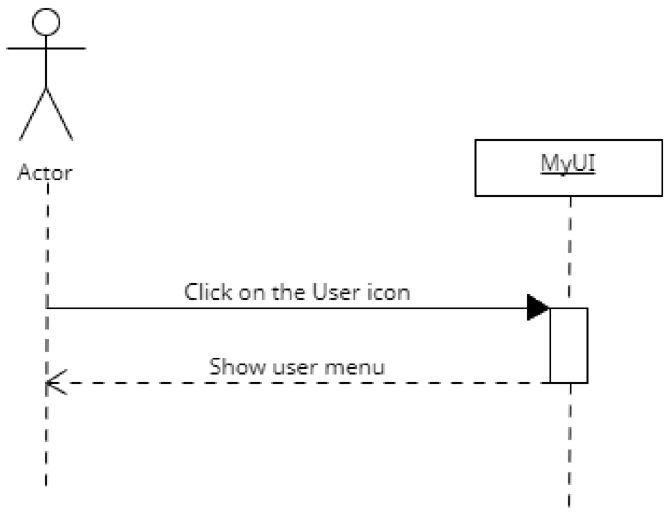


Figure 3.10: Open user menu UML sequence diagram

User sign out

In order to Sign out, the user can click the sign out button in the user menu. The login screen will then be shown.

Goal: A user wants to log out of MyUI.
Preconditions: The user menu is unfolded.
Postcondition: The user is logged out of MyUI.

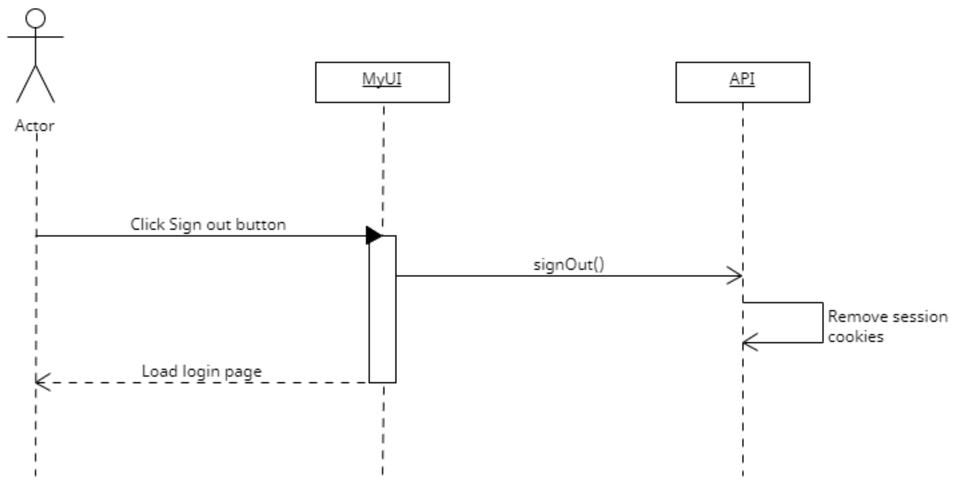


Figure 3.11: User logout UML sequence diagram

Change application language

In order to change the language of MyUI, the user clicks on the button representing the language he/she wants to switch to. MyUI will then update the locale and redirect the user to the home page. Below are two sequence diagrams, one for changing the language to Dutch, and one for changing the language to English.

Goal: A user wants to change the language of MyUI.
Precondition: The user menu is unfolded.
Postcondition: MyUI is displayed in the selected language.

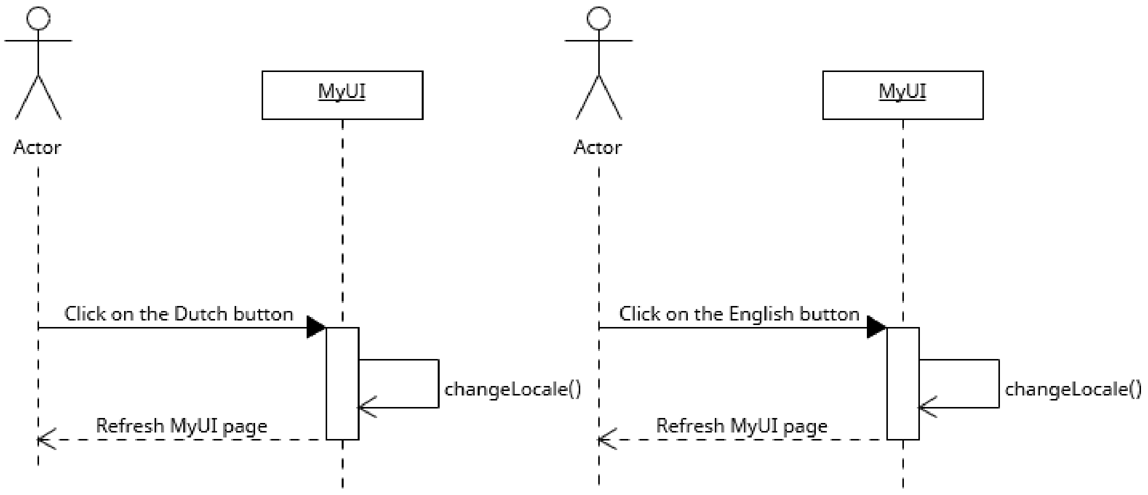


Figure 3.12: Change language UML sequence diagram

Unfold dashboard dropdown menu

In order to see the list of existing dashboards, a user can click the dashboard menu button. The dashboard menu will then be unfolded.

Goal: A user wants to see the list of existing dashboards.
Preconditions: The dashboard dropdown menu is folded.
Postcondition: The dashboard dropdown menu is unfolded.

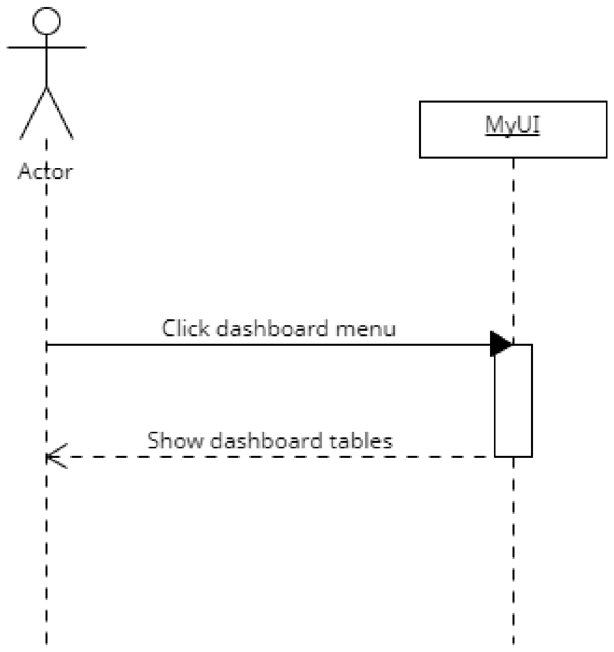


Figure 3.13: Unfolding dashboard dropdown menu UML sequence diagram

Fold dashboard dropdown menu

In order to hide the list of existing dashboards, when it is shown, a user can click the dashboard menu button. The dashboard dropdown menu will then be folded.

Goal: A user wants to fold the list of existing dashboards.
Preconditions: The dashboard dropdown menu is unfolded.
Postcondition: The dashboard dropdown menu is folded.

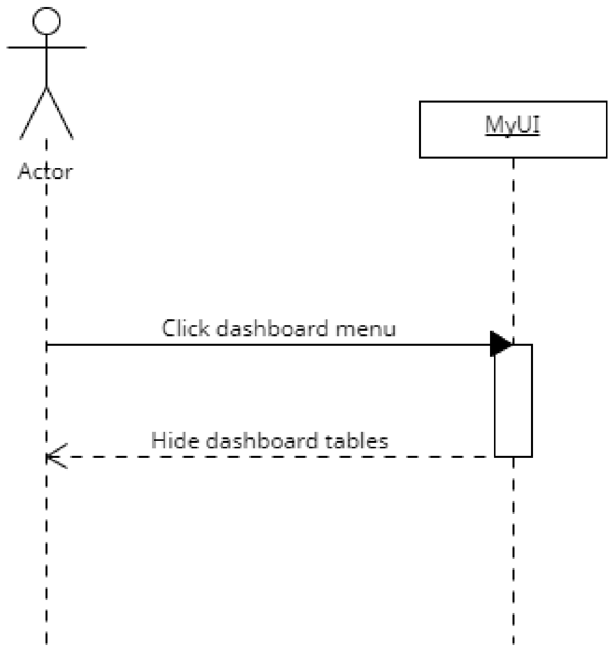


Figure 3.14: Folding dashboard dropdown menu UML diagram

Unfold base table dropdown menu

In order to see the list of base tables the user has access to a user can click the dashboard menu button. The dashboard menu will then be unfolded.

Goal: A user wants to see the list of existing base tables.
Preconditions: The base table dropdown menu is folded.
Postcondition: The base table dropdown menu is unfolded.

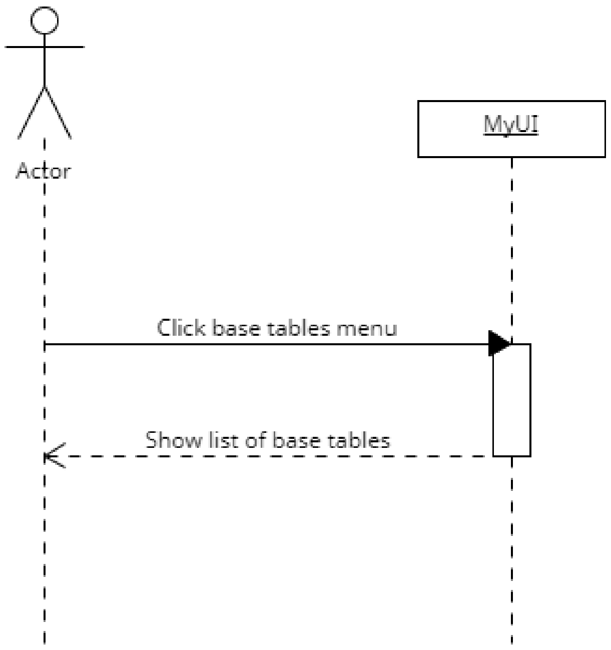


Figure 3.15: Unfolding base table dropdown menu

Fold base table dropdown menu

In order to hide the list of accessible base tables, when it is shown, a user can click the dashboard menu button. The base table dropdown menu will then be folded.

Goal: A user wants to fold the list of existing base tables.
Preconditions: The base table dropdown menu is unfolded.
Postcondition: The base table dropdown menu is folded.

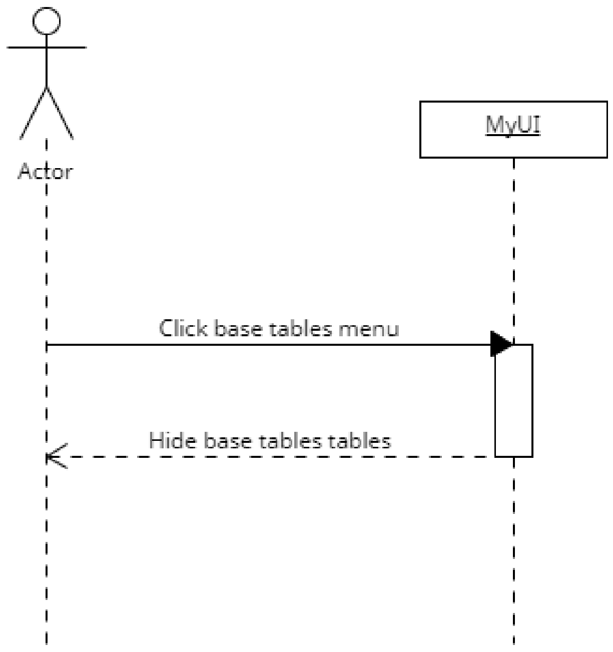


Figure 3.16: Folding base table dropdown menu

Select base table

In order to view a base table, a user can click on the name in the base table dropdown menu corresponding to the table they wish to view. MyUI then fetches the column names, based on the user permission profile. If a timeout error does not occur, the base table is displayed to the user.

Goal: A user wants to view a base table.
Preconditions:

- The selected base table is not shown in the workspace.
- The base table dropdown menu is unfolded.

Postcondition: The selected base table is shown in the workspace.

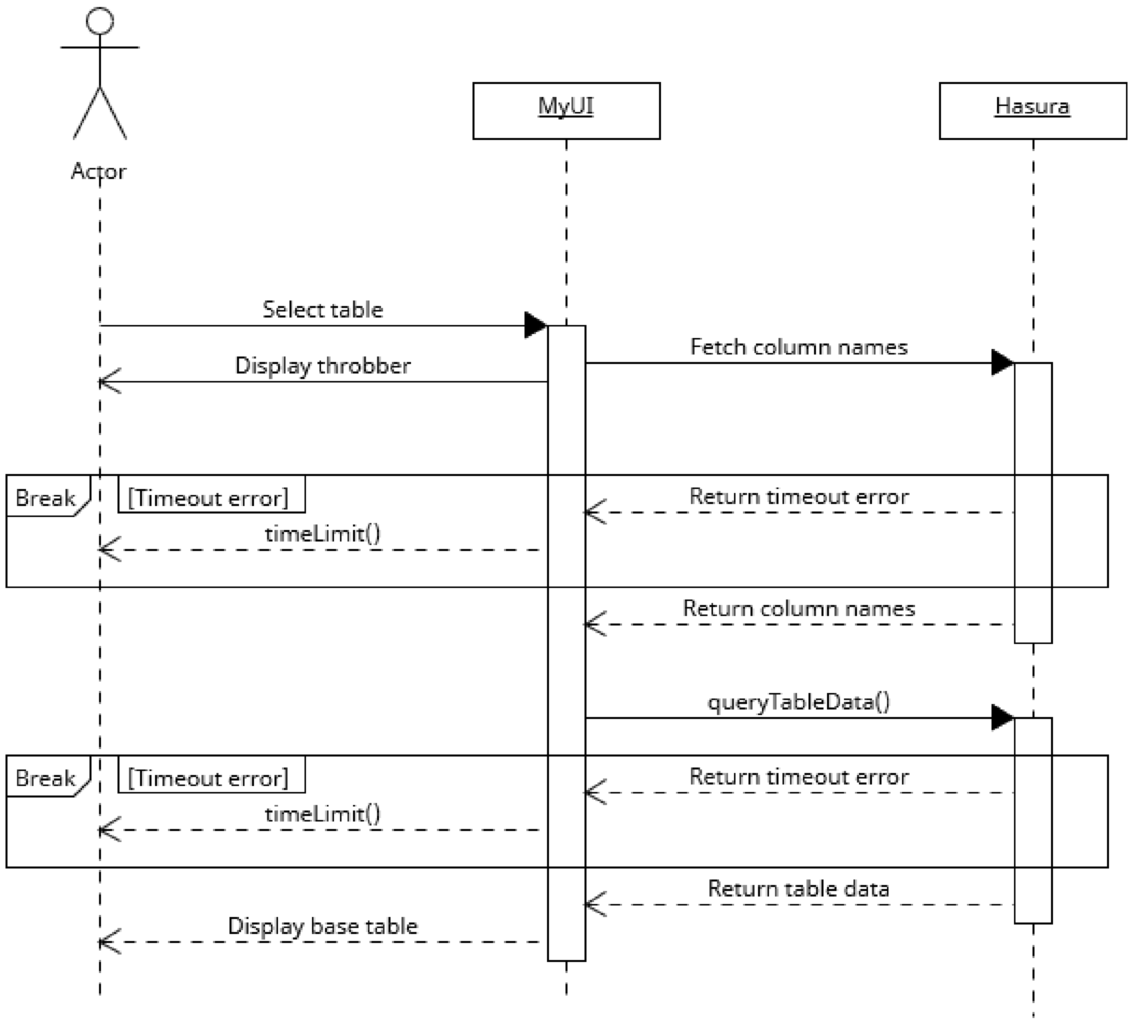


Figure 3.17: Selecting a base table UML sequence diagram

Select a row in a table

In order to select a row, the user can click the checkbox corresponding to the row. The row will then be selected.

Goal: A user wants to select a row in a table.
Preconditions: The desired table is shown in the workspace.
Postcondition: The selected row is highlighted.

