CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Bachelor's Thesis

Modular Dashboard

TRPUX

Patrik Dvořáček
Software Engineering and Technologies

May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Dvořáček Patrik** Personal ID number: **492035**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Software Engineering and Technology**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Modular Dashboard

Bachelor’s thesis title in Czech:

Modulární nástěnka pro CI

Guidelines:

1. Research and analyze the possibilities of visualizing aggregated automated test results.
2. Design a database model for representing data sources and data cards displayed to the user.
3. Implement a modular backend application for retrieving data in a unified format with sample modules for Jenkins and TeamCity.
4. Implement a website frontend corresponding to the developed backend and displaying the retrieved data.
5. Evaluate the viability of your solution via user testing with at least 5 participants.

Bibliography / sources:

[1] P. Duvall, S. Matyas III, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk, 2007
[2] R. Fielding, REST: Architectural Styles and the Design of Network-based Software
rchitectures, 2000
[3] N. Rozentals, Mastering TypeScript: Build enterprise-ready, modular web applications using TypeScript 4 and modern frameworks, 4th Edition, 2021

Name and workplace of bachelor’s thesis supervisor:

Ing. Michal Vaněk Department of Economics, Management and Humanities FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **11.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

Ing. Michal Vaněk Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgement / Declaration

I wish to express my sincere thanks to my supervisor, Ing. Michal Vaněk, for his continued guidance during this project. I am also grateful to Ing. Martin Ledvinka for his well-placed advice. Lastly, I would like to express my gratitude to Ing. Victoria Usan, Ing. Ondřej Gróf, and Bc. Hana Hrubešová for their support and encouragement.

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses. In Prague 20.05.2022

.....

Abstrakt / Abstract

Tato bakalářská práce si kladla za cíl popsat návrh a následný vývoj modulárního webového prostředí pro zobrazování informací z různých datových zdrojů. Největší důraz byl kladen na vytvoření modulárního systému dovolujícího následné rozšíření systému třetí stranou, aniž by byla vyžadována úprava již implementovaných částí. Pro systém byly navrhnuty ukázkové moduly pro serverovou část i webové uživatelské prostředí, jež byly následně předloženy uživatelům k otestování. Dle jejich zpětné vazby je systém připravený ke každodennímu používání. Každopádně je modulární systém připraven na rozšíření skrze nové moduly a uživatelská rozhraní.

Klíčová slova: Node.js, TypeScript, React, Webová aplikace

Překlad titulu: Modulární nástěnka pro CI (TRPUX)

The goal of this bachelor thesis was to design and implement a modular web interface for displaying information from various data sources. The most significant emphasis was placed on creating a modular system that allows for third party extensions without the necessity for updates to the already implemented parts. Example modules were created for both the server part and web user interface. These were then presented to users for testing. According to their feedback, the system is ready for everyday use. Nevertheless, the modular system is prepared for extensions through new modules and user interfaces.

Keywords: Node.js, TypeScript, React, Web Application

Contents /

1 Introduction **1**
1.1 Motivation 1
1.2 Aim of the Project 1

2 Analysis **3**
2.1 Existing Solutions 3
2.1.1 Native Features and Plugins . 3
2.1.2 Kibana 3
2.1.3 Grafana 4
2.1.4 Mozaïk 4
2.2 Functional Requirements 4
2.2.1 Client Requirements 4
2.2.2 Administrator Requirements 4
2.2.3 Developer Requirements . . . 5
2.3 Non-functional Requirements . . 5

3 Design **6**
3.1 Architecture 6
3.1.1 System Context 6
3.1.2 Container 7
3.1.3 Component 7
3.2 Database 8
3.3 API 8
3.4 Web Application 9

4 Implementation **10**
4.1 Database 10
4.2 Server 10
4.3 Web Application 11
4.4 Modularity 11
4.4.1 Parsers 11
4.4.2 Cards 13
4.5 Default Modules 13
4.5.1 Single Value Card 14
4.5.2 Line Graph Card 14
4.5.3 Iframe Card 15
4.5.4 Matrix Card 15
4.5.5 Jenkins Parser 15
4.5.6 TeamCity Parser 17
4.6 Testing 17
4.6.1 Server 17
4.6.2 Web Application 17
4.7 Deployment 18

5 Evaluation **19**
5.1 Client Evaluation 19
5.1.1 Testing Scenario 19
5.1.2 Findings 19
5.1.3 Conclusion 20
5.2 Administrator Evaluation . . . 21
5.2.1 Testing Scenario 21
5.2.2 Findings 21
5.3 Developer Evaluation 22
5.3.1 Parser Creation 22
5.3.2 Card Type Creation 22
5.3.3 Card Creation 22
5.3.4 Authentication Management 23
5.4 Non-Functional Evaluation . . 23

6 Conclusion **24**

References **25**

A Glossary **27**

B Container Diagram **28**

C Component Diagram **29**

D User Interface Preview **30**

E User Manual **33**

F Attached Media Contents **34**

/ Figures

3.1 C4 context diagram 7
4.1 Single value card 14
4.2 Line graph card 15
4.3 Iframe card 16
4.4 Matrix card 16
4.5 Deployment diagram 18
5.1 Dashboard 20
5.2 Administration 21
B.1 C4 container diagram 28
C.2 C4 component diagram 29
D.3 Dashboard Administration 30
D.4 User Administration 30
D.5 Server Creation Modal 31
D.6 Default Login Screen 31
D.7 Default Signup Screen 32
D.8 Card Creation Dialog 32

Chapter 1
Introduction

As the size and complexity of software systems rises, so does their suite of automated tests. This ever-growing system, in turn, complicates the ability of developers and quality assurance engineers to react to adverse test results swiftly. With either no or ad hoc ways to detect failures, easily fixable issues in the test environment can escalate into critical problems in production. A graphical representation, most often in a dashboard, is thus used to circumvent either slow analysis or disordered display of test results. However, most of these solutions are usually tied to a particular continuous integration environment, such as Jenkins or TeamCity.

Nevertheless, in larger organisations, two or more different test environments can be used alongside. In turn, multiple graphical representations of test results need are utilised. In these situations, developers again find themselves analysing various data points at numerous origins. This creates a demand for a centralised solution to query all relevant sources and display their results in a unified interface.

1.1 Motivation

The need for such a system has arisen at Avast Software and gave inspiration to this project. Avast utilises a diverse suite of testing methods, continuous integration environments [1], and reporting systems across the company, teams, and sometimes even products. Such a fractured workspace makes developers' day-to-day work harder, in addition to complicating a high-level overview. Furthermore, while the initial idea aimed to unify only two of these environments, it has grown into a flexible system with more generalised usability.

The first implementation of this project at Avast came before the inception of the general concept and was limited in scope to a single continuous integration environment, Jenkins. The display methods were also limited to a line graph displaying the success rate of tests of a selected job over time. Additionally, the development was stopped as the initial project had achieved its limited purpose. Nevertheless, the project was restarted, and its design has gone through a couple of iterations before reaching the current modular incarnation.

1.2 Aim of the Project

This project aims to design and implement a modular web application that collects and displays real-time data from various sources in a dashboard. The data and system administration, such as user and dashboard management, should be available through a REST API. The system should also be ready to work with a custom front-end portion, even though a single page application is implemented and provided by default.

The biggest hurdle of this project lies in the modular aspect. Because the backend system is implemented in TypeScript [2], a strongly-typed superset of JavaScript, the

modules must follow the typing system's conventions. Not only that, the implementation must be developer-friendly to encourage the creation of custom modules without having a deep understanding of the system's inner functionality.

On the other hand, the project also aims to make the deployment of the system as easy as possible. This constraint implies implementing sane default values to get the system up and running quickly in development without extensive configuration while staying flexible enough to be viable for deployment to a large audience in a more complex setting.

Chapter 2
Analysis

The requirements for the system came from interviews with stakeholders inside Avast Software. Mainly, their needs were concerned with the various display options each of the stakeholders had. The final design requirements contain a wide range of dashboard configurations. For developers and quality assurance engineers, this meant detailed statistics about individual products and test suites. At the same time, managers and other higher-level or non-tech staff asked for top-level assessments.

During this exploration phase, the idea to broaden the project's scope beyond software development crystallised. Since modern people management, finance and other administrative work is done entirely or at least partially through software, it could also be visualised in a dashboard. And with an increased pool of possible data sources, the modular system requirement became essential.

2.1 Existing Solutions

Implementation and deployment of existing solutions have been considered as well. Ultimately none of them fit the given criteria either due to limitations in extendability, missing required core features or the inability to be deployed inside an internal network.

2.1.1 Native Features and Plugins

Native features, like TeamCity Dashboard [3] or JIRA Dashboard [4], and plugins, like BlueOcean Dashboard for Jenkins [5], were first in line to be considered, and all were quickly discarded. Their extendability to provide different visualisation options or retrieve data across environments was either extremely limited or non-existent. Similar problems arose when custom plugins for these systems were proposed. They would work exclusively with a single environment, and their extension to support different systems would be nearly impossible. For these reasons, native solutions were discarded completely.

2.1.2 Kibana

Kibana [6] offers extensive visualisation options and is widely adopted throughout the industry, offering a vast knowledge base and user support. Kibana's source code is openly available and supports custom plugin creation. Unfortunately, Kibana is closely tied to Elasticsearch [7], a system designed to store large amounts of data and provide analysis over them.

Deployment of Kibana with Elasticsearch would mean forced duplication of data and latency between data availability at the source and visualisation on the dashboard, possibly making time-sensitive data outdated. In light of the modular spirit of the system, this feature seemed too restrictive. Additionally, a sizable amount of resources would have to be deployed to accommodate Elasticsearch's needs. Thus, Kibana was not considered for deployment.

2.1.3 Grafana

Grafana [8] is another dashboard system with industry-wide adoption. The open-source version can be deployed inside an internal network, supports custom plugin development and does not rely on a specific data storage technology. Additionally, Grafana boasts a suite of advanced features such as alerts, data snapshots and templates.

The only downside of Grafana is its authentication system. By default, it is limited to OAuth, LDAP, and Auth Proxy [9]. Although these options are more than sufficient in most cases, they are not flexible enough for the modular system. The Auth Proxy feature could be adapted, but it requires an external service to handle the authentication, even if a simple email and password combination would be used. As the implemented system is intended to be deployable by small teams for their purposes, a more extendable and compact authentication is required.

2.1.4 Mozaïk

Mozaïk [10] was found to be the closest available implementation of the requirements. Like Grafana, it is open-source, heavily extendable via user-created packages and queries data in real-time. Even with these features, Mozaïk was not adopted. One of the biggest hurdles would be the deployment itself - Mozaïk uses pure JavaScript instead of the type-safe superset, TypeScript.

