THALES: Automated Test Dashboard for a CMS

Design Project

2024-2A, Group 14

Hanno Remmelg s2960540
Mihai Buliga s3015424
Teodor Pintilie s2920344
Rudolfs Neija s2975157
Volodymyr Lysenko s2880911
Sviatoslav Demchuk s2889811

M11 Design Project TCS (2024-2A)
BSc Technical Computer Science
The University of Twente
Enschede, The Netherlands
March 2025

Contents

1 Introduction **2**
- 1.1 Context . . . 2
- 1.2 Vocabulary . . . 2
- 1.3 Requirements . . . 3
  - 1.3.1 Existing Dashboard . . . 3
  - 1.3.2 Identified Problems . . . 3
  - 1.3.3 Initial Requirements . . . 4
  - 1.3.4 Initial Mock-Up . . . 5
  - 1.3.5 Refined Requirements . . . 6
  - 1.3.6 Extra Requirements . . . 8

2 Design **10**
- 2.1 Global Design Choices . . . 10
  - 2.1.1 Backend – Spring Boot Application . . . 10
  - 2.1.2 Frontend – React + Vite . . . 11
  - 2.1.3 Interactions and Technologies . . . 11
- 2.2 User-Centered and Value-Sensitive Design . . . 11
- 2.3 Dashboard Overview . . . 12
  - 2.3.1 Teams Page . . . 12
  - 2.3.2 Functional Products Page . . . 12
  - 2.3.3 Test Cases Page . . . 13
  - 2.3.4 Detailed Test Case Page . . . 15
- 2.4 API . . . 18
- 2.5 Database . . . 19
  - 2.5.1 Design Tool . . . 19
  - 2.5.2 Schema Evolution . . . 19
- 2.6 Important Design Decisions . . . 23
  - 2.6.1 Breadcrumbs . . . 23
  - 2.6.2 Comments . . . 24
  - 2.6.3 Settings & Quick Filters . . . 25

3 Implementation **26**
- 3.1 Technology & Tools . . . 26
  - 3.1.1 Tools . . . 26
  - 3.1.2 Languages & Frameworks . . . 27
- 3.2 Frontend . . . 27
  - 3.2.1 Project Structure . . . 27
  - 3.2.2 Data Handling . . . 28
- 3.3 Backend . . . 28
  - 3.3.1 Project Structure . . . 28
  - 3.3.2 XML Parsing . . . 29
  - 3.3.3 Logging . . . 30
  - 3.3.4 Documentation . . . 30

4 Testing **32**
4.1 Test Plan 32
4.2 User Testing 32
4.3 Unit Testing 32
4.4 Continuous API Testing 33

5 Discussion & Conclusion **34**
5.1 Discussion 34
5.1.1 Meetings 34
5.2 Distribution 35
5.3 Conclusion 35

1 Introduction

1.1 Context

At Thales Netherlands, every day multiple software teams finish their work once evening strikes. The next morning, automated tests have already been run, and developers expect to review the results efficiently. Currently, their system for viewing the test results, a test regression dashboard, is outdated, hard to understand, and inefficient. Each team is responsible for multiple projects, called FP's (functional products), and each FP can contain tens of tests. Grasping this large amount of information fast is necessary for a good user experience and ultimately can save a great deal of time.

The objective of this project is to replace the test regression dashboard for TACTICOS, a combat management system. This dashboard will help over 20 development teams, allowing them to quickly review whether recent changes have introduced any bugs. Given that automated tests take hours to complete, making sure that data is displayed properly and is easy to interact with is essential for workflow efficiency. The project involves building a new dashboard application from the ground up, incorporating key features such as filtering, sorting, notifications, summarization, and trend analysis.