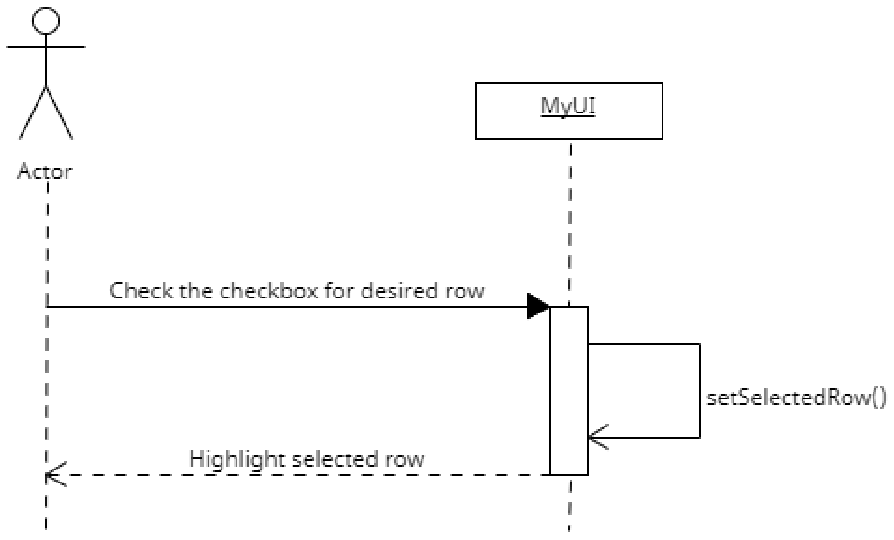


Figure 3.18: Selecting a row UML sequence diagram

Deselect a row in a table

In order to deselect a row, the user can click the checkbox corresponding to the row. The row will then be deselected.

Goal: A user wants to deselect a row in a table.
Preconditions:

- The desired table is shown in the workspace.
- A row has been selected.

Postcondition: The previously selected row is deselected.

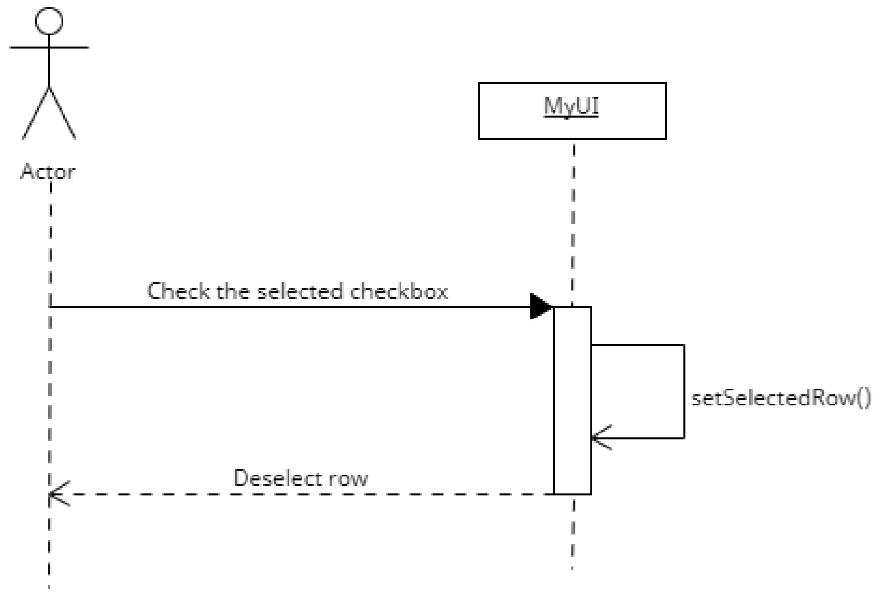


Figure 3.19: Deselecting a row UML sequence diagram

Select all rows on a page

The user can select all the rows on the current page of a table by checking the checkbox in the table header. After this checkbox has been selected, all the rows of the table on that page are highlighted.

Goal: A user wants to select all rows on the current page of a table.
Preconditions:

- The desired table is shown in the workspace.
- All the rows on the current page of the table are not selected.

Postcondition: All rows are selected and highlighted.

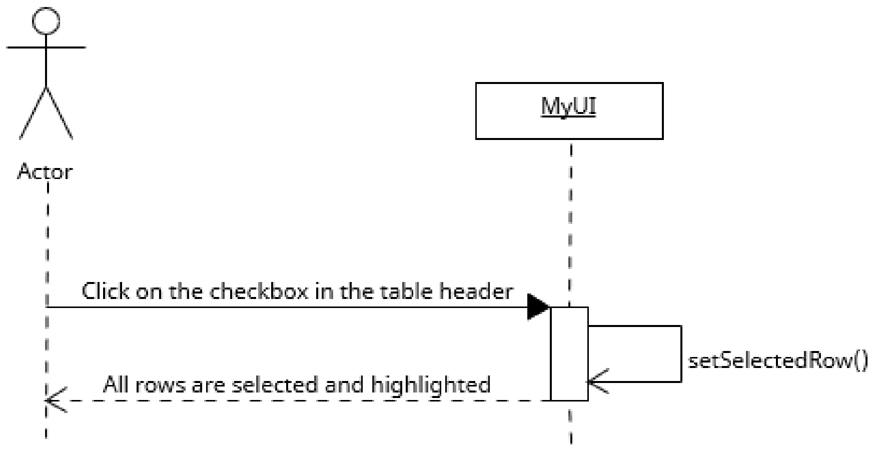

Actor
MyUI
Click on the checkbox in the table header
setSelectedRow()
All rows are selected and highlighted

Figure 3.20: Selecting all rows UML sequence diagram

Deselect all rows on a page

In order to deselect all the rows on a page, a user clicks on the checkbox in the table header. Once deselected, the rows are not highlighted anymore.

Goal: A user wants to deselect all rows on the current page of a table.
Preconditions:

- The desired table is shown in the workspace.
- All the rows on the current page of the table are selected.
- The checkbox in the table header is selected.

Postcondition: All rows are deselected.

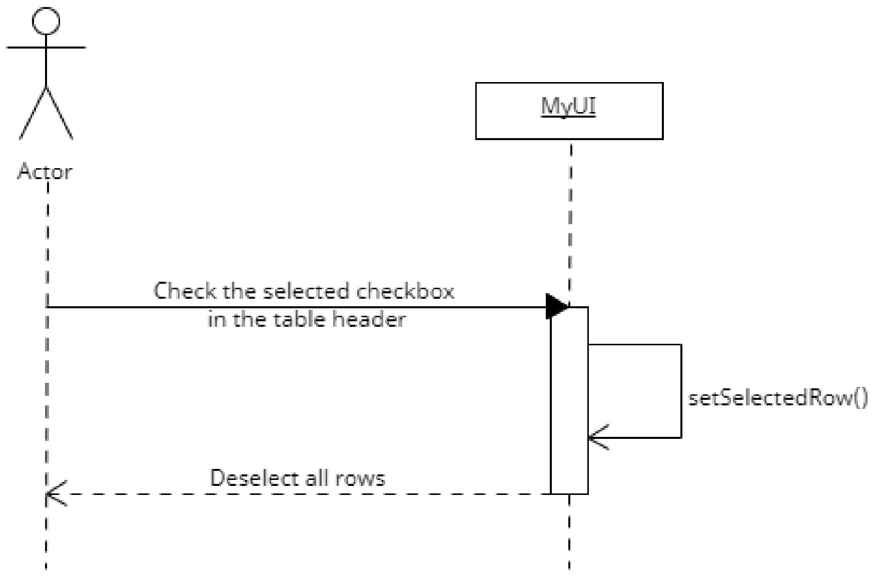


Figure 3.21: Deselecting all rows UML sequence diagram

Insert a row

In order to add a row to a pure table, a user clicks on the Add Row button which opens the add row menu. In this menu, the user can choose to either cancel the addition or fill in all the required attributes. When a user clicks on the OK button, MyUI shall check the validity of the input, and display an error message or add the row to the pure table accordingly.

Goal: A user wants to add a row to a pure table.
Preconditions:

- The user is viewing a pure table.
- The user is not in edit mode.

Postcondition: A row is added to the pure table.

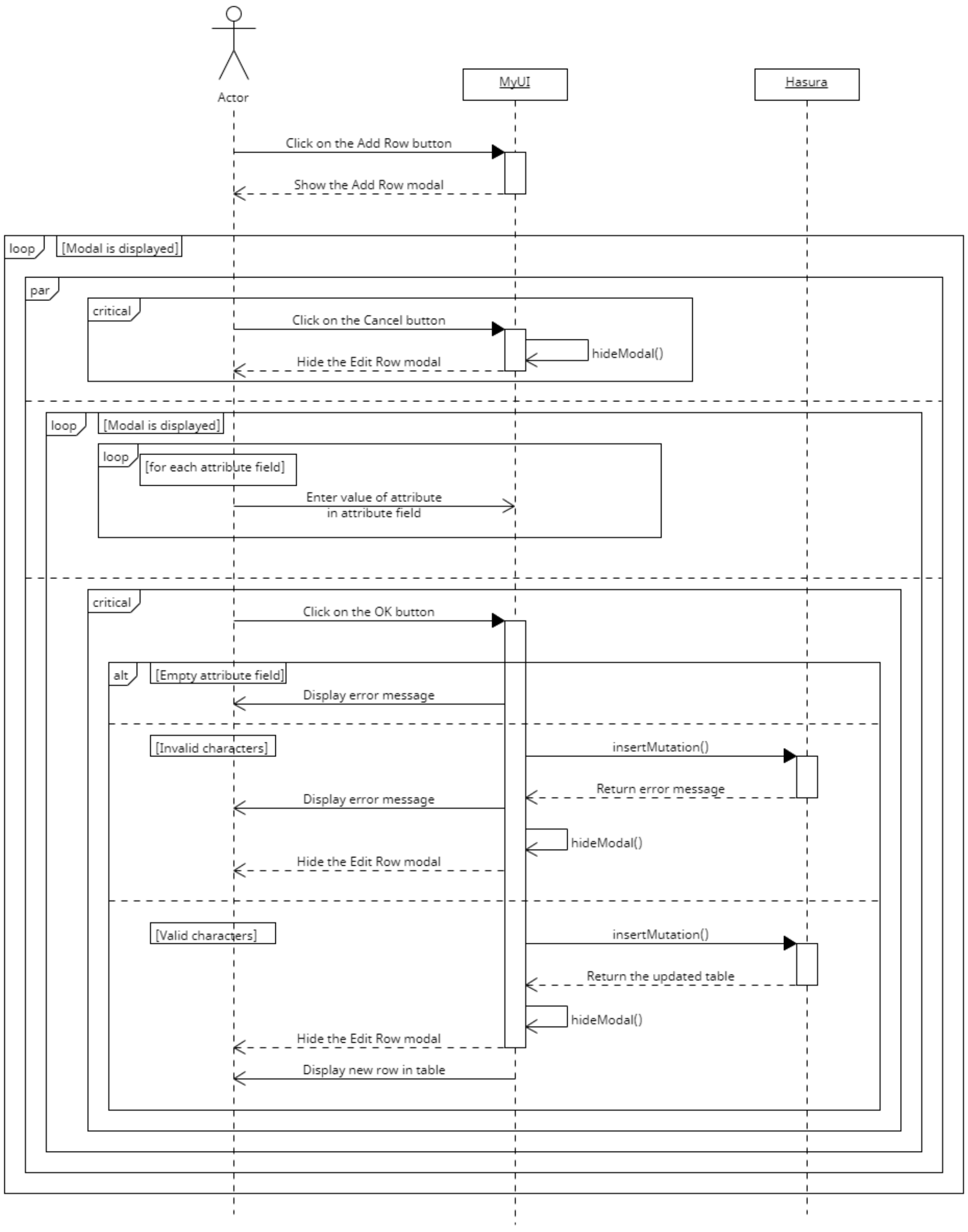


Figure 3.22: Insert a row UML sequence diagram

Delete a row

In order to delete a row from a pure table, a user can click on the delete row button. The selected rows will be deleted.

Goal: A user wants to delete a row from a pure table.
Preconditions:

- The user is viewing a pure table.
- The user is not in edit mode.
- At least one row in the pure table has been selected.

Postcondition: A row is deleted from the pure table.

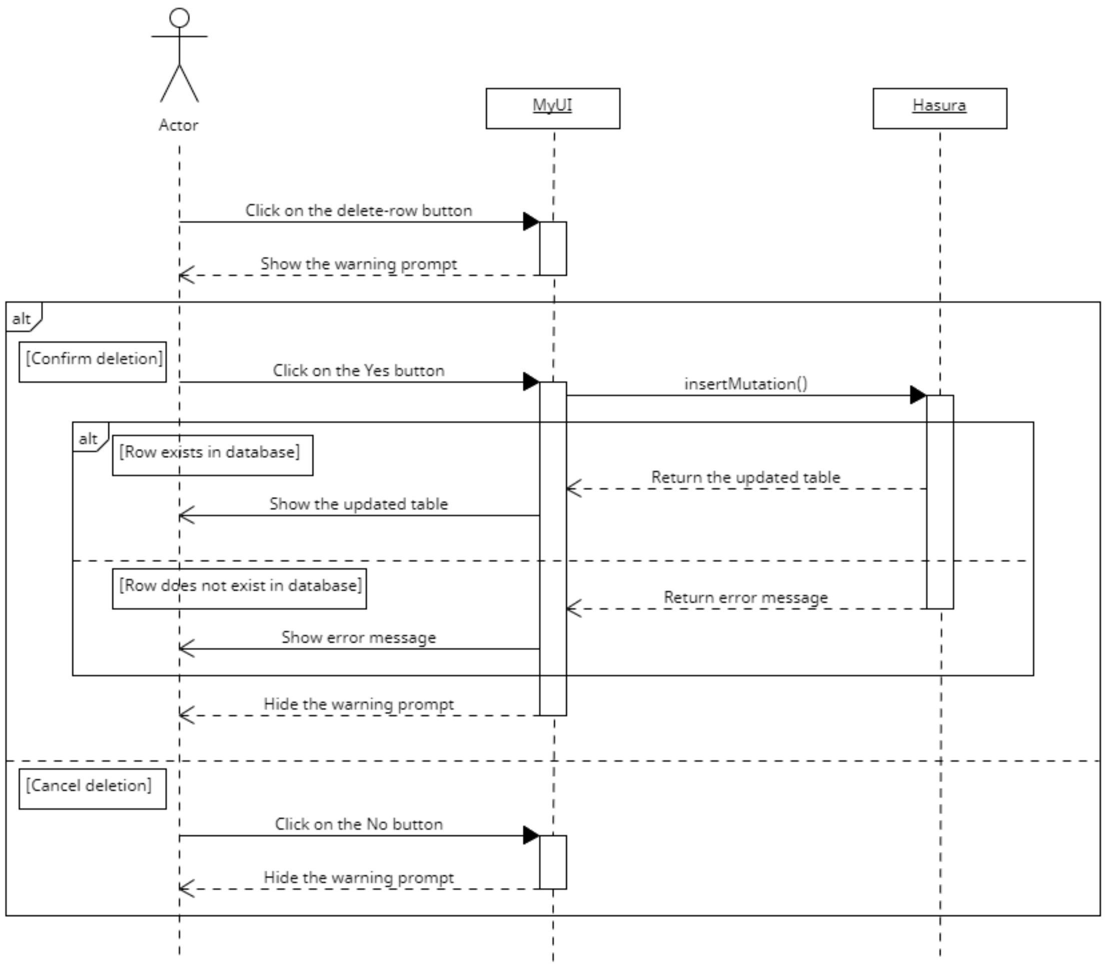


Figure 3.23: Delete a row UML sequence diagram

Edit a row

In order to edit a row in a pure table, a user can click on the Edit Row button. Depending on the user permission profile, they are able to edit the attributes of said row. MyUI will also perform a check for empty fields and invalid characters. When the attribute values are valid, the row will be updated in the table.

Goal: A user wants to edit a row in a pure table.
Preconditions:

- The user is viewing a pure table.
- The user is not in edit mode.

Postcondition: A row in a pure table has been edited and updated.