A complete or partial rewrite would be needed to allow the codebase to work with plugins written in TypeScript and mitigate any possible issues from the non-type-safe code. Furthermore, while this would be feasible, the project has not been maintained for years. The last official release is dated April 2016, almost five years before the work on this project started. With such a high barrier to getting it up-to-date, Mozaïk was not considered as well.

2.2 Functional Requirements

The functional requirements can be divided into two categories: user and developer. The former contains the needs of end-users interacting with the web interface and can be further subdivided into clients and administrators categories. The latter pertains to developers creating custom modules and deploying the system for end-users.

2.2.1 Client Requirements

- Account management: Users should be able to log into their accounts. Furthermore, if enabled by the developer, new users should have an option to create an account.
- Dashboard management: Users should be able to view, create, update and delete their dashboards.
- Card management: Users should be able to view, create, update and delete cards inside their dashboards. This includes the assignment and customisation of sources from which the data for the card should be loaded.

2.2.2 Administrator Requirements

- User management: Administrators should have the ability to create, update and remove user accounts, including those of other administrators.
- Source management: Administrators should be able to add, update and remove data sources and load their data points.
- Dashboard management: Administrators should have the ability to create, update and remove dashboards.

2.2.3 Developer Requirements

- Parser creation: Developers should be able to create custom parsers. Parsers should then be automatically loaded into the system without complex configuration.
- Card type creation: Developers should be able to create custom card types. The system should then use them without violating the type constraints set by TypeScript.
- Card creation: Developers should be able to create new card components to display the data from parsers.
- Authentication management: Developers should have the ability to write custom authentication management algorithms for the system.

2.3 Non-functional Requirements

The non-functional requirements stem from the exploration process as well as from modern software practices.

- Open-source: Transparent codebase prevents end-user problems from being hard or downright impossible to diagnose. In addition, anyone can suggest changes or improvements.
- Ease of use: The project should follow best modern practices both on the development and user experience sides.
- Security: The system should not be able to leak any potentially sensitive data. Furthermore, the system should not become a possible attack vector inside the network where it is hosted.
- Lightweight: The system should run on most modern hardware without taking a significant amount of resources.

Chapter 3
Design

3.1 Architecture

In the spirit of modularity, the client-server model was chosen [11]. As most communication in modern networks happens over HTTP, client-server structure communicating over REST API [12] enables easy setup and maintenance. Moreover, it gives the maintainers a wide range of server hosting options and distributed access, including load balancing or instance duplication.

Furthermore, the REST API allows the implementation of various client interfaces, not only a web application. While a website is the most user-friendly means of interacting with the system, the API opens the possibility to use a wide range of interfaces for the same instance. Among others, the REST API allows for communication with mobile applications or even other APIs that could query the system for further use.

In the following analysis, the C4 model [13] is used as the framework for architecture visualisation. It offers four levels of abstraction to represent a system and its subsequent components. The fourth and lowest level is generally not used, as it is too specific to convey any generally helpful information. It is thus omitted.

3.1.1 System Context

The system context diagram [14] gives a high-level overview of the system. The diagram can be found in the figure 3.1. The main focus of this diagram is to show the actors using the described system and other systems interacting with the main target.

In the case of TRPUX, it contains the system itself, three actors and various data sources abstracted as a single node. The main system consists of the backend portion providing the REST API and an unspecified frontend.

The End User actor represents the people viewing and managing dashboards and cards. The Administrator actors are responsible for deploying and maintaining the system and handling the administrative tasks concerned with users and data sources. Lastly, the Developer actor represents the creators of parsers, cards and custom authentication modules for the TRPUX instance.

The External Data Sources node encapsulates the various data sources the TRPUX instance queries for data to display. These may range from REST APIs through databases to completely custom solutions, depending on the implementation of used parsers.

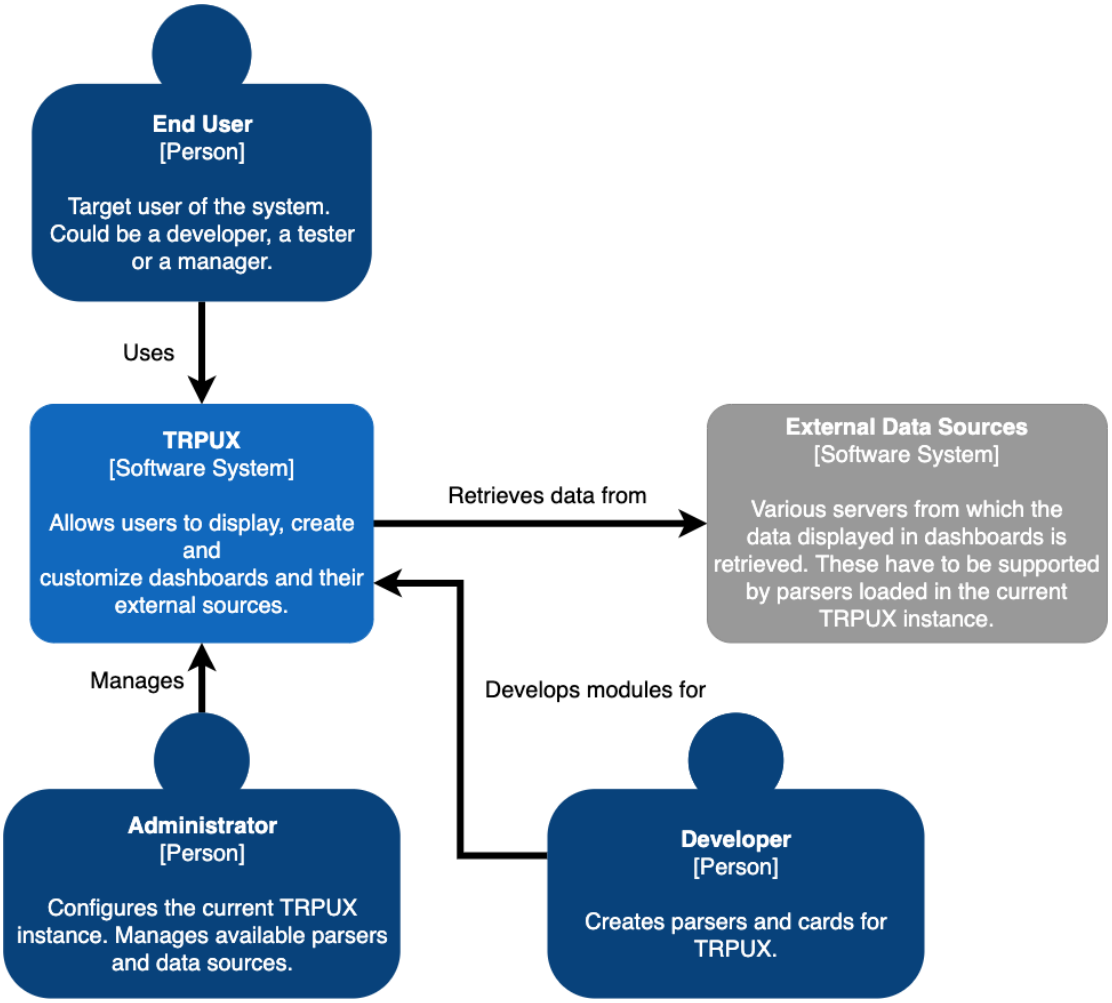

Figure 3.1. C4 context diagram of the system.

3.1.2 Container

As the container diagram [15] would be unreadable in a small size, it is included in the appendix B.

The container diagram builds upon the system diagram and specifies the building blocks of the system in detail. These containers are prominent parts of the system, e.g. web applications. Most of the major technology choices are highlighted in this diagram to give more specific technical details about the system.

Unlike the system diagram, the user interface is specified here as the Web Application and Single Page Application nodes. This change is only for illustration purposes and does not mean that the web application is the only interface that can be used in the system.

The Administrator and User roles from the system context diagram are merged into a single User role. Firstly, both roles should interact with the system through a web interface or any replacement, thus duplicating relationships. Secondly, this association allows for better clarity.

3.1.3 Component

As the component diagram [16] would be unreadable in a small size, it is included in the appendix C.

The highest level of detail is focused on the server container. As no actors are meant to interact with the server directly, save for direct CLI call to the REST API, they are omitted from this diagram. Instead, the single page application and external data sources are the only points of interaction for the system.

All of the static data, such as external data source URLs, dashboard configurations and user information, is stored in a database. The server interacts with the database via an object-relational mapping interface. Nevertheless, developers can choose their database hosting option supported by the selected ORM interface.

Communication with clients is done through discreet routers. Each router has a specific purpose, e.g. User Router serves requests concerning user accounts. A separated design allows for a more painless development and, more importantly, a straightforward testing strategy.

Some of the router paths are not designed to be accessible by unauthorised users or users without administrator privileges. A security component is thus used to prevent access to sensitive routes such as user lists or user account creation. Implementation depends on each developer.

3.2 Database

The design of the database depends on four pillars: Server, Source, Card, and User.

The Server represents an external data source, for example, Jenkins or TeamCity REST APIs, from which data is queried. Each server has different access constraints; some are freely available without authentication, while others need access tokens, credentials, or particular URL parameters. Thus the Server Argument table is provided to hold these values.

The second pillar is the Source table. This table represents a unique data source from an external server, for example, a Build on TeamCity. Sources are divided into distinct categories by their types which are then used to indicate compatibility with specific Card Types. Such a constraint prevents irrelevant matches, e.g. a single value being displayed in a matrix.

The third building block is the Card. As the name suggests, this table represents a single card on a dashboard. Each card entity stores its size and position in the Position table. The Connector table facilitates the connection between a card and its sources as an N to M relationship. As with the Server, a connection can be modified with arguments, thus creating the need for the Connector Argument table. Additionally, the Card Type table finalises the compatibility relationship with the Source table.

The last part is the Dashboard. This table ties all cards to a single place and is owned by a single user.

3.3 API

The API design follows the structure of the routers and functional requirements. All of the paths return HTTP status 500 on failure and error messages indicating what went wrong to leave users with proper feedback on their actions.

The system uses two categories of security: administrators and users. Authenticated users have the ability to create and manage dashboards with their cards and can change their usernames. Administrators are a subset of users and can manage data sources, users and dashboards. The exact implementation of authentication and authorisation is left to the developers, who can even extend this functionality.

As the complete description of the REST API in the OpenAPI format [17] is quite lengthy, it is included in the appendix.

3.4 Web Application

The default point of interaction bundled with the system is a Single Page Application [18]. This approach was chosen because a Single Page Application can be compiled into plain HTML, CSS and JavaScript files that can be served by the server providing the service and can be easily swapped for different implementation.

Chapter 4
Implementation

TRPUX is designed as a framework to be executed inside the Node.js runtime [19]. A developer can import the framework inside their project, configure necessary settings, implement their parsers and then execute their main file. The interactions with the database, route handling and communication with external APIs are handled by TRPUX. However, developers have the option to change these behaviours if needed.