1.2 Vocabulary

Here we explain the common terms used in this report:

- Existing dashboard - the old system that they currently use to look at the test results. Our project would ideally replace this.
- Quick filter – a saved combination of filter values that can be applied with a single click. Useful for frequently used filtering setups.
- RFT - one of two test frameworks Thales uses to run the tests.
- TAF - one of two test frameworks Thales uses to run the tests.
- Team - a single team consisting of a couple developers. Working on FP's.
- Functional product (FP) - a project that a team works on. Can contain many containers.
- Container - a collection of test cases that have been ran together. Different containers in a FP might have been ran on different computer and at different times.
- Test case - a single test, testing a piece of software. Can contain many scenarios.
- Scenario - a part of a test case.

- Flakyness - a flag showing if the test has been switching between failing and succeeding frequently. In the final dashboard, can be manually toggled.

1.3 Requirements

Although the supervisors at Thales were well aware of the limitations and current state of the existing dashboard, they did not have a clear vision of what the new application should look like or which features would provide the most value. Since this was not a matter of extending a mock-up but rather creating a completely new solution, much of the project involved collaboratively exploring what the teams truly needed.

This meant identifying which information was most relevant, how it should be displayed, and what kinds of interactions would support the developers' workflow. At the start of the project, we received initial requirements, but soon after, they were refined. In the end, even these refined requirements were not sufficient and had to be modified; furthermore, a lot of new requirements had to be added. The requirements were shaped mainly through our meetings with different team members.

This evolving process ensured that the final product was closely aligned with the developers' real needs, rather than assumptions, ultimately aiming to greatly enhance the user experience.

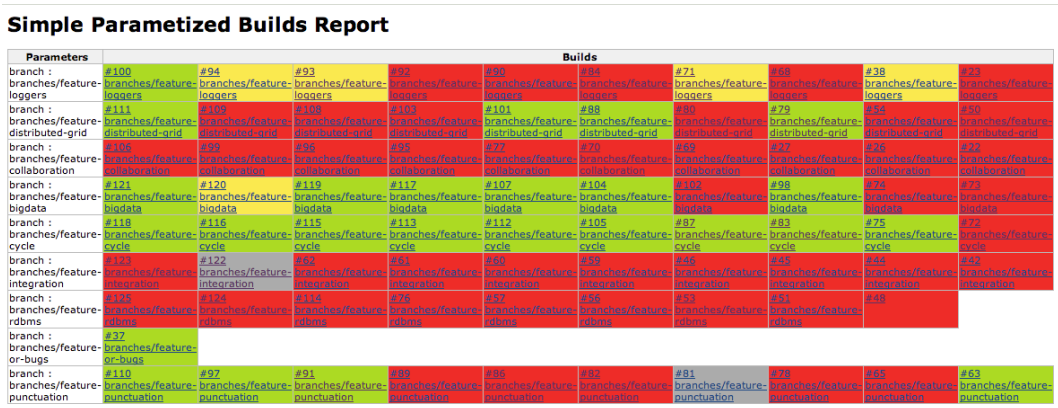
1.3.1 Existing Dashboard

The existing dashboard was a simple table view, with rows being the FP's and columns the days. Each cell contained clickable names of test cases that had failed for that FP. Due to company policy, we cannot provide a picture of the existing dashboard in use, but the given picture shares the overall aesthetic of the existing dashboard. As in the provided graphic, Figure 1, information is cluttered but nicely color-coded. The existing dashboard had 2 weeks of historical data at all times, therefore around 14 columns and an endlessly scrollable list of FP's in the form of rows.

1.3.2 Identified Problems

The main problems identified with the existing dashboard by Thales were the following:

- Lack of information in a single view.
- Overload of redundant information (Showing all history does not provide a good overview of the current situation).
- No (smart) analysis on results.