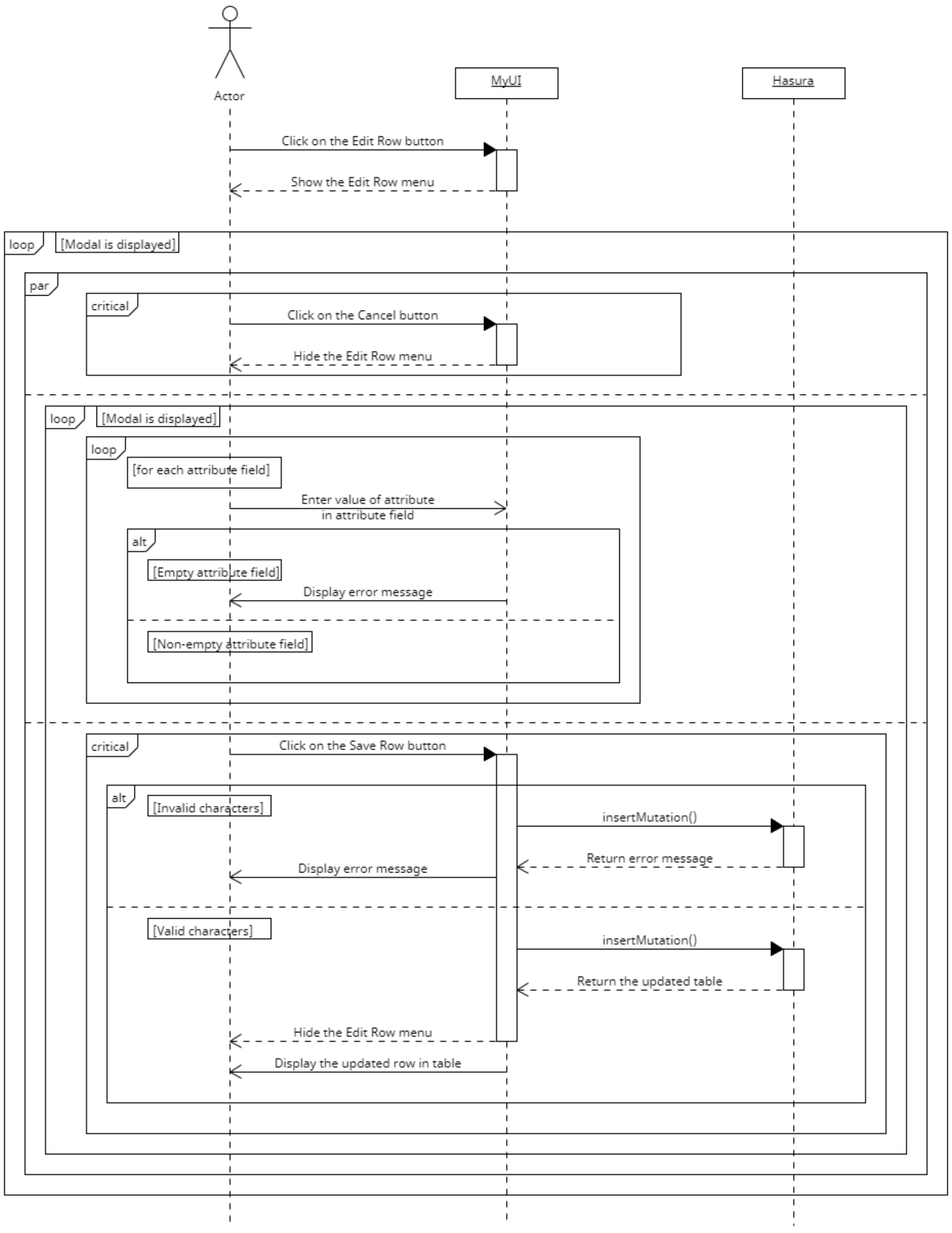


Figure 3.24: Edit a row UML sequence diagram

Ordering rows

In order to change the ordering of the table for a specific column, the user can click on the header of the column. The table will then be ordered ascending on the selected column if the table was not ordered on that column. If the table was ordered ascending on the selected column, the table will be ordered descending on the column. If the table was ordered descending on the selected column the default ordering will be shown.

Sorting a grid view in ascending order

Goal: A user wants to order the rows of the selected grid view in ascending order.
Preconditions: The selected grid view is shown in the workspace.
Postcondition: The rows of the selected grid view are sorted in ascending order.

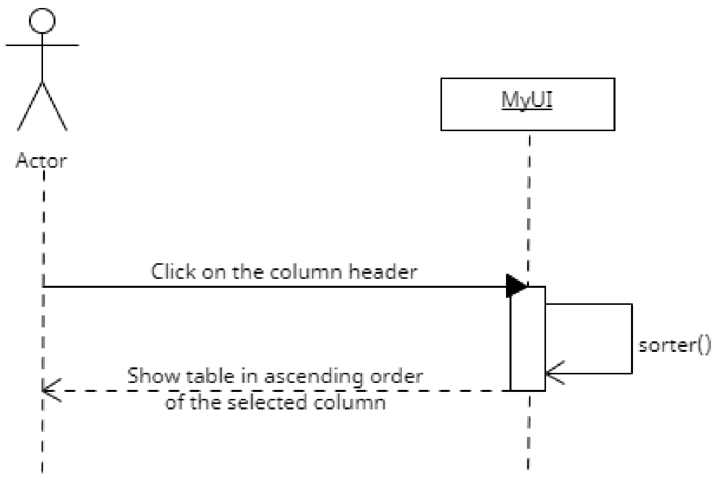

Actor
MyUI
Click on the column header
sorter()
Show table in ascending order of the selected column

Figure 3.25: Sorting the rows of a base table in ascending order UML sequence diagram

Sorting a grid view in descending order

Goal: A user wants to order the rows of the selected grid view in descending order.
Preconditions:

- The selected grid view is shown in the workspace.
- The grid view is ordered in ascending order on the column.

Postcondition: The rows of the selected grid view are sorted in descending order.

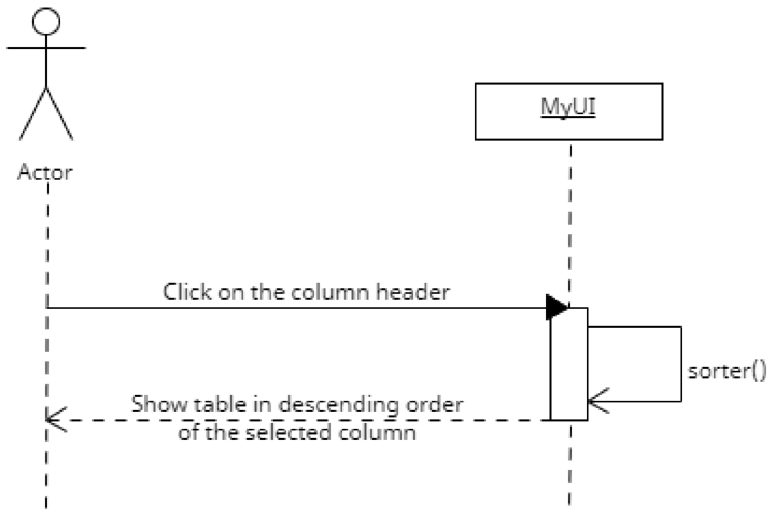

Actor
MyUI
Click on the column header
sorter()
Show table in descending order of the selected column

Figure 3.26: Sorting the rows of a base table in descending order UML sequence diagram

Sorting a grid view in default order

Goal: A user wants to restore the default order of the rows of the selected grid view.
Preconditions:

- The selected grid view is shown in the workspace.
- The grid view is ordered in descending order on the column.

Postcondition: The rows are sorted on the default ordering of the grid view.

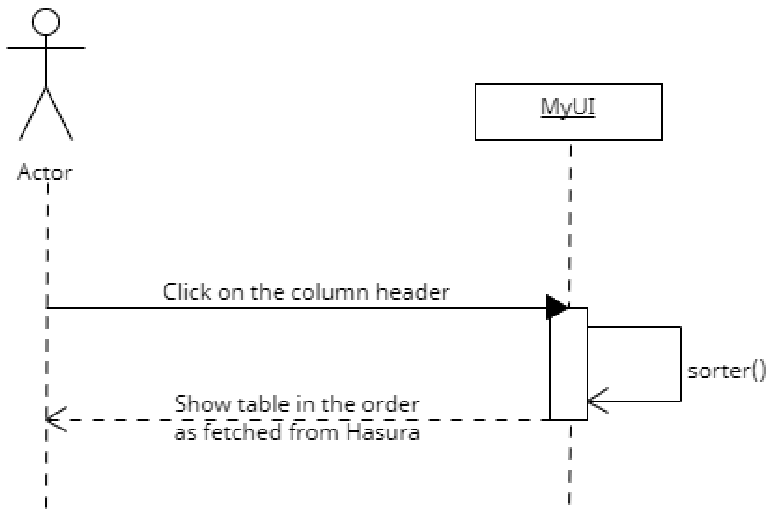

Actor
MyUI
Click on the column header
sorter()
Show table in the order as fetched from Hasura

Figure 3.27: Restore the ordering of the rows of a table UML sequence diagram

Select pagination setting - Select page of a grid view

In order to change the page of the table that is shown, the user can press the previous or next page button. The previous or next page will be shown respectively.

Goal: A user can navigate through multiple pages of a grid view.
Preconditions: The user is viewing a grid view.
Postcondition: The user is viewing a different page.

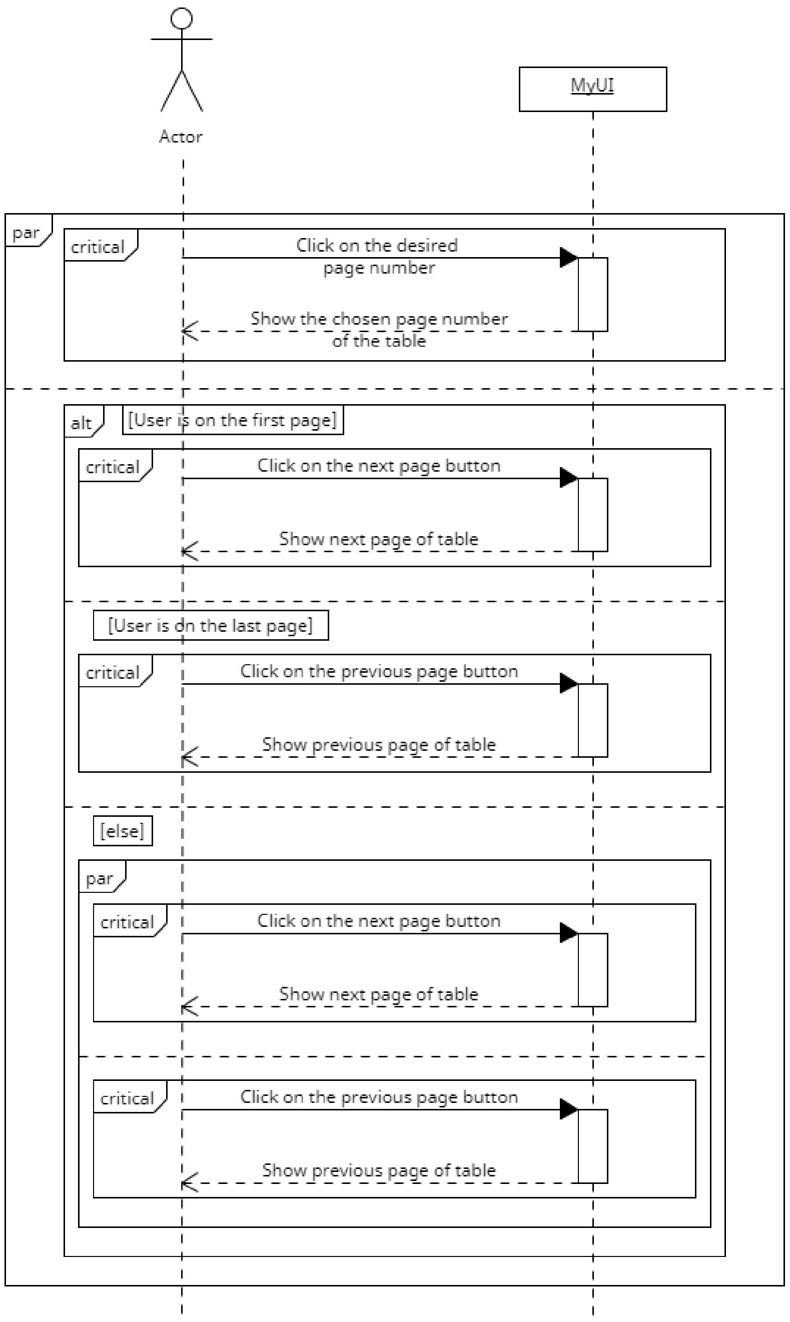

Actor
MyUI
par
critical
Click on the desired page number
Show the chosen page number of the table
alt
[User is on the first page]
critical
Click on the next page button
Show next page of table
[User is on the last page]
critical
Click on the previous page button
Show previous page of table
[else]
par
critical
Click on the next page button
Show next page of table
critical
Click on the previous page button
Show previous page of table

Figure 3.28: Select Pagination Setting UML sequence diagram

Select pagination setting - Change number of rows displayed per page

In order to change the number of rows being shown on one page of a table, the user can click one of the rows per page buttons in the rows per page dropdown menu. The table will show the number of rows per page respective to the selected option.

Goal: A user selects the pagination setting of a grid view.
Preconditions: The user is viewing a grid view.
Postcondition: The user has changed the pagination setting of the table.

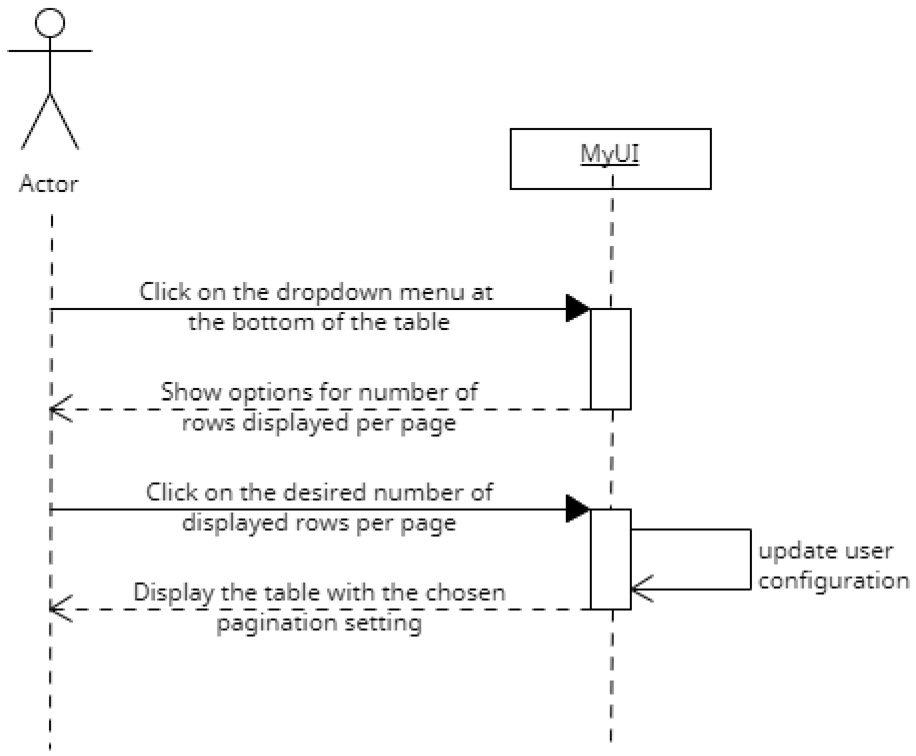

Actor
MyUI
Click on the dropdown menu at the bottom of the table
Show options for number of rows displayed per page
Click on the desired number of displayed rows per page
update user configuration
Display the table with the chosen pagination setting

Figure 3.29: Select Pagination Setting UML sequence diagram

Add a dashboard

In order to add a new dashboard, the user can click on the add new dashboard button. A prompt will be shown where the user can enter a unique name for the dashboard and confirm this name. The dashboard will then be added to the dashboard menu and it will be selected.

Goal: A user wants to add a new dashboard.
Preconditions: The dashboard dropdown menu is unfolded.
Postcondition: The new dashboard is present in the dropdown dashboard menu and the user is shown the new dashboard.