4.1 Database

The TypeORM framework [20] facilitates database schema creation and subsequent communication. This implementation allows for a versatile database deployment as TypeORM is not hardwired to single database technology. Instead, the configuration is exposed to developers, and the choice between the available database options is up to them. Moreover, the database does not have to be hosted inside the same environment or even hosted at all. The TypeORM framework provides a wide range of supported drivers ranging from locally hosted SQLite instance or connection to a remote PostgreSQL database.

The schema is defined through TypeORM entities. This definition allows for a unified programming interface over various hosting options and removes the final development environment further from low-level database management. Furthermore, entity definitions can be extended with additional entities and relationships without modifying the default TRPUX ones.

4.2 Server

The server is implemented with the Express framework [21]. Express is a lightweight and minimalist framework for the creation of web applications and REST APIs. It is implemented as a thin layer of abstraction above Node.js with web application-specific features aimed at rapid development. Additionally, it boasts comprehensive community support, an extensive library of middleware and is easily extensible.

The server is intended to be imported as a library; it exports all the necessary functions and objects. These pertain primarily to the configuration of the server, the database entities and the custom parser setup. Finally, the two most important exports are the functions that create a server instance or start an instance directly.

The default routers are all implemented in separate files and are loaded automatically after the Express middleware is initialised. However, this behaviour can be modified if the server is instantiated through the *createInstance* function. The developer can then configure the Express application with additional routers, middleware and other options.

Authentication and authorisation are handled by the Passport library [22]. Passport is a modular and straightforward authentication middleware for Express that uses various authentication strategies. At the time of implementation, Passport boasted over 500 publicly available strategies in addition to having the ability to implement a custom solution. The default strategy used by the server is custom and only checks if a correct username is sent as a request parameter. It is not meant to be used in a production environment.

4.3 Web Application

The web application is a single page application written in React [23]. The framework was chosen because it is the most popular web framework today [24]. As most web developers today are familiar with React's concepts, the customisation of the web application should be relatively easy. Not only that, React's ecosystem contains more than 190 thousand packages [25], simplifying development even further.

The application uses the Mantine component library. It provides fully featured components with predefined design, React hooks and a notification system. The library is fully customisable and enabled the rapid development and design of the application. Furthermore, the developers can use the components to extend the default application without their additions feeling out of place.

Unfortunately, single page application React projects cannot be packaged as the Express server can be. This limitation comes from the fact that the React code is not executed directly and is instead packaged by a build tool; in the case of TRPUX, the tool is Snowpack [26]. This limitation means that an external configuration code cannot be executed, and thus the developers must edit the code directly.

The communication with the REST API is facilitated by custom calls through the Fetch API [27]. Each endpoint has its own fetch function, and some have corresponding hook functions to be easily incorporated into custom components.

4.4 Modularity

The main focus of TRPUX is its modularity. Third-party extendability is mainly achieved by two features: parsers and cards. While parsers are strictly on the server-side, card types have to be registered on the server while their corresponding components are implemented on the frontend side.

4.4.1 Parsers

Parsers are classes communicating with external data sources and parsing data for the display in supported cards. Parser support for data source types and card types is limited. Thus parsers only need to implement the combinations they support, giving the developers the freedom of choice. Furthermore, even though the parser interface was created with only a single server in mind, parsers can work with multiple server types if the developer chooses.

Parsers need to be declared as an implementation of the *BaseParser* interface and decorated with the *Parser* class decorator. Firstly, the interface implementation ensures the existence of the necessary attributes. Secondly, the class decorator detects the parser declaration at the start of execution, evaluates the declaration and adds it to the list of registered parsers.

The class decorator removes the need for complicated file management or manual registration by the developer. Each parser is automatically loaded and evaluated at the start of execution of each project automatically, no matter where in the file system the parser declaration resides. Unfortunately, the declaration is not detected simply by being declared, and the parser's declaration must be imported into a file being executed. For example, the parser is declared in a separate folder but is then imported into the file that creates and starts the TRPUX instance.

The most complex part of a parser is the *getSource* function retrieving data necessary for a selected card. As TypeScript discourages any as a return type, the function has to return a set type. Unfortunately, listing the types directly would make the addition of new card types complicated. Thus, a generic solution using an interface as a map of possible card maps was selected. An example implementation is shown in the code snippet below.

The *getSource* function receives a map key as a parameter that the function uses to determine what kind of data to retrieve. Secondly, the key indicates to TypeScript the type of the returned value while keeping within the type safety constraints.

```
async getSource<T extends keyof Interfaces.CardTypeMap>(
    type: T,
    source: Model.Source,
    args: Model.ConnectorArgument[]
): Promise<Interfaces.CardTypeMap[T]> {
    if (!this.supportedSources.includes(source.type.name)) {
        throw new Error(
            `Type "${source.type.name}" is not supported!`
        );
    }

    let result: any;
    switch (type) {
    case 'Line Graph':
        result = await this.getGraph(source, args);
        break;
    case 'Iframe':
        result = await this.getIframe(source, args);
        break;
    case 'Single Value':
        result = await this.getSingleValue(source, args);
        break;
    case 'Matrix':
        result = await this.getMatrix(source, args);
        break;
    default:
        throw new TypeError(
            `Card type "${type}" is not supported by this parser!`
        );
    }

    return result as Interfaces.CardTypeMap[T];
}
```

The above code snippet is used by the example Jenkins parser. The Jenkins module uses the REST API to retrieve the necessary data. And while the API provides most of the used information, the internal instance reports test results in XML files that require additional processing to adhere to the formats set out by the provided card types. As such, the Jenkins parser uses a separate technique to request, load and parse these files from the default communication channel.

A similar approach is envisioned for all of the third party parser modules. While a single module is mainly meant to provide data from a single source point, be it a REST API, database or a different kind of data source, a great degree of flexibility is afforded to the developers. The provided examples enhance the functionality of the single source in a meaningful way to provide the best user experience.

4.4.2 Cards

Cards are more easily extensible on the server-side than parsers since they are only data structure definitions without logic. By default, the card types line graph, matrix, iframe and single value are defined on the server. Any additional definitions have to be added through declaration merging [28].

In this process, two or more interfaces with the same name are declared within the same name. They only differ in their attributes which are then merged by TypeScript during execution. Thus, developers can declare their custom *CardTypeMap* with new card types and their names without destroying the default values.

This approach has double-edged consequences. As any of the declared attributes can be declared again in any of the subsequent declarations, the original attributes can be overridden. This means that developers can redefine default card types to a custom shape. On the other hand, this redeclaration can happen unintentionally and lead to hard-to-track-down issues for less experienced developers.

The implementation of the card components for the default web application is relatively straightforward as well. The component has to be wrapped in the *CustomCard* interface, which also contains the card type. The web application uses this flag to determine which component to render inside the card.

Due to the limitation with the configuration outlined in the Web Application section above, one more step is required to register the custom card component. This is done by adding the component inside the *CustomCard* wrapper to the list of cards inside the configuration class. Afterwards, the bundling system will include the declaration in the final build.

4.5 Default Modules

The system provides the two types of modules described in the section above and two customisable components: the authentication strategy and database. Both are provided through third-party libraries, Passport and TypeORM, respectively. The default database technology is SQLite, as it does not require any additional setup. The pre-provided authentication strategy uses simple username based authentication. These defaults are intended to provide a quick setup during development but should be re-evaluated for production.

Unlike the two components, TRPUX provides default card types and a parser to give developers a better understanding of modules. Depending on the requirements of the deployed instance, these modules can, however, be replaced or completely removed.

And, since the web interface is optional as well, a different rendering system can be used altogether.

By default, all of the cards contain the name of the card in the top left corner, unless listed otherwise.

4.5.1 Single Value Card

Single value is the simplest default display option. It's purpose is to display a one kind of information, either a number or a string. The most common use case for this card is to display a status of the selected build from continuous integration environment. Additionally, this card type could be used for a wide variety of information, for example the current temperature, stock price or the date for the next release of a product.

The single value card can have only one source assigned at a time. An example can be seen in the figure 4.1.

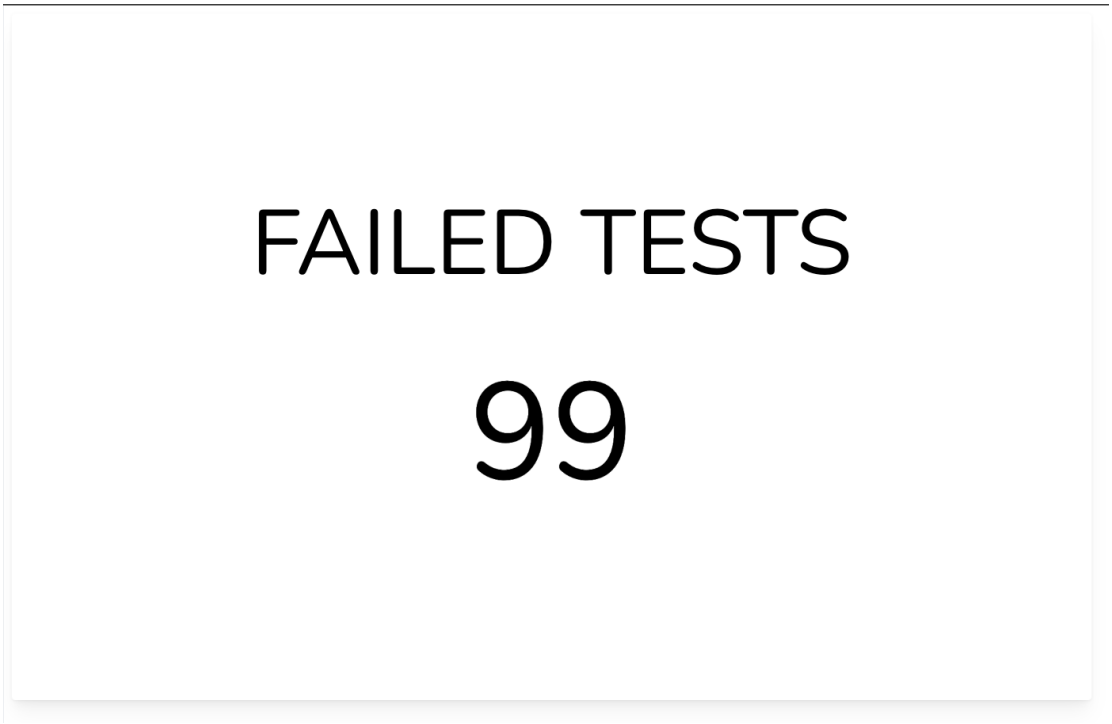

Figure 4.1. Single value card type example.

4.5.2 Line Graph Card

The line graph is meant as a representation of a selected value over a period. The period can be set as time, for example daily active users, or with an arbitrary indexing, e.g. last 20 builds. In the example below, the card displays the amount of failed tests for a specific build.

The line graph can be aggregated and displays a sum of the values per index. Only indexes with values for each source are displayed. Because the implementation uses the Chart.js [29] library, the title is displayed in the center above the graph through the included title functionality.