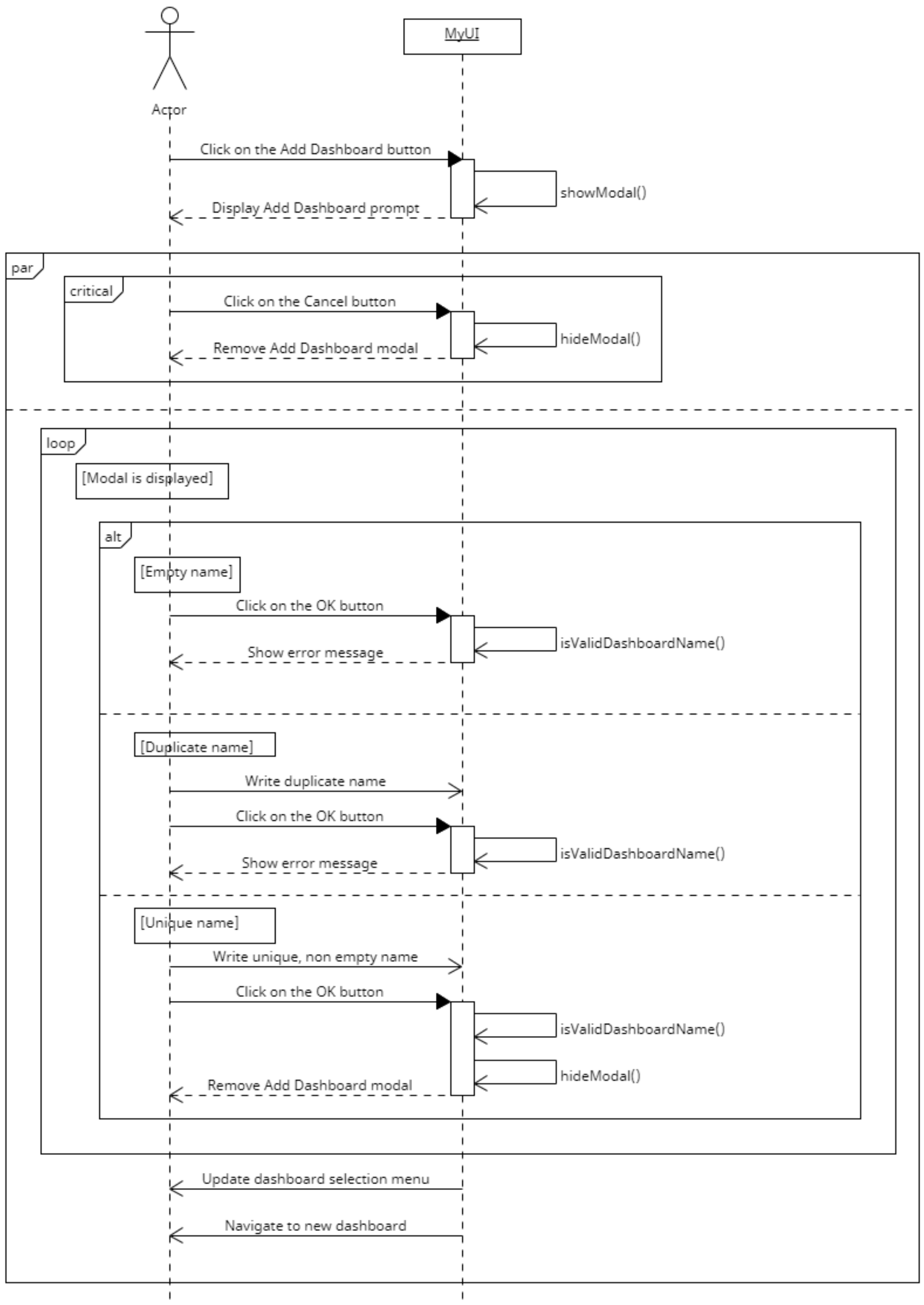


Figure 3.30: Adding a dashboard UML sequence diagram

Delete a dashboard

A dashboard can be deleted by clicking on the Delete Dashboard button. A user can then enter the name of the dashboard to be deleted. The validity of the dashboard name will be checked, and if the corresponding dashboard does not exist, an error message will be displayed.

Goal: A user wants to delete a specific dashboard.
Preconditions: The dashboard dropdown menu is unfolded.
Postcondition: The selected dashboard is deleted and no longer visible in the dropdown dashboard menu.

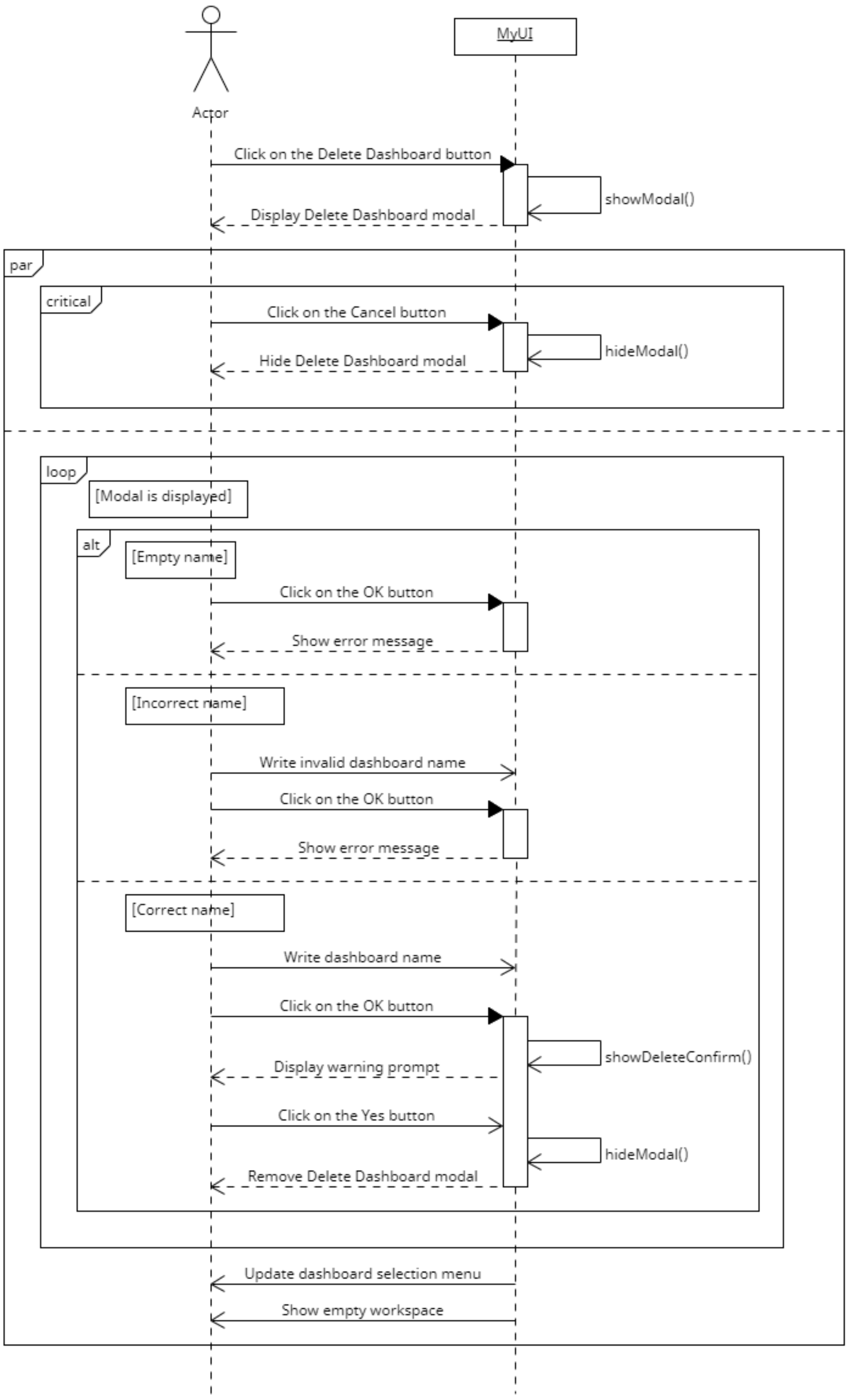


Figure 3.31: Deleting a dashboard UML sequence diagram

Select a dashboard

A user can select a dashboard by clicking on the name of the dashboard in the dropdown menu. The dashboard and its application elements will be shown in the workspace.

Goal: A user wants to select a specific dashboard.
Preconditions:

- The dashboard dropdown menu is unfolded.
- The selected dashboard is not shown in the workspace.

Postcondition: The selected dashboard is shown in the workspace.

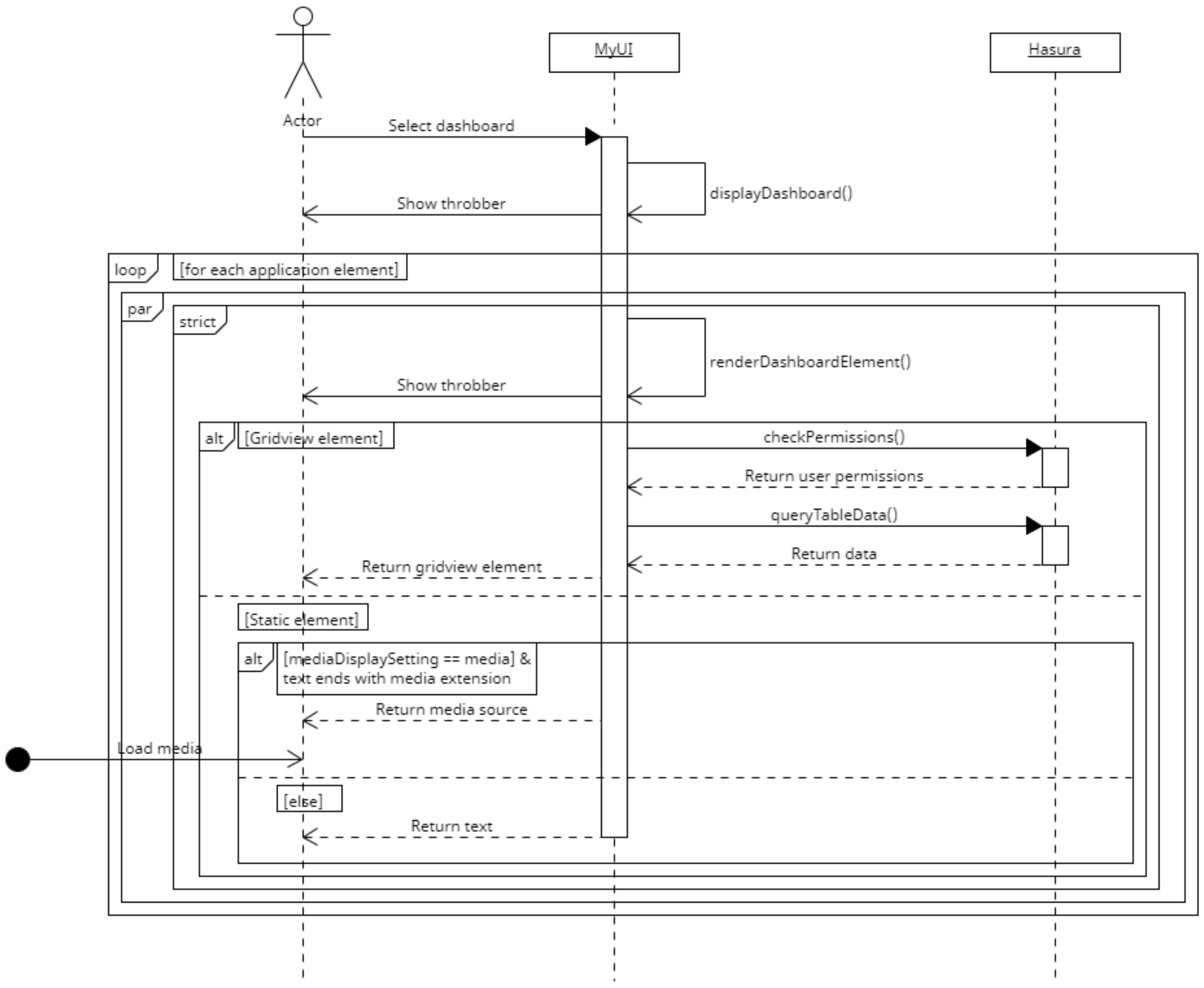


Figure 3.32: Selecting a dashboard UML sequence diagram

Enter edit mode

The user can click on the gear in the header of the application to enter edit mode. To signify that the user has entered edit mode, the gear that was clicked on in the header starts spinning.

Goal: A user enters edit mode.
Preconditions:

- The user has a dashboard open.
- The user is not in edit mode on the dashboard.

Postcondition: The user has a dashboard open in edit mode.

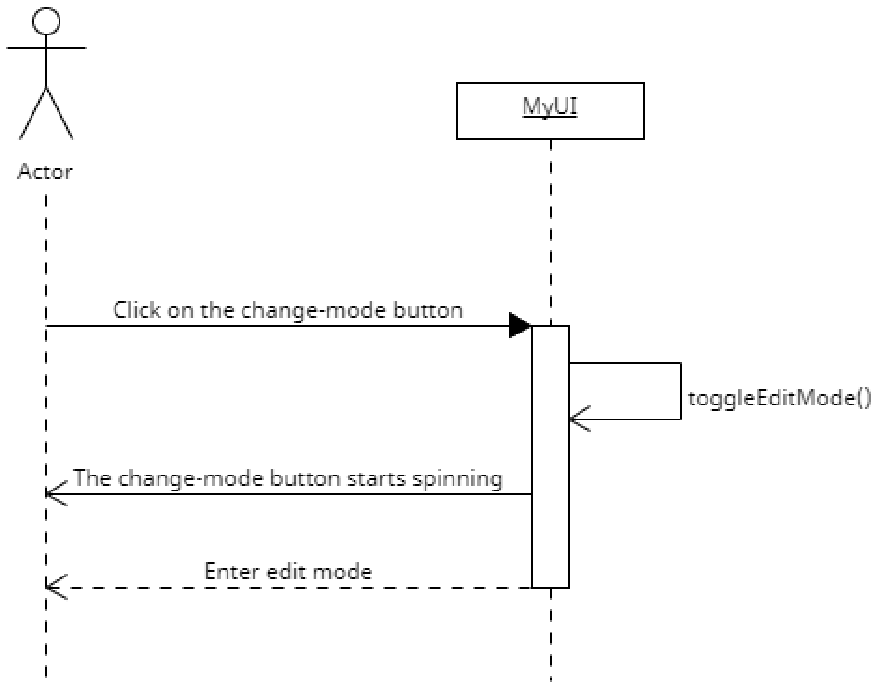

Actor
MyUI
Click on the change-mode button
toggleEditMode()
The change-mode button starts spinning
Enter edit mode

Figure 3.33: Entering edit mode UML sequence diagram

Exit edit mode

A user can click on the spinning gear in the header of the application to exit edit mode. If the user has not saved changes to the dashboard, they are given an option to save their dashboard or to exit edit mode without saving the changes. To signify that the user has exited edit mode, the gear that was clicked on stops spinning.

Goal: A user exits edit mode.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.

Postcondition: The user has exited edit mode.