An example can be seen in the figure 4.2.

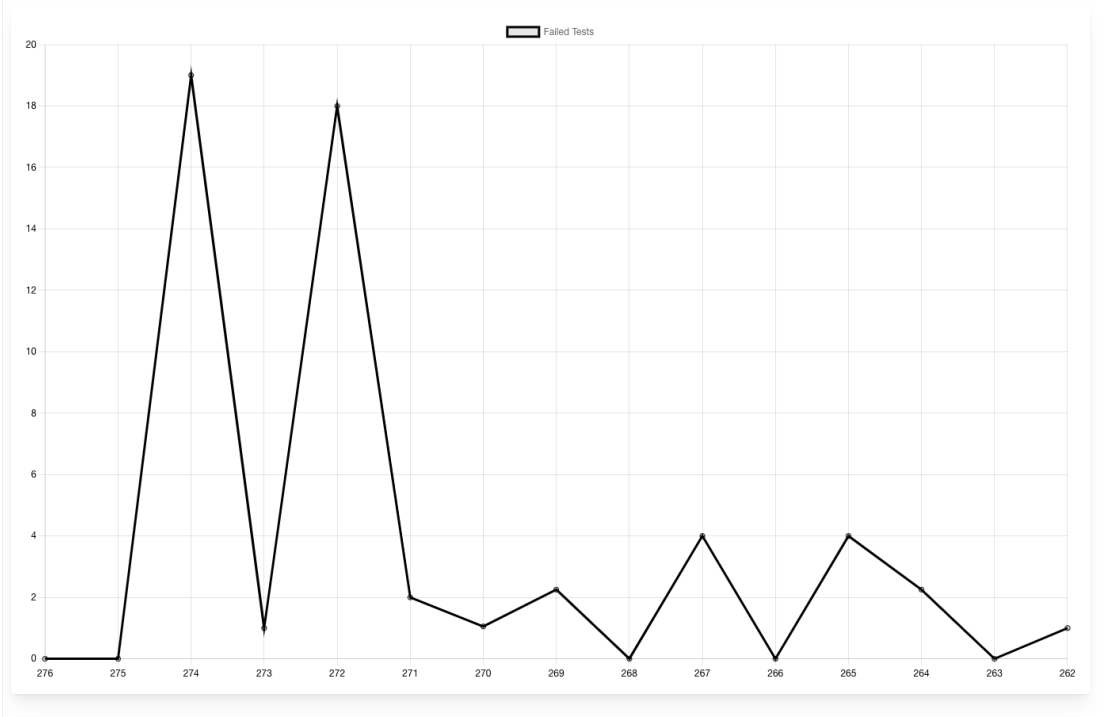

Figure 4.2. Line graph card type example.

4.5.3 Iframe Card

One of the requirements originating from Avast Software s.r.o. was to provide a way to display a HTML page in one of the cards. This is due to the fact that some test results are reported in such a format and their parsing would be too brittle for long term use, as the layout of the results changed quite often. In turn, the iframe card type was proposed to display these results directly.

As the iframe card only provides a way to display a foreign HTML document, it cannot be aggregated. An example can be seen in the figure 4.3.

4.5.4 Matrix Card

As the project was initially thought of as a test result visualiser, the matrix card type as one of the first ideas to be requested. The two dimensional matrix is meant to display results for executions of various software versions on various platforms. For example, at Avast Software s.r.o. the most common use case is testing of multiple editions of the Avast Antivirus on different versions of the Windows operating system.

The matrix card can be assigned only a single data source. This is mainly due to the fact that the card is meant to display a single execution run and the main goal is to display each unique combination.

An example can be seen in the figure 4.4.

4.5.5 Jenkins Parser

The Jenkins parser is an example of a module acquiring data from a REST API with an enhanced functionality. The modules uses the API to retrieve list of available build configurations, metadata about a particular build configuration and data about a particular build run. This data is parsed into data sources and card type data to be displayed and customised through the user interface.

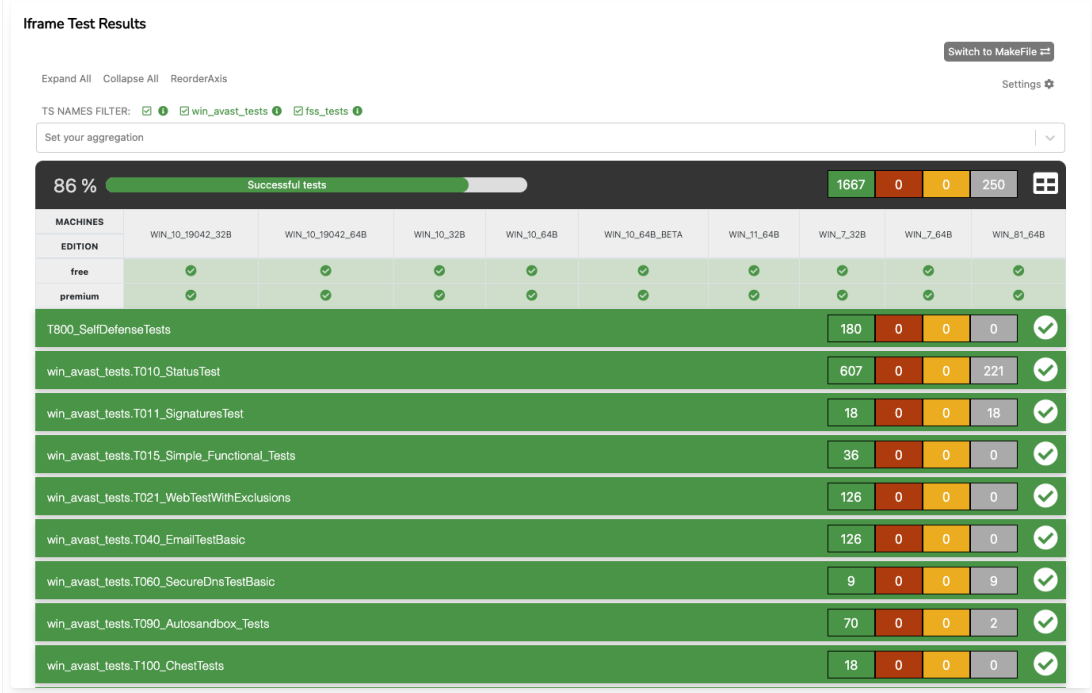

Figure 4.3. Iframe card type example.

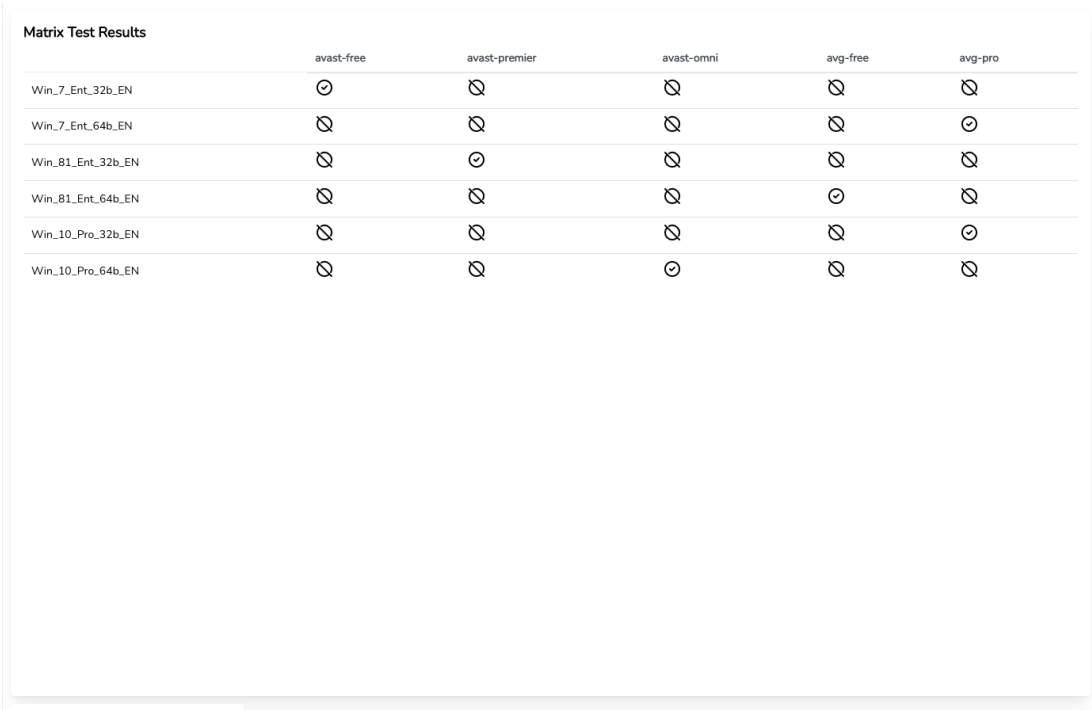

Figure 4.4. Matrix card type example.

Additionally, the parser is capable of downloading and parsing XML data results. The XML data format is used by an internal test runner of Avast Software s.r.o. and provides a unified interface for running tests in multiple environments, not only on

Jenkins. Due to this, the REST API can only be used to retrieve the URLs of the XML files and the files have to be requested and parsed separately.

A similar mechanism is used to retrieve the HTML pages. Fortunately, no download of the file or parsing is necessary and only the URL is passed onto the user interface in form of a URL.

4.5.6 TeamCity Parser

TeamCity parser was the second default parser that was to be provided. Unfortunately, the security policy at Avast did not permit the usage of the TeamCity REST API to test the parser. The request to use the service was not granted in time and, thus, the development time was spent on other parts of the system.

However, the parser was intended to use the REST API in almost the same way as the Jenkins one. However, as TeamCity provides more information than Jenkins, the customisation options would be considerably expanded.

4.6 Testing

4.6.1 Server

The server is tested with Jest [30] and Supertest [31] frameworks. Jest is used as the test runner with support for asynchronous testing, coverage collection, and mocking of functions or whole modules. In contrast, Supertest is used to test HTTP requests built upon the Superagent library. Such an approach ensures that the testing requests behave the same way as in a production scenario.

The test suite is fully automated with a test SQLite database running in memory. Importance has been placed upon integration tests instead of unit tests, as integration better evaluates the system's state. Additionally, unit tests can be too brittle and would make changes to the project more costly.

Each of the REST API endpoints is covered with tests for both successful and error-producing requests, where applicable. As these integration tests cover most of the functionality, the remaining coverage is focused on the parser registration process and Source Service, as these two parts contain the main logic of the system.

4.6.2 Web Application

The web application testing approach is similar to the server's. The limited testing of small parts of the system is ignored in favour of integration testing. But unlike on the server-side, the tests do not execute any function directly; instead, interacting exclusively with the website. This ensures that the results are identical to the end-user experience.