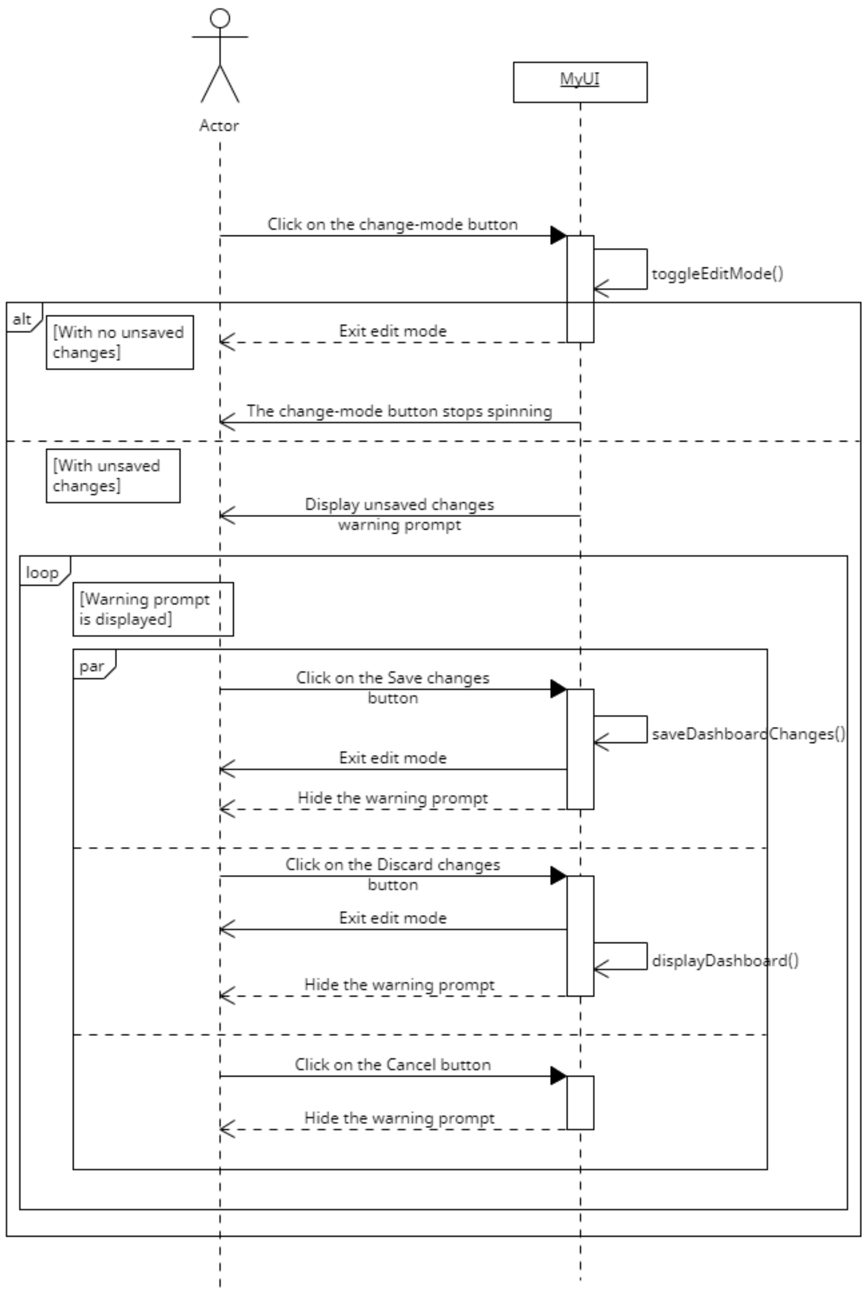

Actor
MyUI
Click on the change-mode button
toggleEditMode()
alt
[With no unsaved changes]
Exit edit mode
The change-mode button stops spinning
[With unsaved changes]
Display unsaved changes warning prompt
loop
[Warning prompt is displayed]
par
Click on the Save changes button
saveDashboardChanges()
Exit edit mode
Hide the warning prompt
Click on the Discard changes button
Exit edit mode
displayDashboard()
Hide the warning prompt
Click on the Cancel button
Hide the warning prompt

Figure 3.34: Exit edit mode UML sequence diagram

Adding a static element

A user can add a static element to the dashboard workspace by dragging the static element from the sidebar to the workspace. Once dropped, the edit menu of the added element will be shown.

Goal: Adding a static element.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.

Postcondition: A static element is added to the dashboard.

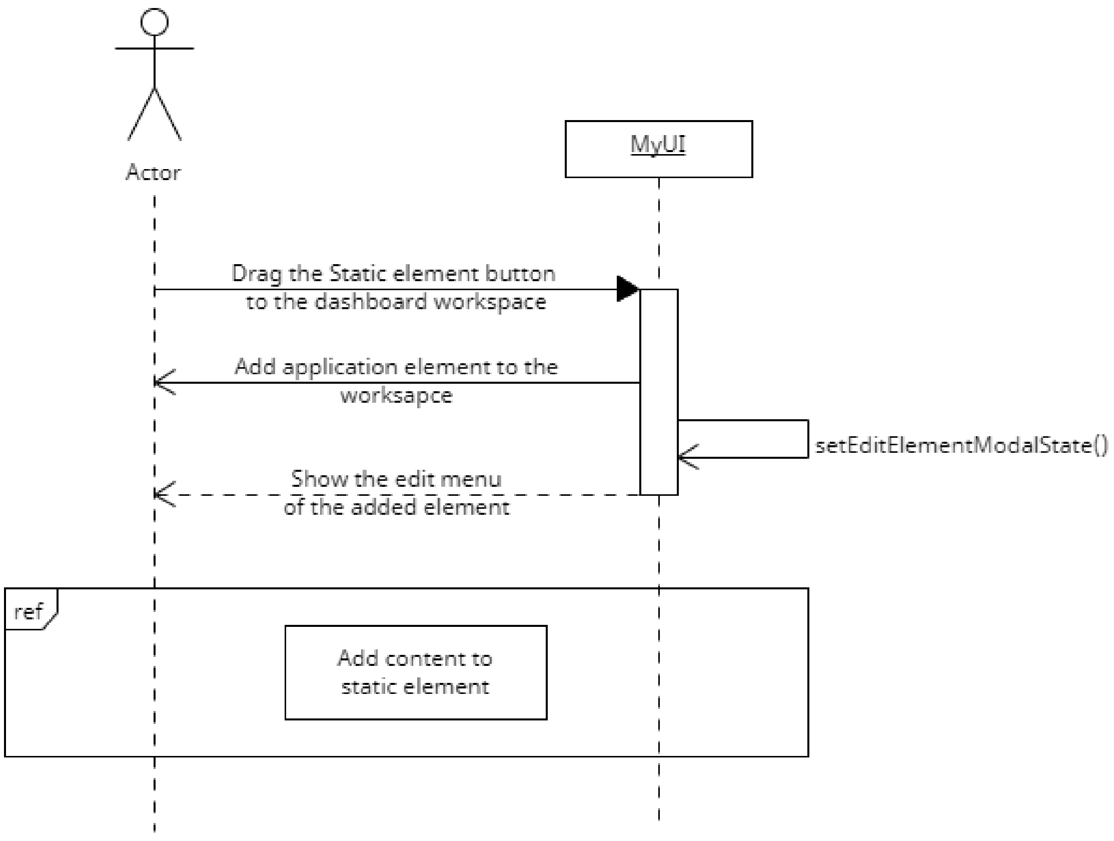

Actor
MyUI
Drag the Static element button to the dashboard workspace
Add application element to the worksapce
setEditElementModalState()
Show the edit menu of the added element
ref
Add content to static element

Figure 3.35: Adding a static element UML sequence diagram

Adding content to a static element

While entering text in the edit menu of a static element, MyUI will check the format of the text. MyUI will display an error message when the text contains invalid characters. After submitting, the entered content is loaded on the dashboard.

Goal: A user adds content to a static element.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.
- The user has added a static element to the workspace.
- The edit element menu of the static element is open.

Postcondition: The entered content is loaded on the dashboard.

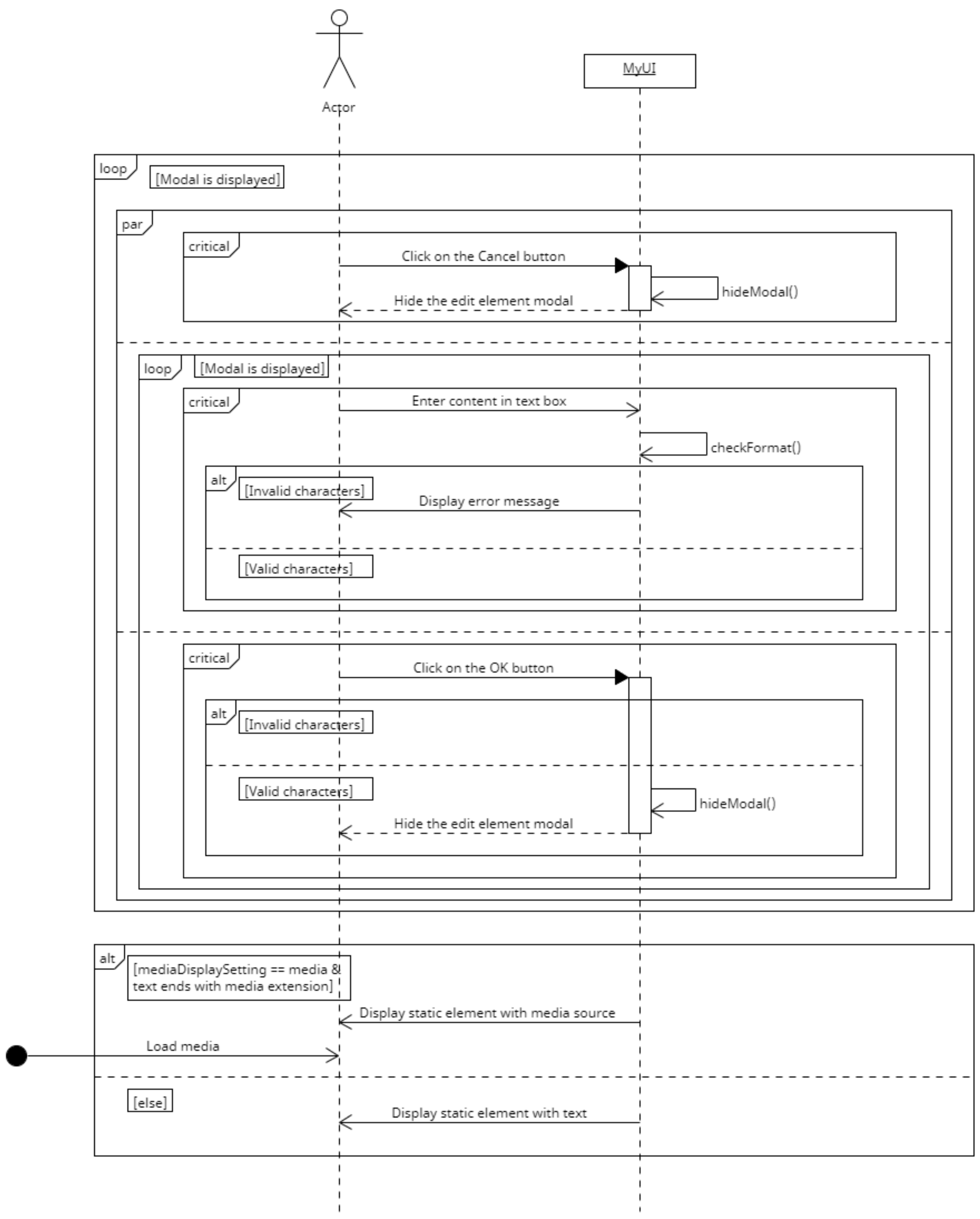


Figure 3.36: Adding content to a static element UML sequence diagram

Adding a gridview element

A user can add a grid view element to the dashboard workspace by dragging the grid view element from the sidebar to the workspace. Once dropped, the edit menu of the added element will be shown.

Goal: Adding a gridview element.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.

Postcondition: A grid view element is added to the dashboard.

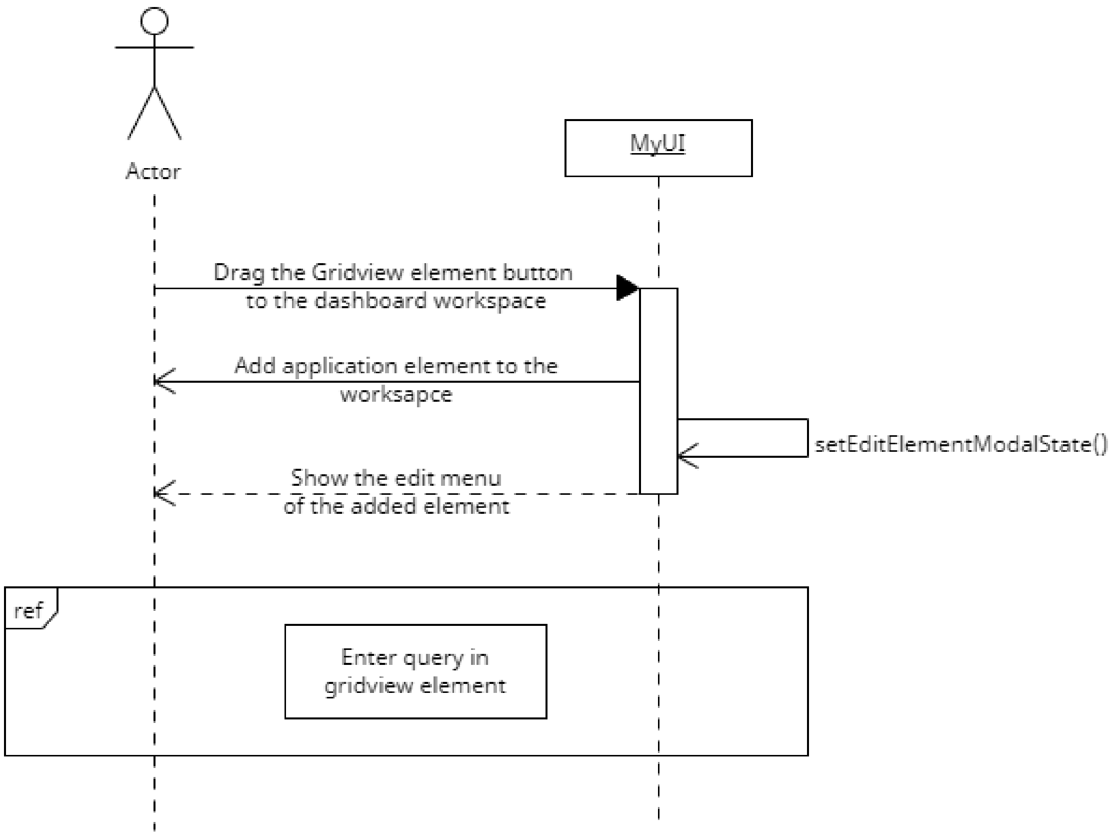

Actor
MyUI
Drag the Gridview element button to the dashboard workspace
Add application element to the worksapce
setEditElementModalState()
Show the edit menu of the added element
ref
Enter query in gridview element

Figure 3.37: Adding a grid view element UML sequence diagram

Entering a query in a grid view element

While entering a query in the edit menu of a grid view, MyUI will check the syntax of the query. MyUI will display an error message when the syntax is incorrect. When the user enters a correct query in a grid view element, the data from the query is fetched from the base table and loaded in the element on the workspace. The edit element menu closes once the query has been entered.

Goal: A user enters a query in a grid view element.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.
- The user has added a grid view element to the workspace.
- The edit element menu of the grid view element is open.

Postcondition: The data from the query is loaded on the dashboard.

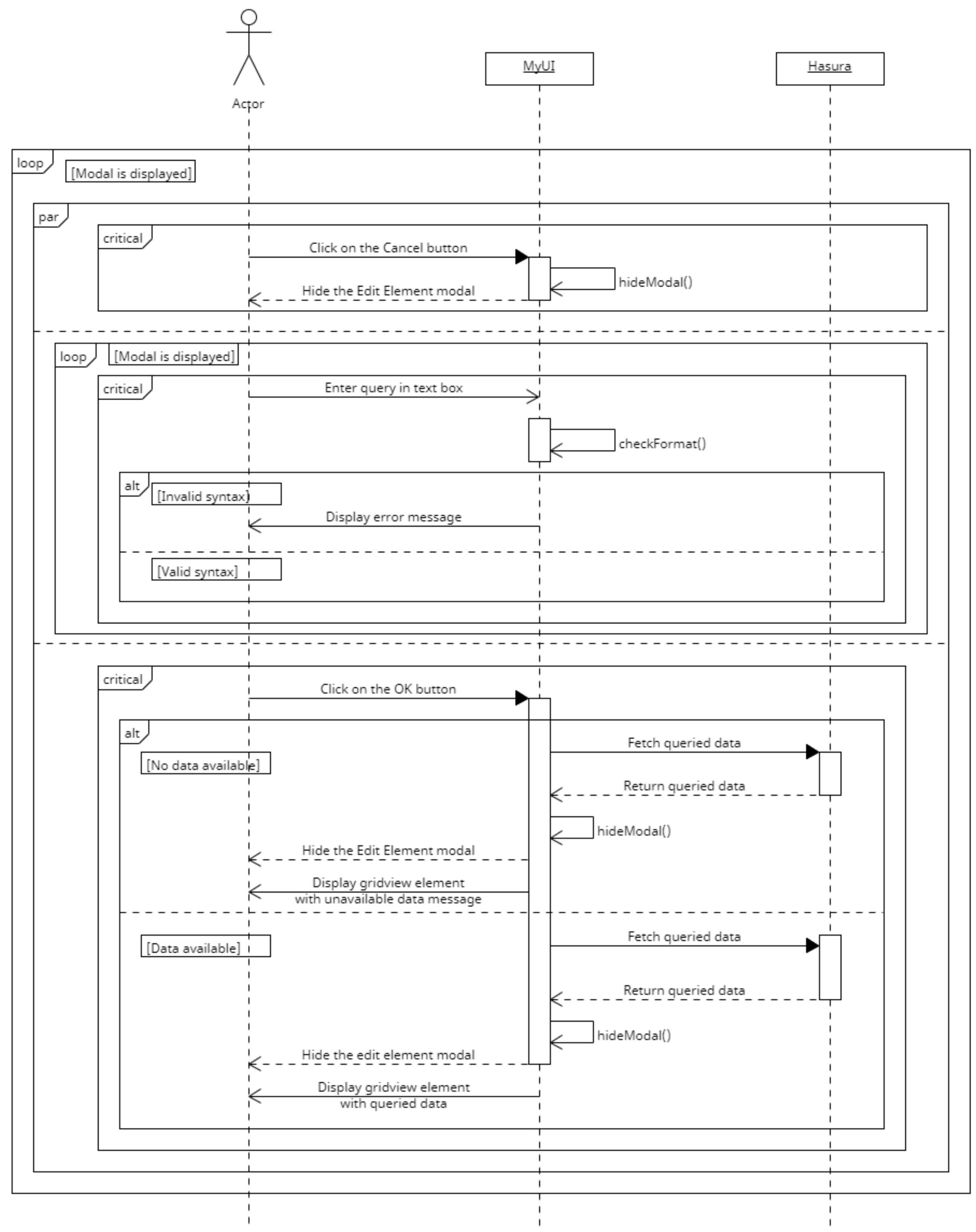


Figure 3.38: Entering a query in a grid view element UML sequence diagram

Editing an application element

Users can edit an application element by double-clicking on it. This action from the user opens an edit menu with a text field that the user can then edit. When a user edits a static element, either text or media links can be added to the text field of the edit menu, as seen in Figure 3.39. When a user edits a grid view element, a GraphQL query of correct syntax must be entered in the text field of the edit menu, as seen in Figure 3.40.

Goal: An application element is edited.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.
- The dashboard has an application element.

Postcondition: The edit menu of the dashboard is shown.

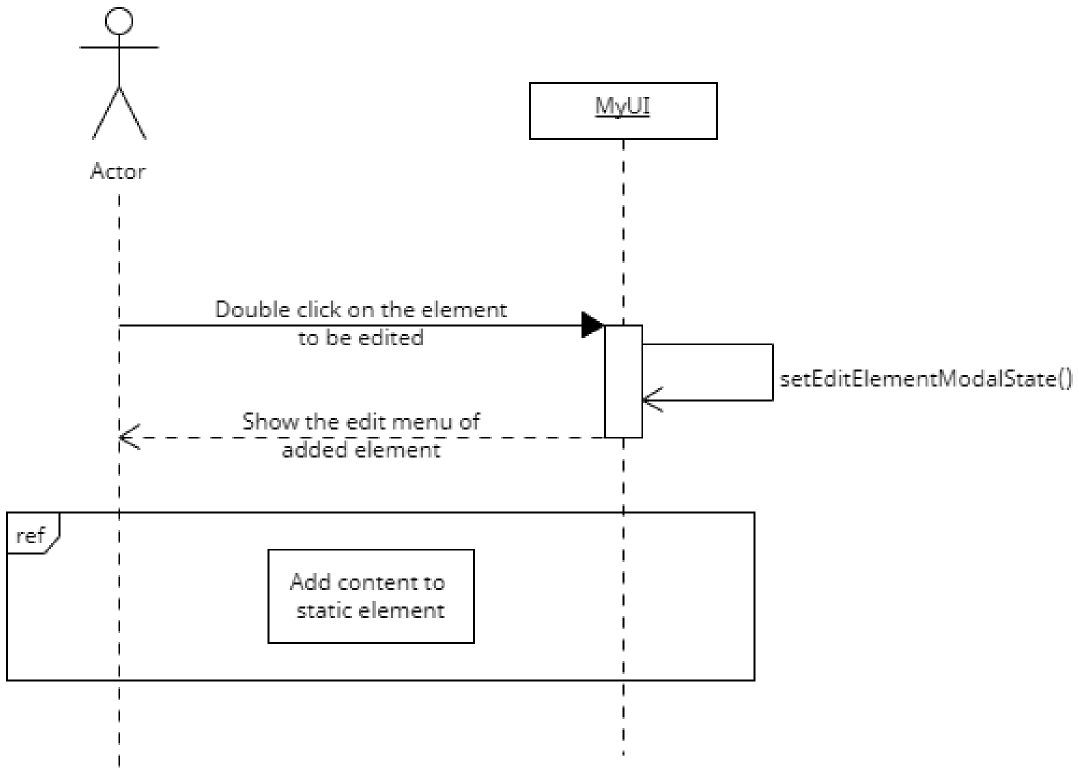

Actor
MyUI
Double click on the element to be edited
setEditElementModalState()
Show the edit menu of added element
ref
Add content to static element

Figure 3.39: Editing a static element UML sequence diagram

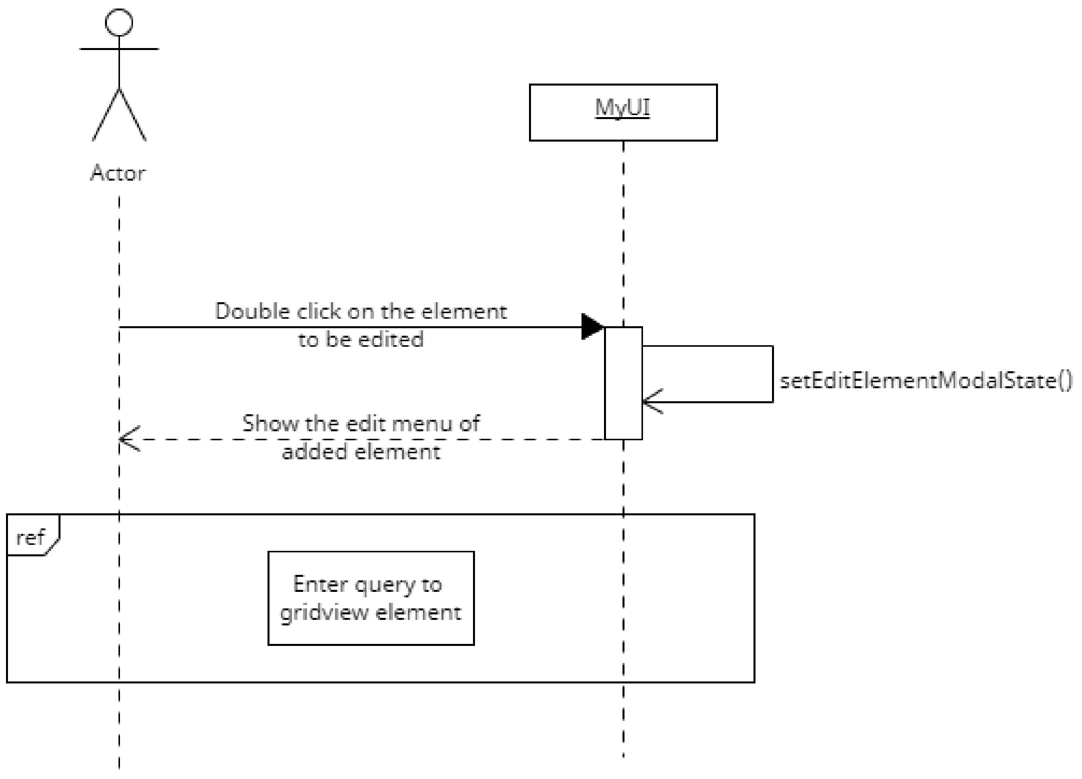

Actor
MyUI
Double click on the element to be edited
setEditElementModalState()
Show the edit menu of added element
ref
Enter query to gridview element

Figure 3.40: Editing a grid view element UML sequence diagram

Deleting an application element

Users can delete one of the elements they've previously added to their dashboard. When a user enters edit mode, a delete element button with a trash icon shows up on every element they have on the dashboard. Clicking on this button will yield a delete confirmation prompt for the element. Users can click on 'Yes' to delete or 'No' to cancel the deletion.

Goal: User removes an application element from the dashboard.
Preconditions:

- A dashboard is open in the workspace area.
- Edit mode is enabled for this user.
- The dashboard has an application element.

Postcondition: The application element is no longer present in the dashboard.

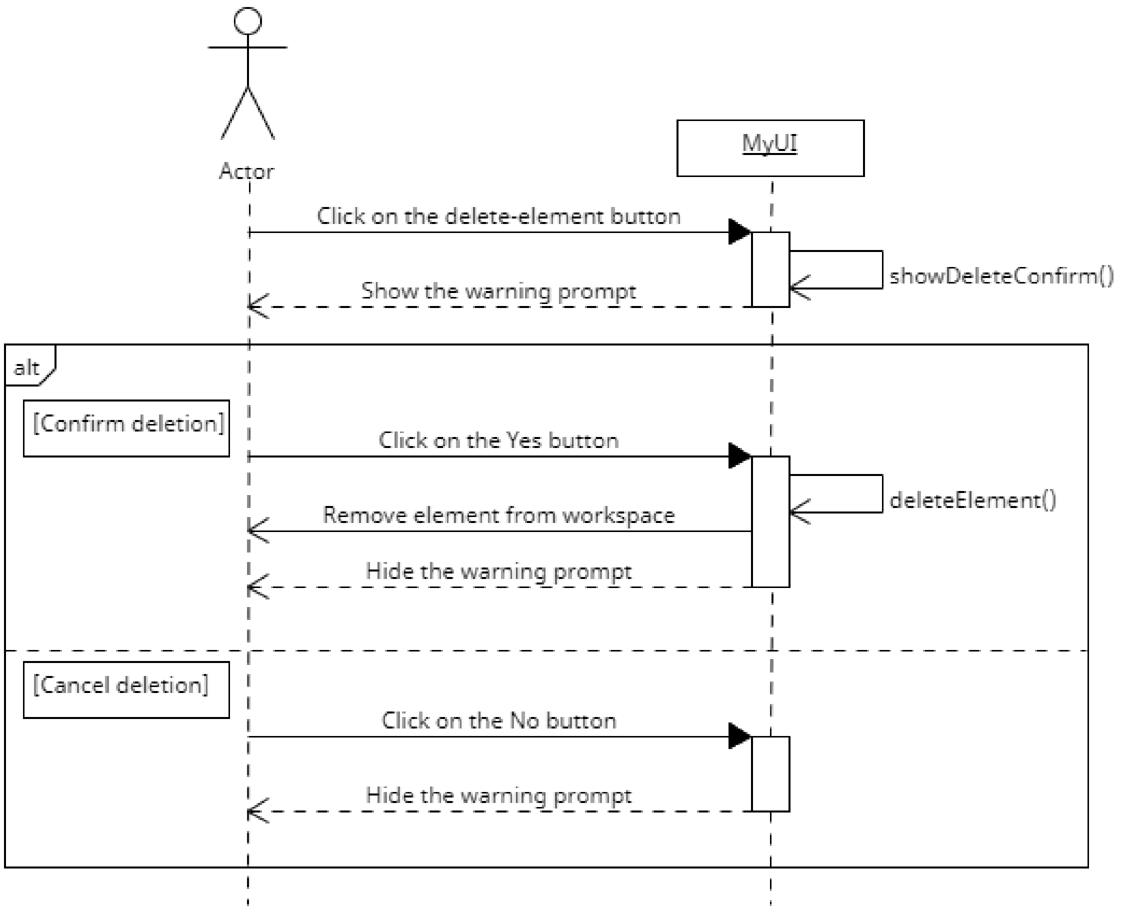


Figure 3.41: Deleting an application element UML sequence diagram

Saving a dashboard

When a user presses the save button while being in edit mode for a dashboard, MyUI will update the user configuration and notify the user of the successful save.

Goal: A dashboard is saved.
Preconditions:

- The user has a dashboard open.
- The user is in edit mode on the dashboard.

Postcondition: The user's user configuration is updated.

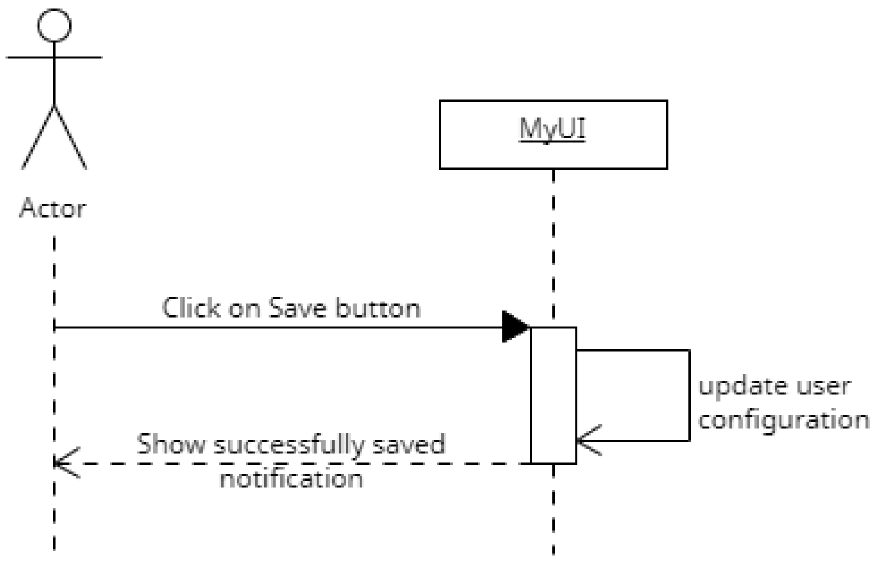

Actor
MyUI
Click on Save button
update user configuration
Show successfully saved notification

Figure 3.42: Saving a dashboard UML sequence diagram

Upload a dashboard

Users can upload a dashboard configuration when creating a new dashboard. After giving the new dashboard a name, the user can upload the dashboard configuration JSON file. After creation, the dashboard will be shown in the workspace.

Goal: Download a dashboard configuration.
Preconditions: N/A.
Postcondition: A new dashboard with the dashboard configuration is added and shown in the workspace.

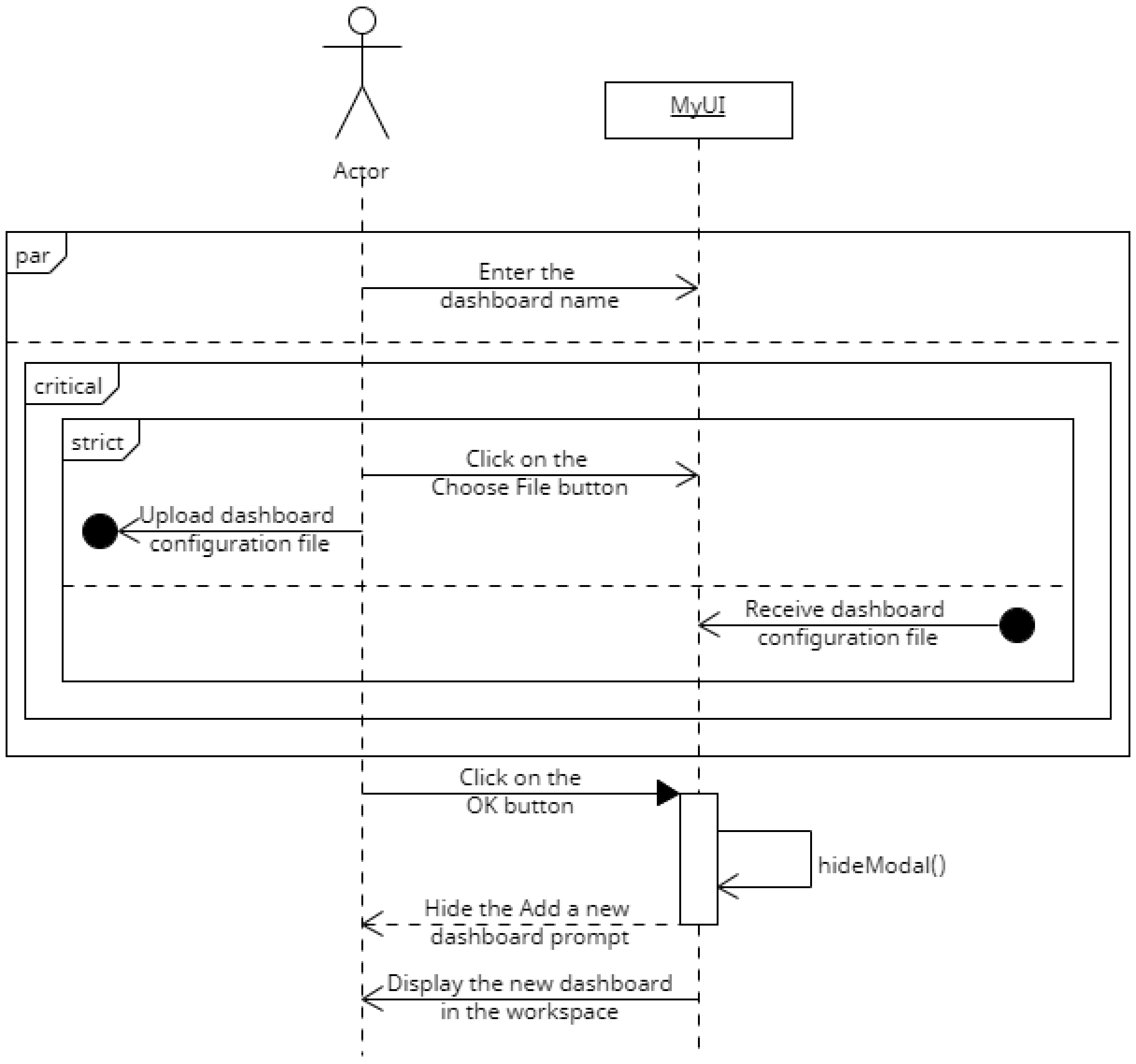

Actor
MyUI
par
Enter the dashboard name
critical
strict
Click on the Choose File button
Upload dashboard configuration file
Receive dashboard configuration file
Click on the OK button
hideModal()
Hide the Add a new dashboard prompt
Display the new dashboard in the workspace

Figure 3.43: Uploading a dashboard UML sequence diagram

Download a dashboard

Users can download a dashboard for which they are currently in edit mode by pressing the download button. The dashboard configuration will be downloaded as a JSON file.