The chosen framework for these tests is Cypress [32]. In contrast to other frameworks, such as Selenium [33], Cypress is native to TypeScript and executes in the same run-loop. Additionally, the tests can run in different browsers and even in Electron [34], although with a significant caveat. The Electron distribution bundled with Cypress is not necessarily up-to-date, and since it runs headlessly, the results may differ from test runs with a proper GUI.

The tests cover all of the pages and significant functionality. The approach for each test is to be as natural as possible. This means going through the login process for each test if required and interacting with the application in the same manner as a user would. Such a process ensures the application is fully functional, even at the price of an extended test run time.

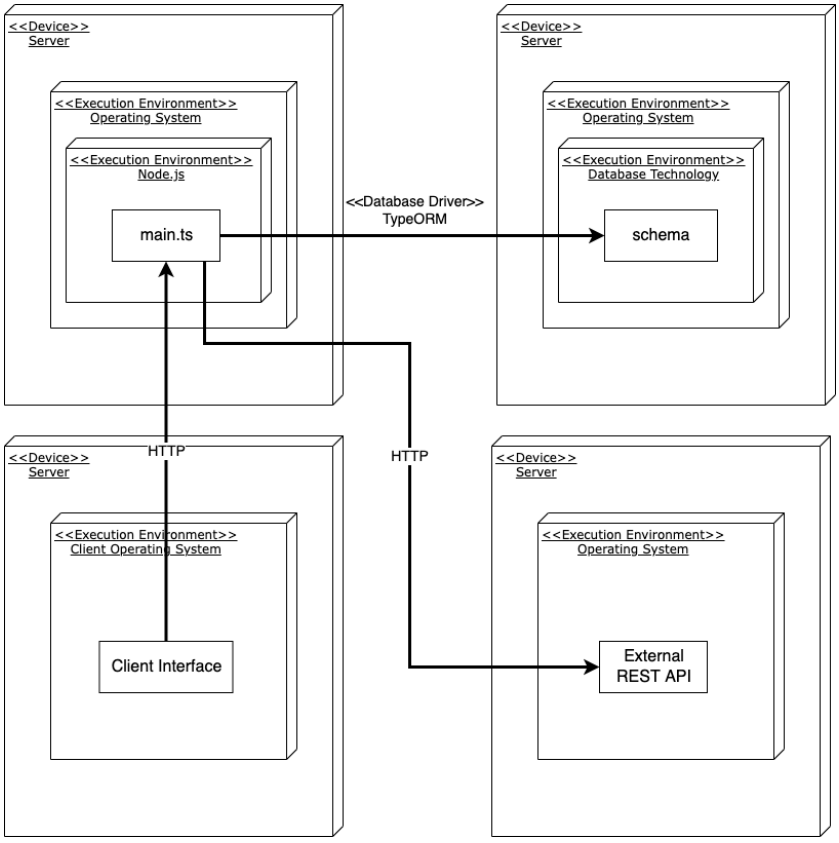

Figure 4.5. Deployment diagram of TRPUX.

4.7 Deployment

The server-side of the project is intended to be distributed as a package through the Node Package Manager. A version compiled to JavaScript coupled with TypeScript typings would be uploaded to the network and downloaded by the developers. They would then deploy their custom project inside a Node.js instance.

The only constraints would come from the Node.js version used during the development of TRPUX and TypeORM requirements. Some language features may not be available in versions of Node.js older than v16. TypeORM has its specific requirements, as well as does the selected database distribution.

The web application can be bundled with the server in the built form as pure HTML, CSS and minified JavaScript code. Due to React's limitations, any developers aiming to customise the code would have to download the source code and build the final distributed package themselves.

The figure 4.5 provides a visual representation of the deployment model.

Chapter 5
Evaluation

The project had to be evaluated based on the function and non-functional requirements and their categories. These divisions are not strictly separated, as, for example, most of the non-functional requirements have an impact on the developer experience. Nonetheless, such categorisation has been chosen as the best way for the evaluation of the project.

5.1 Client Evaluation

The client requirements were primarily concerned with functionality intended to be provided by the web application interface. Fortunately, as a more profound knowledge of the system is not required for the evaluation of any of them, the requirements could be assessed with both users involved in the analysis process and complete outsiders.

The client requirements were reviewed through usability testing [35] with five different users. Such an approach allows for a close simulation of real-world usage. As the tests are constructed as scenarios, they cannot test the requirements directly. Instead, the feedback was structured into findings categorised by their priority.

5.1.1 Testing Scenario

The following scenario was used:

1. Look at the homepage
2. Find the login page
3. Switch to the sign-up page
4. Create a new account
5. Log in
6. Create a new dashboard
7. Create a new line graph card with sources
8. Resize the card
9. Rename the dashboard

5.1.2 Findings

The findings from the usability test were divided into three categories: high, medium and low priority. The first category of findings indicates a significant flaw preventing the usage of the system. The medium priority represents a non-blocking problem, but its removal would improve the user experience. The last category contains manageable issues without the immediate need for repair.

An example of the dashboard interface can be seen in the figure 5.1. The following issues were identified:

1. Card Customisation
   - Priority: Medium
   - The card customisation was the most significant pain point for every tester. The testers were confused by the interface and were unsure how to proceed with data source customisation. The prime example being setup of line graph with multiple sources. The users were not sure if the card will render separate line for each source or combine the sources into a single line.
   - Recommendation: Improve the labelling and interface of card customisation.
2. Unclear Icon Navigation
   - Priority: Low
   - The navigation through icons is not clear enough at first glance. Fortunately, the icons provide tooltips on mouse hover and do not block usability.
   - Recommendation: Use clearer icons.
3. Dashboard Customisation
   - Priority: Low
   - The dashboard options are hidden behind a sliding mechanism triggered by an icon button. However, just like the icon navigation, this problem is minimised by the provided tooltip.
   - Recommendation: Make the dashboard customisation more easily discoverable.

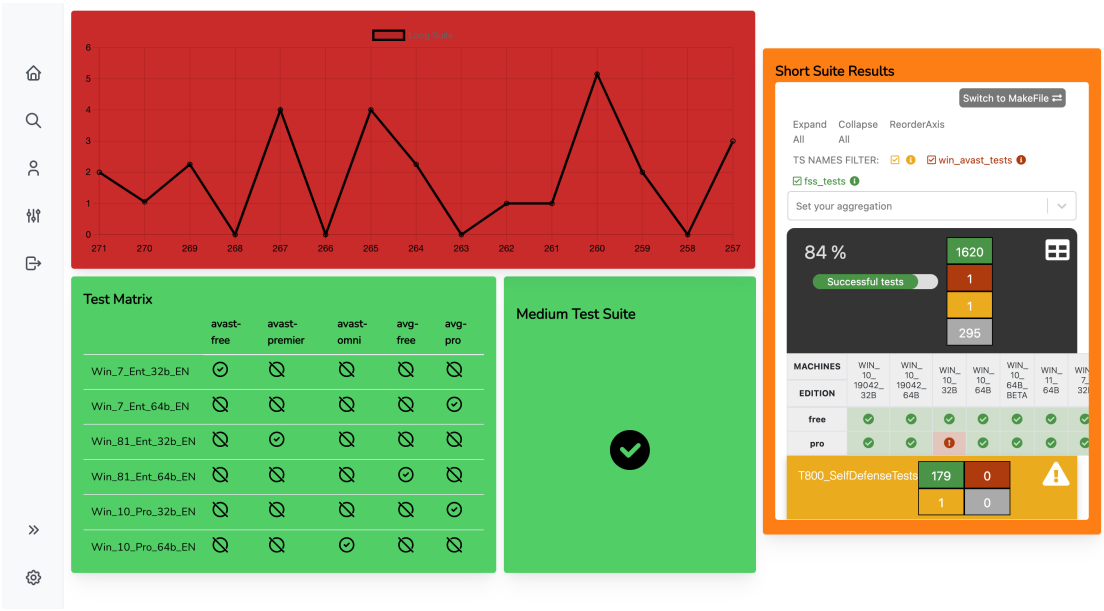

Figure 5.1. Dashboard with the default cards

5.1.3 Conclusion

The confusing configuration options were highlighted as the major letdown of the whole project for the end-users. Not only were the Jenkins setup options improperly labelled, but the default cards also proved to be confusing as well. The line graph, for example, gave some testers the impression that a separate line would be drawn for each of the sources assigned to the card. Contrarily, the default line graph aggregates all of the sources into a single line.

The card feedback proved to be critical for the future of the project. While the goals of the requirements were met, the current iteration of the web application interface would need to be updated before being released to a broader audience. Additionally, better documentation for the card options or help from a user experience designer would be required.

5.2 Administrator Evaluation

Just like the client requirements, the administrator evaluation was heavily targeted toward the user interface. The same set of testers was used, even though some context was provided to users without prior knowledge of the project.

5.2.1 Testing Scenario

1. Find the login page
2. Log in with an administrator account
3. Find the administration page
4. Create a new server
5. Refresh sources
6. Assign a dashboard to a new user
7. Rename added server

5.2.2 Findings

Just like the client evaluation, three levels of priority were used:

1. Refresh Sources
   - Priority: Low
   - The less experienced users were not sure what the Refresh Sources button does.
   - Recommendation: Add an explanation of the process to the interface.

The figure 5.2 showcases the administration page of the web user interface.

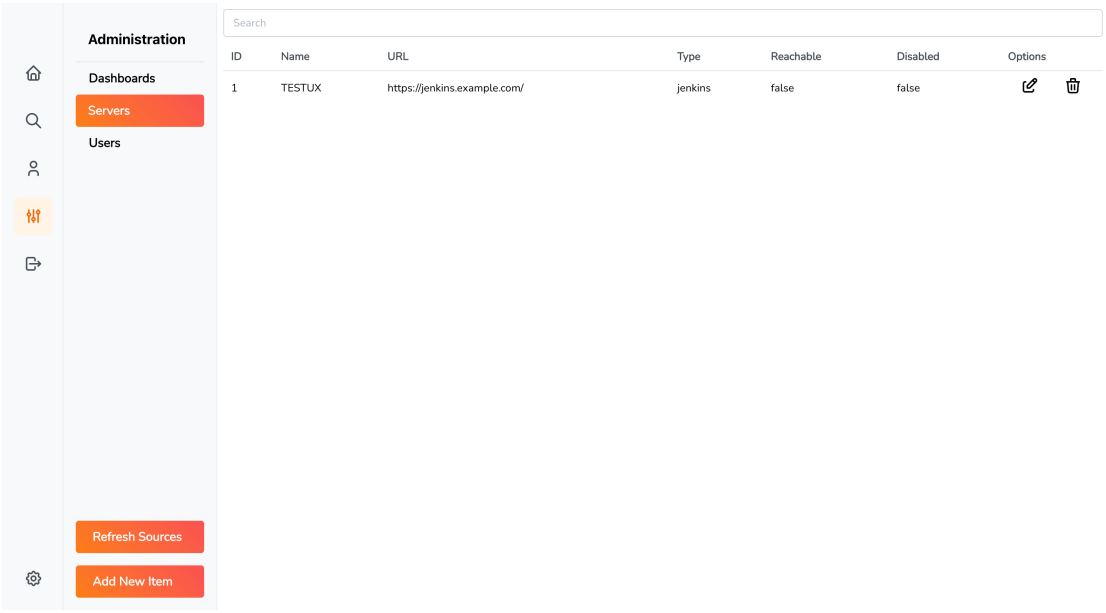

Figure 5.2. Administration page of the web interface

5.3 Developer Evaluation

The developer evaluation process was vastly different from the previous two processes. The developers are not meant to be interacting with the user interface, and they are instead meant to be working only with the provided codebase. Only users with experience in development with TypeScript were asked to give feedback on this part of the system.