Goal: Download a dashboard configuration.
Preconditions:

- User is in edit mode.
- A dashboard is present in the workspace area.

Postcondition: User receives the dashboard configuration in JSON format.

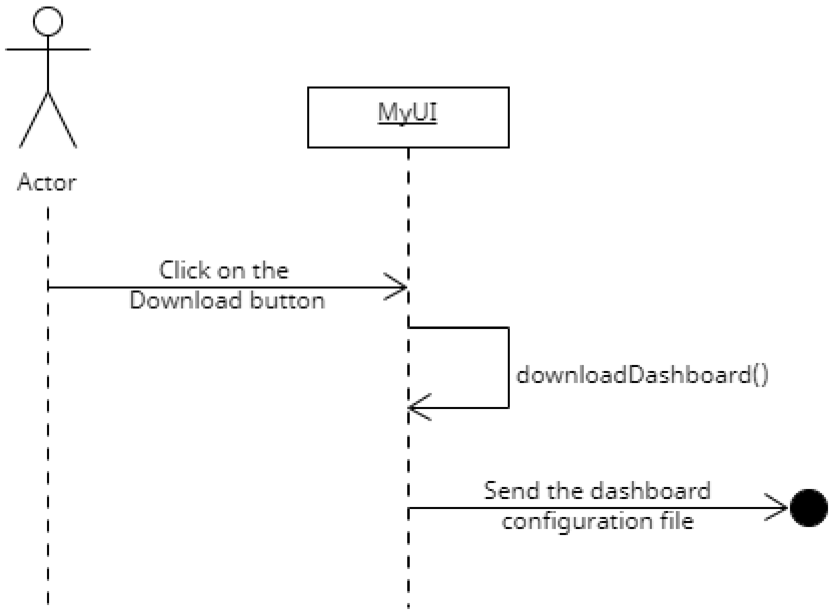

Actor
MyUI
Click on the Download button
downloadDashboard()
Send the dashboard configuration file

Figure 3.44: Downloading a dashboard UML sequence diagram

Change global settings

When an admin opens the global settings menu, they can change the media display settings to URL or media.

Goal: Global settings are changed.
Preconditions: The user menu is open.
Postcondition: The global settings are changed.
User: Admin.

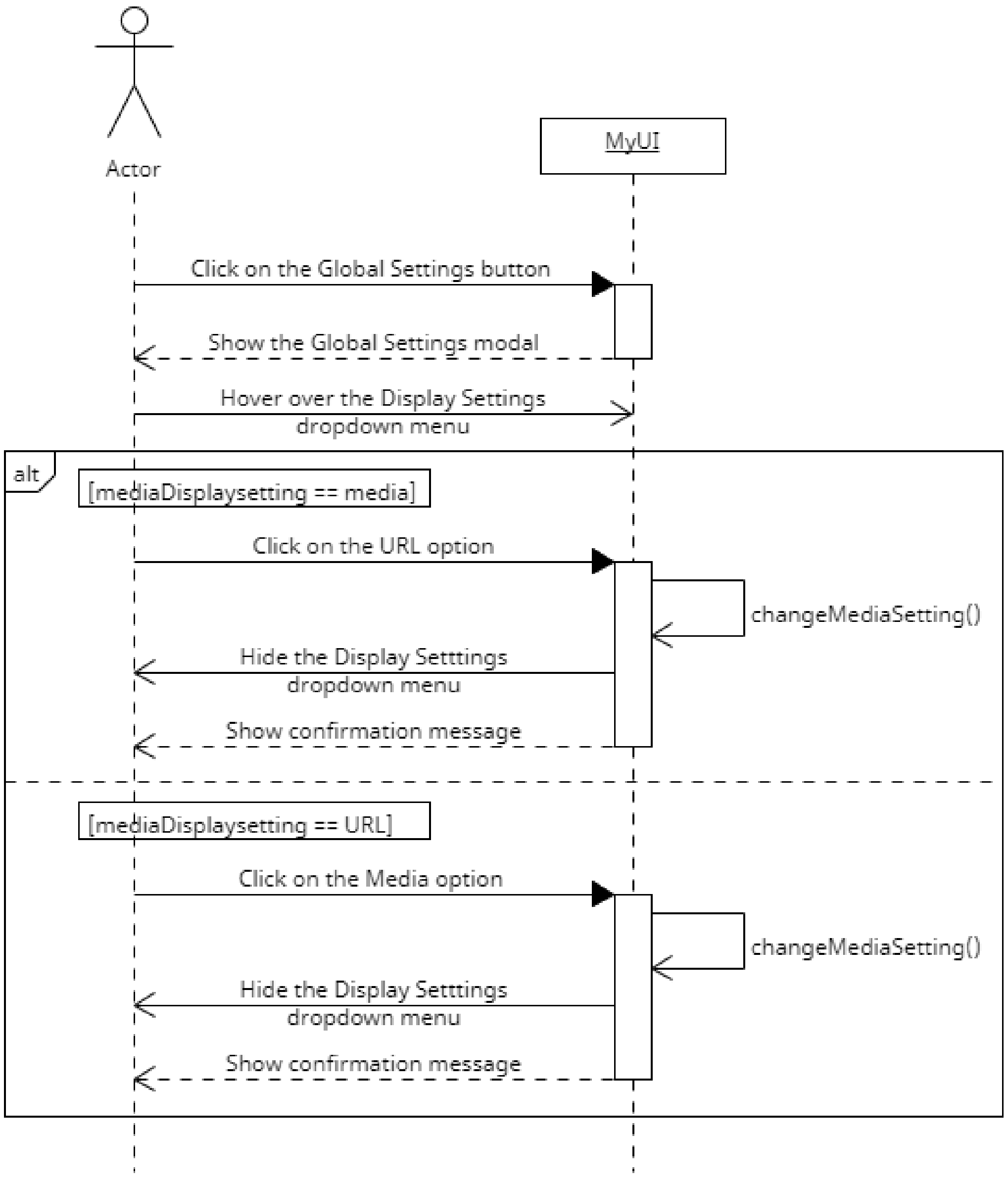

Actor
MyUI
Click on the Global Settings button
Show the Global Settings modal
Hover over the Display Settings dropdown menu
alt
[mediaDisplaysetting == media]
Click on the URL option
changeMediaSetting()
Hide the Display Setttings dropdown menu
Show confirmation message
[mediaDisplaysetting == URL]
Click on the Media option
changeMediaSetting()
Hide the Display Setttings dropdown menu
Show confirmation message

Figure 3.45: Changing the global settings UML sequence diagram

3.4 | Data model

The database of MyUI is implemented in PostgreSQL to store the relational data of the application. By altering the HASURA_GRAPHQL_METADATA_DATABASE_URL and PG_DATABASE_URL in `docker-compose.yml` of the source code of MyUI to the URL of their PostgreSQL database, an administrator can connect their own PostgreSQL database to the application. For MyUI to function according to its specification as defined in the URD [4], the database needs to adhere to some requirements:

- The database needs to have a single scheme called *public* in which all the tables are stored. Depending on the permission profiles of these tables, as defined in Hasura, these tables will be accessible to users within MyUI.
- The scheme *public* needs the following tables and attributes:
  1. The table *users*[2] has columns *id* (integer), *name* (text), *created_at* (date), *updated_at* (date), *password* (encrypted text using a 128-bit salt), *role* (user, editor, or admin).
  2. The table *user_versioned_config* has columns *user_id* (foreign key to *users*), *date* (date), and *config* (JSON string).

  These two tables allow for user authentication and the saving of the personalized user interfaces.

The ERD of the required tables is shown in Figure 3.46.

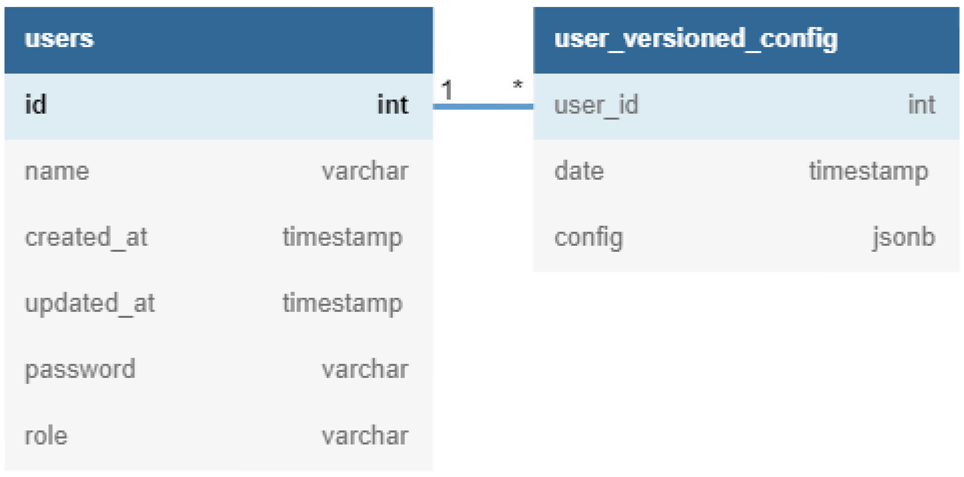


Figure 3.46: ERD

The administrator of the PostgreSQL and Hasura instance can add these tables by running the following SQL commands in the Hasura console:

```
CREATE TABLE "public"."users" ("id" serial NOT NULL, "name" text NOT NULL, "password" text
NOT NULL, "created_at" timestamptz NOT NULL DEFAULT now(), "updated_at" timestamptz NOT NULL
DEFAULT now(), PRIMARY KEY ("id") );

CREATE TABLE "public"."user_versioned_config" ("user_id" integer NOT NULL, "date" timestamptz
NOT NULL DEFAULT now(), "config" json NOT NULL, PRIMARY KEY ("user_id","date") , FOREIGN
KEY ("user_id") REFERENCES "public"."users"("id") ON UPDATE restrict ON DELETE restrict);
```

[2] If there already exists a table called *users* in the database, this table can be renamed to any other name after changing the source code of MyUI

These two tables can also be manually added to the database of MyUI by an administrator, but this is not recommended.

To be able to use the applications, users need to be added. This can either be done by the administrator of the PostgreSQL and Hasura instance via the Hasura console, or by a user with the *admin* role in MyUI. By default, users with an *admin* role can perform any operation on any table. By creating at least one user with an *admin* role, this user can add new users within MyUI. As MyUI hashes all passwords automatically, it is recommended to use the latter approach for adding users.

3.5 | External interface definitions

The external systems that MyUI communicates with are covered in this section. We shall provide a description of the interfaces required by MyUI to guarantee the proper interaction with the external systems.

The GraphQL API offered by the Hasura GraphQL Engine is the primary external interface used by MyUI. The Hasura GraphQL Engine also enables the configuration of REST endpoints, which MyUI uses in situations where the GraphQL API is unusable. The next two sections will elaborate on the Hasura GraphQL API and the Hasura REST Endpoints.

3.5.1 | Hasura GraphQL API

As mentioned in Section 3.1.1, the context in which MyUI can be plugged in consists of a PostgreSQL database and Hasura. The primary external interface used by MyUI is the GraphQL API provided by Hasura. This API allows users to fetch and update data in the PostgreSQL database, which is done through Hasura actions. These actions consist of GraphQL queries sent by MyUI that are handled in accordance with the permissions of the currently logged in user. Hasura sends these actions to the PostgreSQL database, thereby allowing users to indirectly interact with the data on the database. The use of these actions requires that a user permission profile is present. However, this is not the case for all database interactions carried out by MyUI. The following subsection will describe these interactions and how they are handled. Figure 3.47 gives a graphical representation of the context of MyUI.

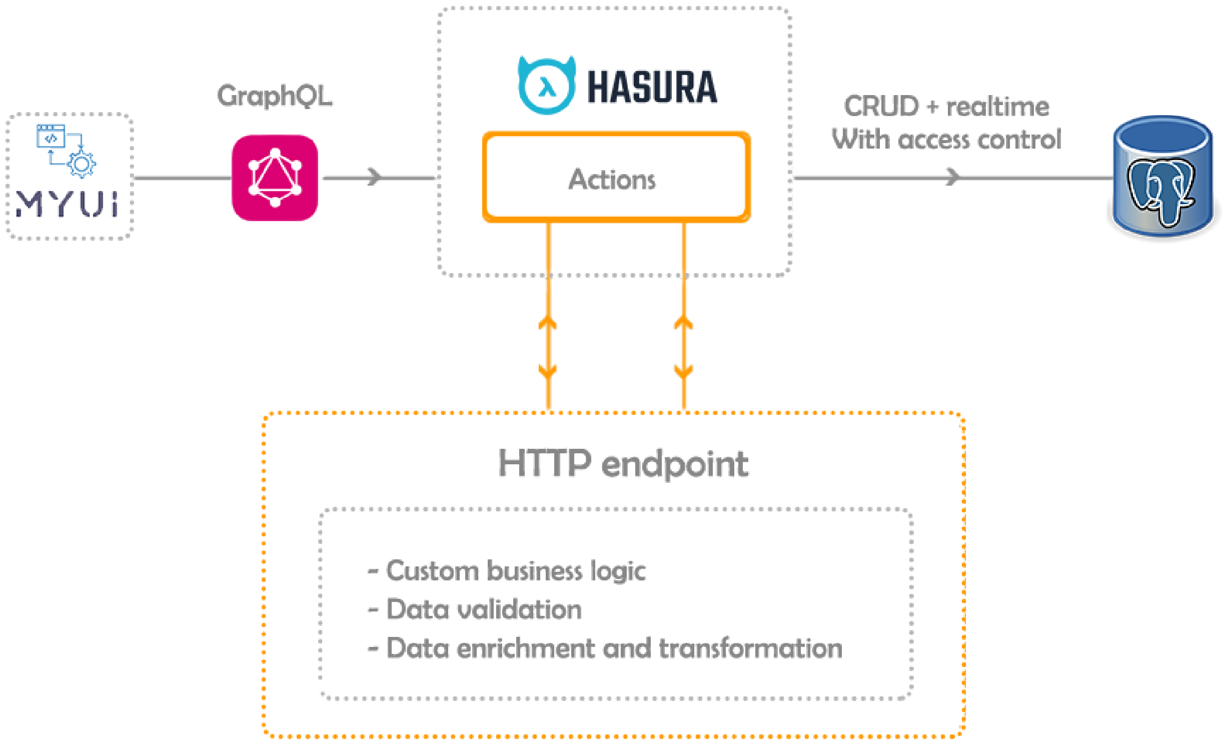


Figure 3.47: The context of MyUI

3.5.2 | Hasura REST Endpoints

For the use-cases where the GraphQL API is not used, Hasura's REST endpoints are used instead. These endpoints do not require user permissions to be present. Furthermore, using the REST API model also provides us with the advantages of caching GET requests, which improves responsiveness. Hasura's Rest endpoints are used in the following use-cases:

- **Login**: Before logging in, the permissions of a user are not yet known by MyUI. Because of this, a REST API endpoint is used to fetch a list of all users and their credentials during login.
- **User configuration**: Fetching the latest configuration of a user after login is considered a global query, and is not affected by the role-base access control as defined in Hasura. Using a REST endpoint for this query makes sure that there are no discrepancies between users with different permissions while loading their configuration after login.
- **Introspection query**: To get information about the structure of the database a specific query called an introspection query is used. As this query will be sent every time the user accesses the application, the query is executed using a REST endpoint to make use of its caching functionality.

When a REST endpoint is created in Hasura, it stores the data obtained from a GraphQL query or mutation and returns this data when a request is made to one of the endpoints. One can specify the methods that are allowed for the created endpoint such as GET, HEAD, POST, PUT, and DELETE.

The process of using the REST endpoints is as follows:

1. The REST endpoints are created in Hasura by executing the `hasura metadata apply` command which is a part of the `make setup` command as described in MyUI's STD [5].
2. All calls to the Hasura REST endpoints are made in the NextJS API routes, thus in the */pages/api/* directory in order to adhere to the framework principles of NextJS.
3. The IP address of the Hasura GraphQL engine is obtained in the back-end code with the help of the IP library. [6]
4. In the request to the REST endpoint, the authentication headers for Hasura still need to be passed. Depending on the query, this authentication header can be either one of the following two options:
   - A bearer token [7] along with the JWT token of the current session as generated by the authentication library in a compatible format [8].
   - The Hasura admin secret key [9].
5. After obtaining the data from Hasura, this data is processed in MyUI.

Figure 3.48 provides a graphical representation of the implemented REST endpoints in Hasura, as used in MyUI.

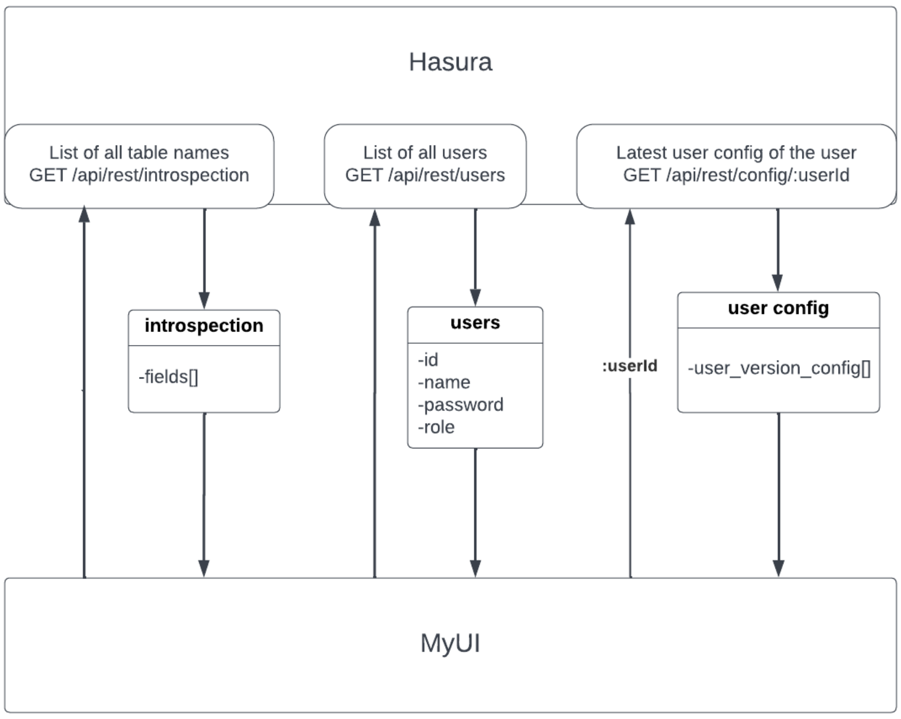


Figure 3.48: Hasura REST Endpoints

- **Introspection**:
  - Route: `/api/rest/introspect`
  - Methods: `GET`
  - This endpoint returns all the available table names and is used to generate the list of base tables on the client-side, based on the user's permission profile.
- **Users**:
  - Route: `/api/rest/users`
  - Methods: `GET`
  - This endpoint returns all the users in the *users* table along with their passwords and roles. This is used by the authorization logic in the back-end for checking the validity of the credentials entered by users at login.
- **User config**:
  - Route: `/api/rest/config/:userId`
  - Methods: `GET, POST`
  - This endpoint returns the user configuration of the user with ID passed in the *userId* route parameter.

3.6 | Design rationale

Initially, we built MyUI using different routes for each individual page. When switching between pages using this method, all components had to be reloaded. This resulted in an inconvenient user experience. To tackle this issue, the decision was made to develop a single-page application. A single-page application dynamically rewrites the current webpage, as opposed to loading an entirely new page. This absence of page refreshes resulted in a faster application improving both the responsiveness and user experience of our application.

In the remainder of this section, we will discuss the design decisions taken to build this single-page application. We will explain the rationale behind each design decision, as well as any trade-offs and potential alternatives that were taken into account.

3.6.1 | TypeScript

As the main functionality of our application is the User Interface, which implies a front-end oriented workflow, we were in need of a front-end oriented programming language. Because several members of the group had prior experience with TypeScript and our customer recommended it, we chose to use TypeScript [10], a superset of JavaScript. Using TypeScript allowed us to catch bugs early on in the development process and to increase the readability of our code. As opposed to JavaScript, which is dynamically typed and hence only checks types at run-time, TypeScript utilizes static typing. This implies that type-errors cannot make their way into the production phase.

3.6.2 | React

To make the development of the application easier, we decided to use a front-end library for building the user interface. There are a few widely used front-end libraries we considered, namely React [11], Angular [12], and Vue [13]. From these three, the customer recommended using React, while highly discouraging using Vue. Angular offers a two-way data binding system, which means that the model state changes if a UI element is changed. This increases the difficulty of understanding the code. React, on the other hand, offers one-way data binding which makes the code easier to understand.

All of the aforementioned factors, as well as the fact that some team members had prior experience with React, convinced us to use React as the front-end library of MyUI.

3.6.3 | Ant Design

We decided to use a front-end library that would provide us access to fundamental components like buttons, tables, and modals so that we wouldn't have to design every component used in MyUI ourselves. There are various libraries that offer such components, namely Ant Design [14], MUI Core (formerly known as MaterialUI) [15], and Bootstrap [16]. For MyUI, some components were of significant importance. For example, an alert to signify to the user that an error has occurred. Table 3.21 displays a few necessary components as well as their accessibility in the distinct libraries.

Table 3.21: Comparison of components Ant Design, MUI Core, Bootstrap [17]

	Ant Design	MUI Core	Bootstrap
Number of components	60	35	35
Alert / Messages	✔	✗	✔
Loader	✔	✗	✔
Notifications	✔	✗	✗
Popover	✔	✔	✗
Tooltip	✔	✗	✔

As can be seen in the table above, all necessary components are implemented by Ant Design. Moreover, it has the highest number of components out of all options. Furthermore, Ant Design is written in TypeScript which assures compatibility with MyUI.

Keeping the aforementioned in mind, we chose Ant Design over MUI Core or Bootstrap.

3.6.4 | i18next

The language requirements stated in the URD of MyUI [4] implied the need for internationalization. i18next [18] is an internationalization framework for JavaScript, with React integration. The framework makes use of plurals, context, interpolation, and localization as a service. This enabled us to quickly add more languages on top of the ones that are already accessible. As the framework provided us with all needed functionalities to apply internationalization in our application, we did not further consider alternative frameworks.

3.6.5 | Node.js

Node.js [19] is the framework, built on top of Google Chrome's V8 JavaScript Engine, that provides the environment to run the application with React. The primary benefit of Node.js over other libraries with comparable features is that processes can be executed in the background while the server deals with additional requests. Without interfering with ongoing queries, Node.js pulls data from earlier requests. This allows the application to handle server requests in real-time. Its nature as an event-driven, non-blocking model makes it the ideal candidate to build scalable, data-intensive network applications. Due to the framework's widespread adoption and documentation, we chose to use it for MyUI.

3.6.6 | Next.js

Next.js [20] is the other primary framework that was used for the back-end. Next.js is a JavaScript framework built on top of React to help developers in building high-performing online applications and fast static webpages. We decided to use Next.js due to its support for TypeScript, rapid refresh, API routes, and code splitting. The rapid refresh streamlines the development process by enabling developers to automatically see their changes in the application without having to manually refresh. A trade-off was made between the time spent learning about Next.js and the features it would give us because no team member had any prior familiarity with it. As Hasura is commonly used together with Next.js [21], no alternative to Next.js was considered.

3.6.7 | React Query

To handle data-fetching in the project, we needed a data-fetching library that worked with React because we used it to build our application. In order to integrate with Hasura, the library also needed to support the GraphQL query language. There were three libraries that met our needs: React Query, SWR, and Apollo Client. We found that React Query offered more functionality that would be relevant for our application after evaluating these three libraries, as shown in Table 3.22. For instance, React Query allows the cancellation of queries, which the other two libraries do not support [22].

Table 3.22: Comparison functionalities React Query, SWR, and Apollo Client [22]

	React Query	**SWR**	**Apollo Client**
Platform	React	React	React, GraphQL
Query Syntax	Promise, REST, GraphQL	Promise, REST, GraphQL	GraphQL
Queries	✔	✔	✔
Query Cancellation	✔	✗	✗
Offline Caching	✔	✗	✔

3.6.8 | Database Management System

Postgres, a popular open-source DBMS was used as the database for MyUI. As MyUI interacts with the database via Hasura, only databases that are supported by the Hasura GraphQL engine were considered during development. The Hasura GraphQL engine supports the following DBMS: Postgres, Citus, MC SQL Server, and BigQuery [23]. The database-wise support for the different GraphQL features under schema, mutations, and subscriptions, are shown in Tables 3.23, 3.24, and 3.25. The queries GraphQL features are supported for all databases.

Table 3.23: Schema supported features [23]

	Postgres	Citus	SQL Server	BigQuery
Table Relationships	✔	✔	✔	✔
Remote Relationships	✔	✔	✔	✔
Views	✔	✔	✔	✗
Functions	✔	✔	✗	✗
Enums	✔	✔	✗	✗
Computed Fields	✔	✔	✗	✗
Data Validations	✔	✔	✔	✔
Relay Schema	✔	✔	✗	✗

Table 3.24: Mutations supported features [23]

	Postgres	Citus	SQL Server	BigQuery
Insert	✔	✔	✔	✗
Upsert	✔	✔	✔	✗
Update	✔	✔	✔	✗
Delete	✔	✔	✔	✗
Multiple per Request	✔	✔	✗	✗

Table 3.25: Subscriptions supported features [23]

	Postgres	Citus	SQL Server	BigQuery
Value of Field	✔	✔	✔	✗
Updates to Rows	✔	✔	✗	✗
Value of Derived Field	✔	✔	✔	✗

From the tables, it is clear that all GraphQL features are supported for both Postgres and Citus. Hence, these make for the most viable candidates. Citus [24] is an open-source extension that transforms Postgres into a distributed database. All of Postgres' capabilities, tooling, and ecosystem may be used because it is a Postgres extension.

The software used previously by the customer, MotorAdmin, used both Hasura and Postgres for data storage. In order to maintain consistency, and because of the support of all GraphQL functionalities, Postgres was chosen as the database for MyUI.

3.6.9 | Package managers

Package managers are used to manage a project's dependencies, which include installing and updating packages that are required for the project to run. The importance of package managers grows as a function of the project size, since managing all the dependencies of a project becomes increasingly difficult. Hence package managers are advised to handle this functionality.

We started the development process by using npm [25]. The motivation behind this was that the initial boilerplate project was created using Next.js, using npm as its primary package manager. This is why we continued using npm in our project. However, we realized that for some libraries the package manager was resulting in compatibility and build errors that were blocking the development of MyUI. Examples of such libraries are bcrypt [26], which is used for encryption and authentication, as well as Jest [27], which was used for unit testing.

Because of this, we decided to add both these libraries to our project using the Yarn package manager [28]. Doing this solved the build errors and development could move forward. This is why our project currently implements the installation of libraries through both npm and Yarn.

3.6.10 | Docker

The customer requested that MyUI runs via image containerization using Docker [29]. Docker makes use of OS-level virtualization to package an application and all its dependencies into a *container*. These containers can be run on any Windows, Linux, or macOS system. This allows MyUI to be abstracted from the host OS, meaning that it can easily be moved between different systems while continuing to operate consistently. This enables us to have a near-identical development environment for all developers, regardless of their local settings. Moreover, Docker allows for multistage builds. A multistage build allows for a high level of flexibility by allowing the setup of development as well as production environments in the same project folder without any further configuration.

One could select an alternative containerization service, such as kubernetes [30], as opposed to Docker. The widespread acceptance of Docker, however, makes it easier to gather information from a variety of web resources. This, together with our customer's request, persuaded us to use Docker.

4 | Feasibility and resource estimates

This section gives an overview of the feasibility and resource estimates necessary to run the development and client systems for MyUI. A performance metric is also provided based on the resource requirements below and the performance requirements of the URD [4].

4.1 | Resource requirements

This section contains the minimum system requirements needed to run the MyUI application. As the application is run in Docker containers, the client is able to deploy the application on their own server. Below, we will give the server requirements for said server, depending on the operating system it uses. Moreover, we will give the requirements for the end-user system.

4.1.1 | Server requirements: Windows

Below are the minimum system requirements needed to run MyUI on a Windows machine [31].

Table 4.1: Windows requirements

CPU	64-bit processor with SLAT
Operating System	Windows 10 64-bit, Windows 11 64-bit
RAM	4GB
Disk Space	5GB[3]

4.1.2 | Server requirements: macOS

Below are the minimum system requirements needed to run MyUI on a macOS machine. In order to be able to run Docker, Intel's hardware support for memory management unit virtualization, including Extended Page Tables and Unrestricted Mode must be enabled. This implies that the hardware must be from 2010 or newer [32].

Table 4.2: macOS requirements

CPU	1.7 GHz Core i5 (I5-3317U)
Operating System	macOS Catalina (10.15) or higher
RAM	4GB
Disk space	5GB

4.1.3 | Server Requirements: Linux

Below are the minimum system requirements needed to run MyUI on a Linux machine [33].

Table 4.3: Linux requirements

CPU	x86_64 processor (Xeon, AMD 1.5 GHz) [34]
Operating System	x86_64 / amd64 architecture
RAM	4GB RAM
Disk space	5GB

[3]The minimum disk space, for all listed systems, has been estimated via addition of the minimum disk space needed to run each Docker container, Hasura, and an empty Postgres database.

4.1.4 | End-user system requirements

Below are the minimum system requirements needed for an end-user to run MyUI. An end-user is not limited by a specific operating system.

Table 4.4: End-user system requirements: Windows

CPU	Intel Pentium 4 processor with SSE3
Operating System	Windows 7
Network	1 Mbps is recommended
Browser	Google Chrome Version 101, Mozilla Firefox Version 99, Microsoft Edge Version 100

Table 4.5: End-user system requirements: macOS

CPU	2.0 GHz Intel Core 2 Duo Merom
Operating System	macOS El Capitan (10.11) or higher
Network	1 Mbps is recommended
Browser	Google Chrome Version 101, Mozilla Firefox Version 99, Microsoft Edge Version 100 and Safari Version 15

Table 4.6: End-user system requirements: Linux

CPU	x86_64 processor (Xeon, AMD 1.5 GHz)
Operating System	64-bit Ubuntu 18.04+, Debian 10+
Network	1 Mbps is recommended
Browser	Google Chrome Version 101, Mozilla Firefox Version 99, Microsoft Edge Version 100

4.1.5 | Additional software

Listed below are the additional software and their respective versions needed to run MyUI on a server. Other needed software and packages are automatically installed when running `make setup` as described in Chapter 3 of the STD of MyUI [5].

Table 4.7: Additional Software

Name	Version
Docker	v20.10
Hasura	v2.8.1
npm	v8.13.2
Yarn	v22.19
Node.js	v10.19.0

4.2 | Performance metrics

The performance of MyUI is thoroughly dependent on the size of the database and server being used. The modular design principles on which MyUI is built imply that every Postgres database can be connected to MyUI via Hasura, as long as the preconditions described in Chapter 3.4 hold. Hence, no specific performance metrics can be given and/or tested.

The computational effort will grow at least linearly with the table size. In order to handle databases with large tables, MyUI makes use of an in-built query time limit present in Hasura. While the query is being computed, a throbber will be displayed. Once the time limit has been reached, a warning stating that the time limit has been exceeded will be displayed.

External factors may also influence MyUI's performance. Think, for example, of network latency, DNS issues, and the user's system hardware. These factors are however not governable by MyUI. As a result, they are out of the scope of this application. Hence we can conclude that MyUI's performance is mainly correlated to the size of the database and the hardware of the user's system.