5.3.1 Parser Creation

From the developer's perspective, the parser modules are the most critical part of their work with the system. As TRPUX, by default, is not able to retrieve any data by itself, this functionality has to be instead provided by the developers.

The parser creation process was praised for the way the parsers are meant to be implemented. The Parser interface was pointed out as the best possible way for the developers to adhere to the necessary parser requirements. On the other hand, the provided examples of parsers were said to be either bit too simple or too complex, in the case of the provided Jenkins parser, to follow and build upon with their code.

The main negative point of the parser examples was concerned with the retyping of return values inside the parser. The current solution, as showcased in the Jenkins parser, uses retyping to and from the "any" type was considered convoluted and not totally in the spirit of TypeScript guidelines. However, as no better alternative was proposed, the current solution stayed.

The other negative feedback was attributed to the mechanism through which a parser is registered to the system. Class decorators are not a widely known concept in the TypeScript ecosystem, so their functionality had to be at least partially explained for some testers to understand. But the more significant issue was that the registration was not fully automated, which means that the developer had to import the declaration into a directly executed code file.

Fortunately, these issues were easily overcome by improving the documentation. And as such, the parser creation system has been considered a success.

5.3.2 Card Type Creation

Card type creation is much easier to understand, as declaration merging is much more prevalent than decorators. Additionally, all of the work concerning the retyping and type safety is already provided by the system, putting even less work on the developers.

The only issue that surfaced during the evaluation of card type creation was getting enough inspiration to construct a new card type. Most testers were short-sighted by the already provided card types and could not come up with new and unique ideas.

5.3.3 Card Creation

Following the card type creation process, the creation of individual cards to be displayed was relatively easy. Testers only wrote possible rendering options for their new card types.

One major hiccup was caused by the registration of the new card type. As React is, unlike the server code, bundled by a build tool, the process is quite convoluted in comparison. Unfortunately, a proper remedy was not immediately available, and the system was left in its current state.

5.3.4 Authentication Management

Authentication management is provided fully through Passport with the corresponding functions exposed by the server in the configuration. Any of the result issues were accredited to the functionality of Passport and deemed as not fixable.

5.4 Non-Functional Evaluation

The non-functional requirements proved to be much harder to evaluate than their functional counterparts. Not only because of their more abstract nature but also due to the personal biases of the evaluators. Nevertheless, most of the requirements were considered a success.

Chapter 6
Conclusion

This project aimed to analyse, design and implement a modular system for data display. The implementation had to follow both functional and non-functional requirements set by the analysis and design stages.

The project has been a great learning experience for me. Starting with the analysis stage, the compilation of requirements from all relevant stakeholders inside Avast Software s.r.o. has not been straightforward and sometimes proved to be even contradictory. Thankfully, there was enough time to analyse the data correctly and filter out the essential features.

Most of the requirements have already been implemented in other, mostly separate, works, thus giving the project a reasonable frame of reference. Even then, the work on the project has not been straightforward. The hardest complications came unsurprisingly from the modular system, especially on the web application side combining the additional modules. With all that said, the goal of this project has been successfully reached.

The implementation itself was the most complex stage of all. My proficiency in TypeScript was on a good level before, but even then, I have only continued learning during the coding process. The work on the modular aspect of the system has given me a new perspective on the typing system of TypeScript and taught me valuable lessons that I can utilise in future projects in general.

The main goal of the project, to create a modular system capable of being extended on an as-needed basis, has been achieved. The current system provides ways to be expanded with new data sources and rendered card types without disruption to the already existing components. As such, the system can accommodate requirements that were not implemented during the project or were discovered afterwards.

However, TRPUX can still be upgraded to provide the best user experience, even though it is currently deployed only internally inside Avast Software s.r.o. The possible upgrades are:

- Automated continuous deployment of newest features to the internal instance.
- Implementation of additional modules based on needs of various teams inside the company.
- Application of remedies for the issues discovered during usability testing.

References

[1] Duvall, Paul M, Steve Matyas, and Andrew Glover. *Continuous integration : improving software quality and reducing risk*. Addison-Wesley, 2007.

[2] Rozentals, Nathan. *MASTERING TYPESCRIPT - FOURTH EDITION : build enterprise-ready, modular web applications... using typescript 4 and modern frameworks*. Packt Publishing Limited, 2021.

[3] s.r.o., JetBrains. *TeamCity Dashboard*. [cit. 2022-05-12]. Available from `https://www.jetbrains.com/help/hub/Dashboard.html`.

[4] Atlassian. *What is a Jira dashboard? | Jira Software Cloud*. [cit. 2022-05-09]. Available from `https://support.atlassian.com/jira-software-cloud/docs/what-is-a-jira-dashboard/`.

[5] Community, Jenkins. *Blue Ocean Dashboard*. [cit. 2022-05-09]. Available from `https://www.jenkins.io/doc/book/blueocean/dashboard/#dashboard`.

[6] B.V., Elasticsearch. *What is Kibana?* [cit. 2022-05-12]. Available from `https://www.elastic.co/what-is/kibana`.

[7] B.V., Elasticsearch. *What is Elasticsearch | Elastic*. [cit. 2022-05-12]. Available from `https://www.elastic.co/what-is/elasticsearch`.

[8] Labs, Grafana. *Introduction to Grafana*. [cit. 2022-05-09]. Available from `https://grafana.com/docs/grafana/latest/introduction/`.

[9] Labs, Grafana. *Auth Proxy*. [cit. 2022-05-09]. Available from `https://grafana.com/docs/grafana/latest/auth/auth-proxy/`.

[10] Benitte, Raphaël. *Mozaïk | Mozaïk*. [cit. 2022-05-12]. Available from `http://mozaik.rocks/`.

[11] Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Ph.D. Thesis.

[12] Gupta, Lokesh. *What Is REST – Learn to Create Timeless REST APIs*. [cit. 2022-05-12]. Available from `https://restfulapi.net/`.

[13] Brown, Simon. *The C4 model for visualising software architecture*. Available from `https://c4model.com/`.

[14] Brown, Simon. *System Context Diagram*. [cit. 2022-05-09]. Available from `https://c4model.com/#SystemContextDiagram`.

[15] Brown, Simon. *Container Diagram*. [cit. 2022-05-09]. Available from `https://c4model.com/#ContainerDiagram`.

[16] Brown, Simon. *Component Diagram*. [cit. 2022-05-09]. Available from `https://c4model.com/#ComponentDiagram`.

[17] Software, SmartBear. *OpenAPI Specification - Version 3.0.3 | Swagger*. [cit. 2022-05-12]. Available from `https://swagger.io/specification/`.

[18] Smith, Steve, Tarun Jain, David Pine, Youssef Victor, Genevieve Warren, Pablo Marcano, and Maira Wenzel. *Choose between traditional web apps and*

single page apps. [cit. 2022-05-12]. Available from https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps.

[19] Foundation, Node.js. *About | Node.js*. [cit. 2022-05-12]. Available from https://nodejs.org/en/about/.

[20] Community, TypeORM. *TypeORM - Amazing ORM for TypeScript and JavaScript (ES7, ES6, ES5). Supports MySQL, PostgreSQL, MariaDB, SQLite, MS SQL Server, Oracle, WebSQL databases. Works in NodeJS, Browser, Ionic, Cordova and Electron platforms*. [cit. 2022-05-12]. Available from https://typeorm.io/.

[21] Foundation, OpenJS. *Express - Node.js web application framework*. [cit. 2022-05-12]. Available from https://expressjs.com/.

[22] Hanson, Jared. *Passport.js*. [cit. 2022-05-12]. Available from https://www.passportjs.org/.

[23] Meta Platforms, Inc. *Main Concepts of React*. [cit. 2022-05-12]. Available from https://reactjs.org/docs/hello-world.html.

[24] Statista. *Most used web frameworks among developers globally 2020 | Statista*. [cit. 2022-05-12]. Available from https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/.

[25] npm, Inc. *List of React Related Packages on NPM*. [cit. 2022-05-09]. Available from https://www.npmjs.com/search?q=react.

[26] Schott, Fred. *Snowpack*. [cit. 2022-05-12]. Available from https://www.snowpack.dev/.

[27] Contributors, MDN. *Fetch API - Web APIs | MDN*. [cit. 2022-05-12]. Available from https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[28] Microsoft. *Documentation - Declaration Merging*. [cit. 2022-05-09]. Available from https://www.typescriptlang.org/docs/handbook/declaration-merging.html.

[29] Contributors, Chart.js. *Chart.js | Open source HTML5 Charts for your website*. [cit. 2022-05-20]. Available from https://www.chartjs.org/.

[30] Facebook, Inc. *Jest · Delightful JavaScript Testing*. [cit. 2022-05-12]. Available from https://jestjs.io/.

[31] visionmedia. *visionmedia/supertest*. [cit. 2022-05-12]. Available from https://github.com/visionmedia/supertest.

[32] Cypress.io. *End to End Testing Framework*. [cit. 2022-05-12]. Available from https://www.cypress.io/how-it-works/.

[33] Conservancy, Software Freedom. *About Selenium*. [cit. 2022-05-12]. Available from https://www.selenium.dev/about/.

[34] Foundation, OpenJS, and Electron Contributors. *Electron | Build cross-platform desktop apps with JavaScript, HTML, and CSS.*. [cit. 2022-05-12]. Available from https://www.electronjs.org/.

[35] Moran, Kate. *Usability Testing 101*. Available from https://www.nngroup.com/articles/usability-testing-101/.

Appendix A
Glossary

- API — An application programming interface is a connection between computers or between computer programs. It is a type of software interface, offering a service to other pieces of software.
- CSS — Cascading Style Sheets is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.
- GUI — The graphical user interface is a form of user interface that allows users to interact with electronic devices through graphical icons and audio indicator such as primary notation, instead of text-based UIs, typed command labels or text navigation.
- HTML — The HyperText Markup Language or HTML is the standard markup language for documents designed to be displayed in a web browser. It can be assisted by technologies such as Cascading Style Sheets and scripting languages such as JavaScript.
- HTTP — The Hypertext Transfer Protocol is an application layer protocol in the Internet protocol suite model for distributed, collaborative, hypermedia information systems.
- LDAP — The Lightweight Directory Access Protocol is an open, vendor-neutral, industry standard application protocol for accessing and maintaining distributed directory information services over an Internet Protocol network.
- ORM — Object–relational mapping in computer science is a programming technique for converting data between type systems using object-oriented programming languages.
- REST — Representational state transfer is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of an Internet-scale distributed hypermedia system, such as the Web, should behave.
- URL — A Uniform Resource Locator (URL), colloquially termed a web address, is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it.
- XML — Extensible Markup Language is a markup language and file format for storing, transmitting, and reconstructing arbitrary data. It defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

Appendix B
Container Diagram

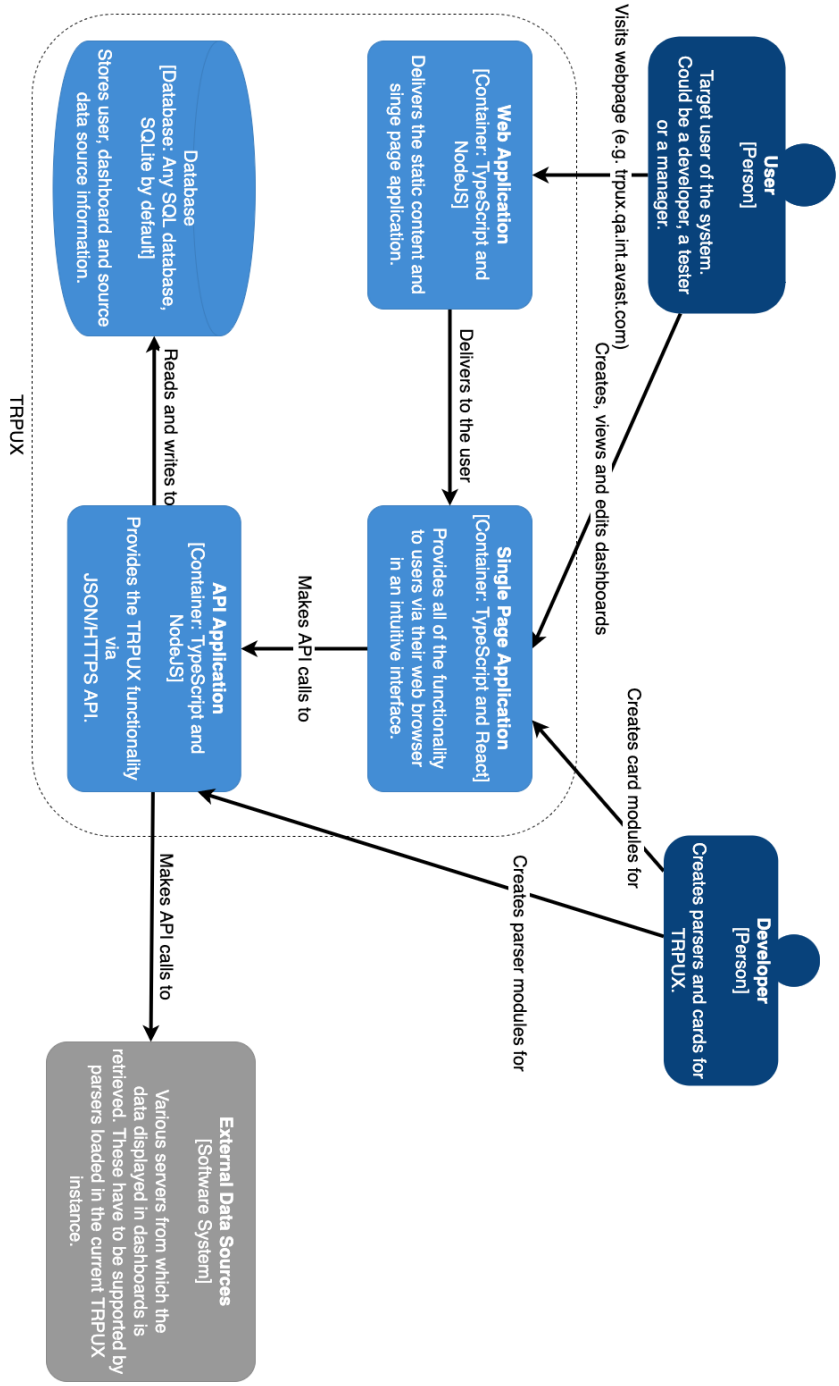

Figure B.1. C4 container diagram of TRPUX.

Appendix C
Component Diagram

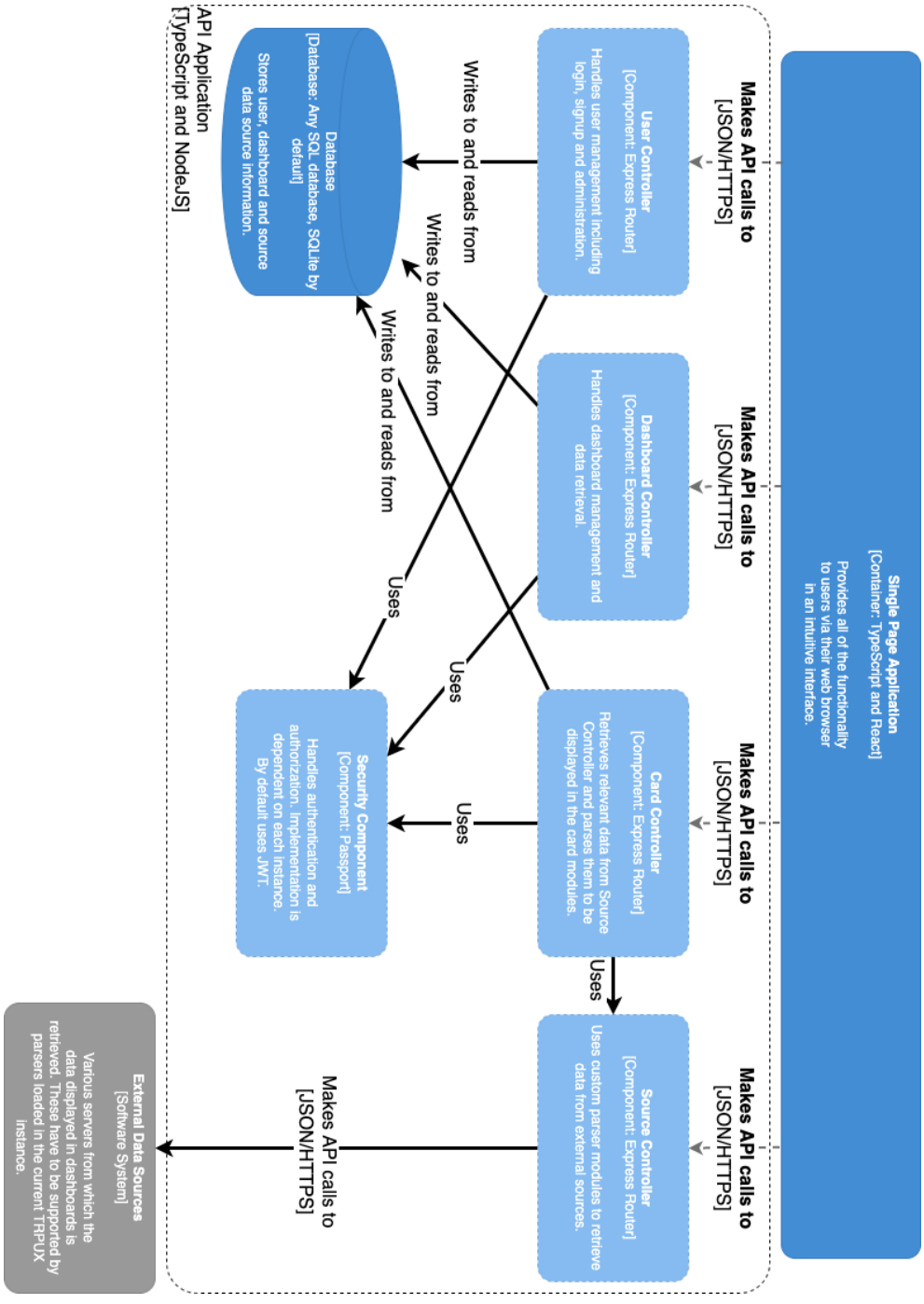

Figure C.2. C4 component diagram of the TRPUX server.

Appendix D
User Interface Preview

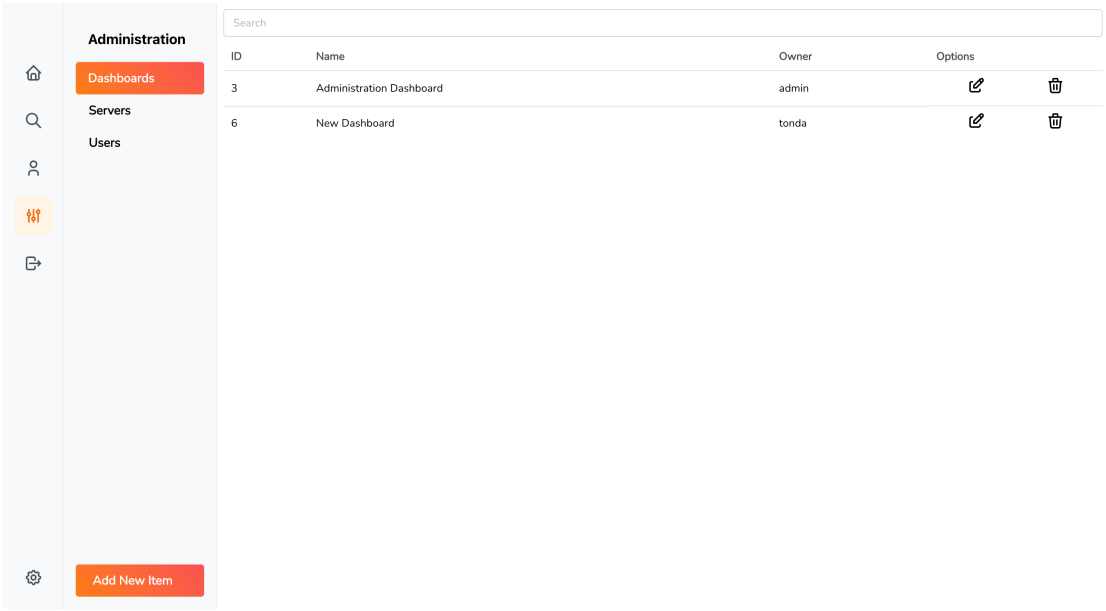

Figure D.3. Dashboard administration page.

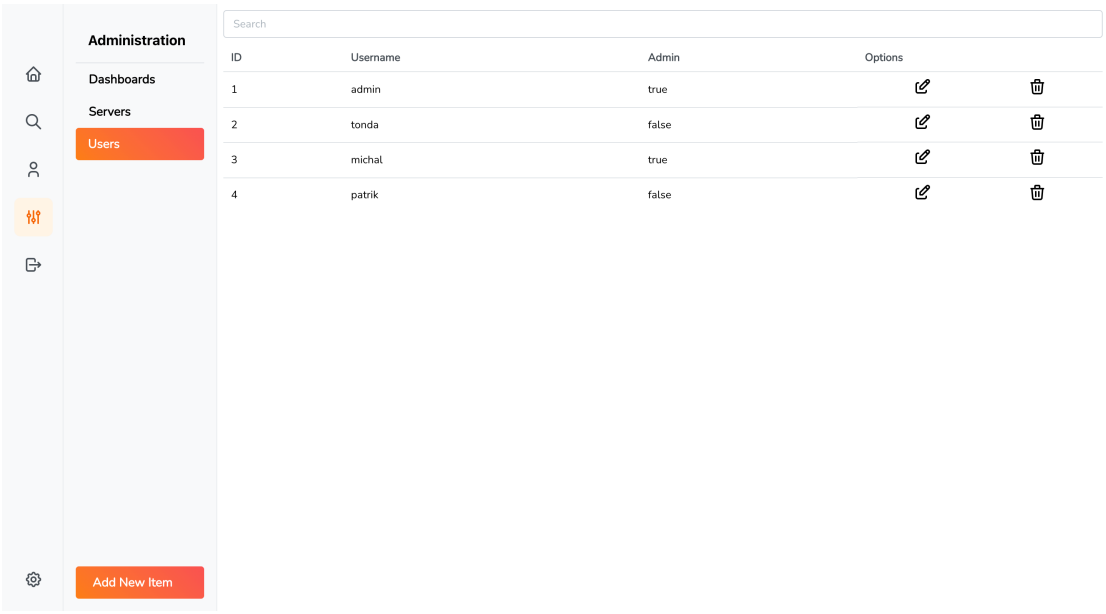

Figure D.4. User administration page.

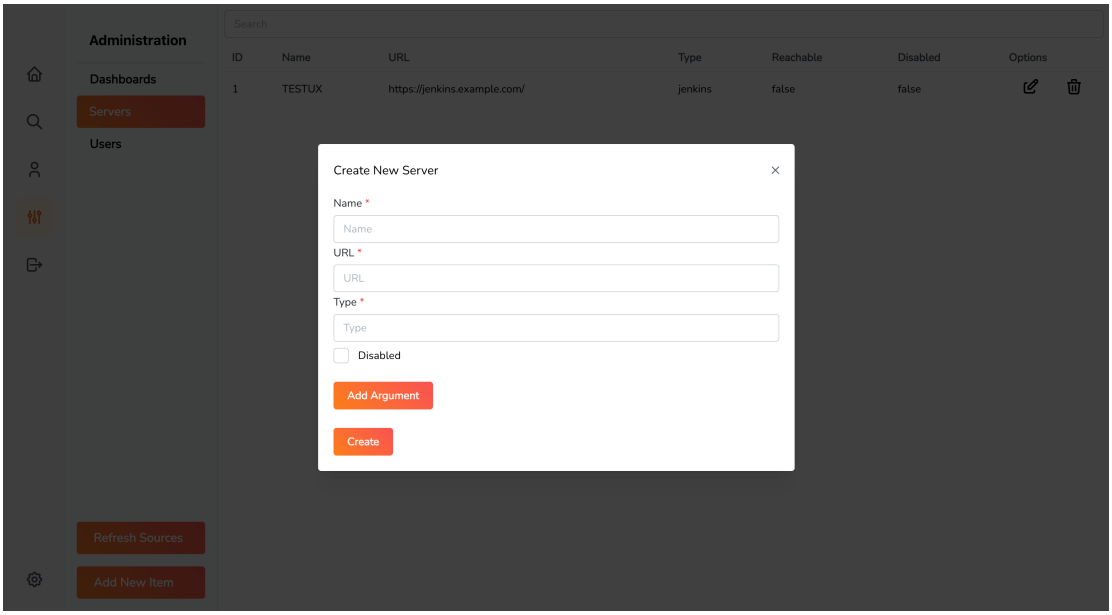

Figure D.5. Server creation dialog.

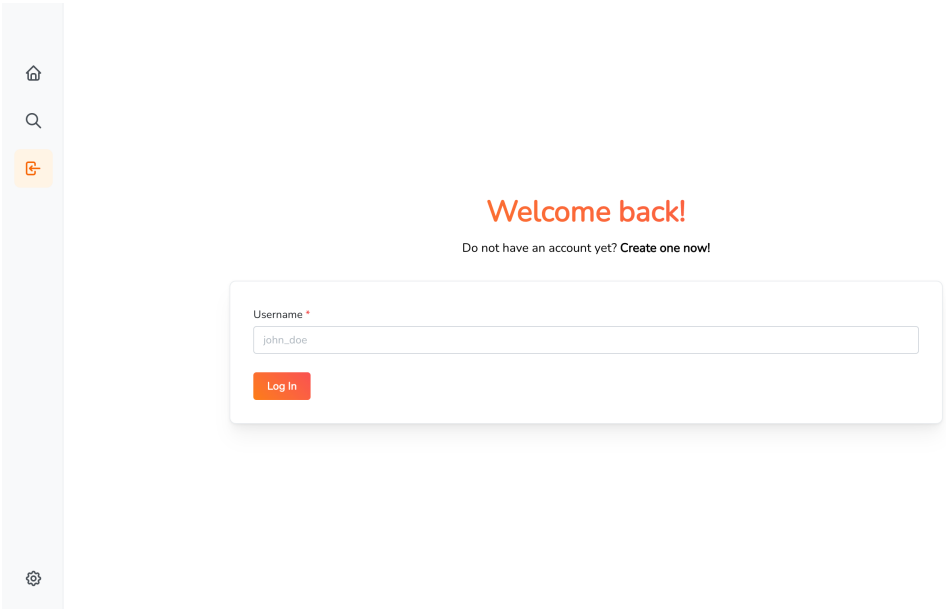

Figure D.6. Default login screen.

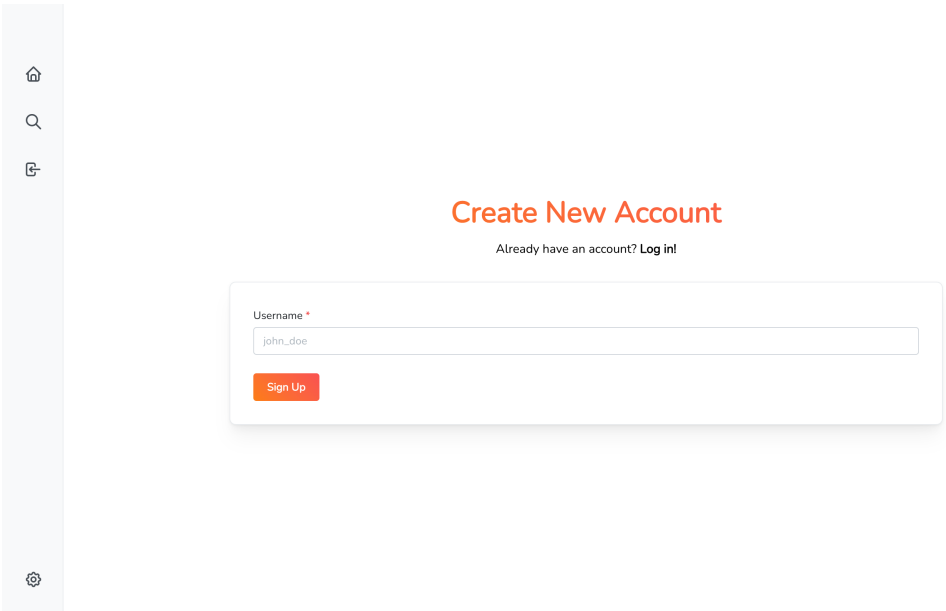

Figure D.7. Default sign-up screen.

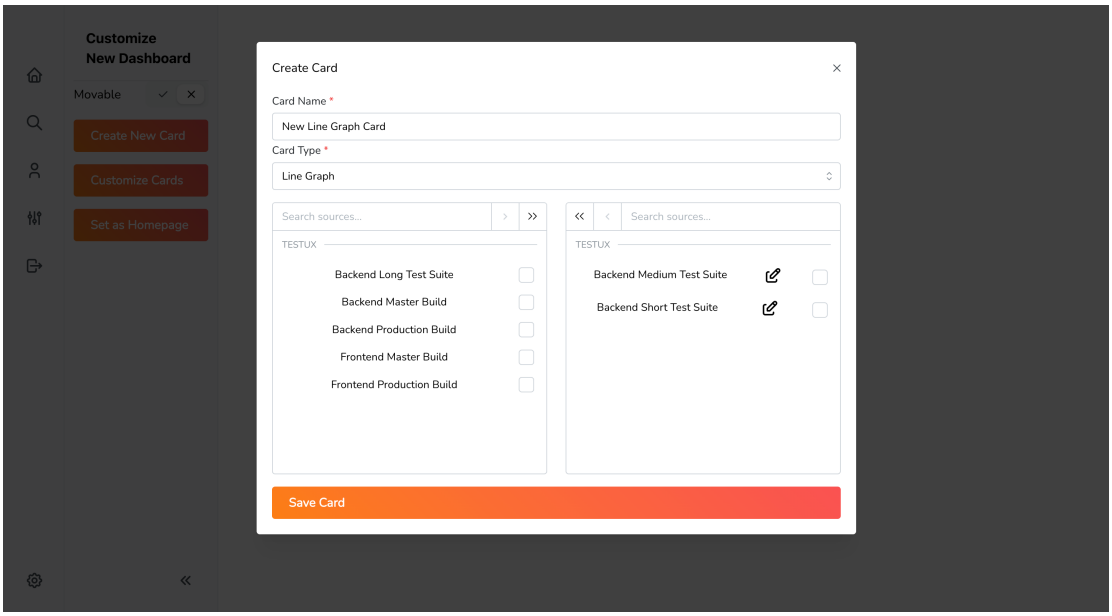

Figure D.8. Card creation setup.

Appendix E
User Manual

- Requirements
  - Operating System: Windows, MacOS or Linux
  - Node.js runtime v16 or newer
  - Available port to provide the REST API on
- Server Setup
  - Create a new Node.js project
  - Add the *@trpux/server* package as dependency
  - Import the *createServer* function
    - Optionally pass custom configuration as a argument to the function
  - Start the project either through the ts-node package or compile to JavaScript and execute the resulting build

Appendix F
Attached Media Contents

- The file *bachelor.pdf* is the digital version of this work
- The directory *source_code* contains the source code of the project
- The directory *openapi* contains the OpenAPI specification of the REST API and it's visualisation.