Simple Parametized Builds Report

Parameters	Builds									
branch : branches/feature-loggers	#100 branches/feature-loggers	#94 branches/feature-loggers	#93 branches/feature-loggers	#92 branches/feature-loggers	#90 branches/feature-loggers	#84 branches/feature-loggers	#71 branches/feature-loggers	#68 branches/feature-loggers	#38 branches/feature-loggers	#23 branches/feature-loggers
branch : branches/feature-distributed-grid	#111 branches/feature-distributed-grid	#109 branches/feature-distributed-grid	#108 branches/feature-distributed-grid	#103 branches/feature-distributed-grid	#101 branches/feature-distributed-grid	#88 branches/feature-distributed-grid	#80 branches/feature-distributed-grid	#79 branches/feature-distributed-grid	#54 branches/feature-distributed-grid	#50 branches/feature-distributed-grid
branch : branches/feature-collaboration	#106 branches/feature-collaboration	#99 branches/feature-collaboration	#96 branches/feature-collaboration	#95 branches/feature-collaboration	#77 branches/feature-collaboration	#70 branches/feature-collaboration	#69 branches/feature-collaboration	#27 branches/feature-collaboration	#26 branches/feature-collaboration	#22 branches/feature-collaboration
branch : branches/feature-bigdata	#121 branches/feature-bigdata	#120 branches/feature-bigdata	#119 branches/feature-bigdata	#117 branches/feature-bigdata	#107 branches/feature-bigdata	#104 branches/feature-bigdata	#102 branches/feature-bigdata	#98 branches/feature-bigdata	#74 branches/feature-bigdata	#73 branches/feature-bigdata
branch : branches/feature-cycle	#118 branches/feature-cycle	#116 branches/feature-cycle	#115 branches/feature-cycle	#113 branches/feature-cycle	#112 branches/feature-cycle	#105 branches/feature-cycle	#87 branches/feature-cycle	#83 branches/feature-cycle	#75 branches/feature-cycle	#72 branches/feature-cycle
branch : branches/feature-integration	#123 branches/feature-integration	#122 branches/feature-integration	#62 branches/feature-integration	#61 branches/feature-integration	#60 branches/feature-integration	#59 branches/feature-integration	#46 branches/feature-integration	#45 branches/feature-integration	#44 branches/feature-integration	#42 branches/feature-integration
branch : branches/feature-rdbms	#125 branches/feature-rdbms	#124 branches/feature-rdbms	#114 branches/feature-rdbms	#76 branches/feature-rdbms	#57 branches/feature-rdbms	#56 branches/feature-rdbms	#53 branches/feature-rdbms	#51 branches/feature-rdbms	#48	
branch : branches/feature-or-bugs	#37 branches/feature-or-bugs									
branch : branches/feature-punctuation	#110 branches/feature-punctuation	#97 branches/feature-punctuation	#91 branches/feature-punctuation	#89 branches/feature-punctuation	#86 branches/feature-punctuation	#82 branches/feature-punctuation	#81 branches/feature-punctuation	#78 branches/feature-punctuation	#65 branches/feature-punctuation	#63 branches/feature-punctuation

Figure 1: Representative example of the existing dashboard. This image is not from the actual system and is used purely for illustrative purposes due to confidentiality constraints.

- Very time consuming to analyse test results as information is cluttered and dense.

1.3.3 Initial Requirements

From the start, we were expected to deliver 4 different pages as part of the new dashboard:

- Teams page - Team overview page showing all the development teams.
- FP's page - Functional Product overview page showing all the FP's of a single team.
- Tests page - Test case overview page showing all tests of a single FP.
- Test case page - Detailed test case page showing the detailed information about a single test.

The initial MOSCOW requirements, that saw quite some changes, were the following:

- Add system status about each FP (like machine information, test versions etc.) and parse errors that are logged (M)
- Functional Product separation based on ownership of team (M)
- Zoom level navigation towards more detailed overview (i.e. Functional Product overview → Test case overview → Detailed test case) (M)
- Parsing of RFT test results (S)
- Add screenshot of failed test case (S)

- Add delta status showing +/- ratio of tests passing/failing (S)
- Add logging (of stack-trace) of failure (S)
- RFT and TAF reporting in same style (C)
- Jira test issue possible status changes (i.e. assignee, under test) (C)
- Detailed test case page (C)
- Add filtering to test case table (C)
- Add test duration: total of FP and per test case (C)
- Ability to select each team and dedicated server where all results are send towards (C)
- Test case priority listing: Based on set rules provide a prioritization list to provide focus to the team on test cases that should have the highest priority to be fixed (W)

1.3.4 Initial Mock-Up

Along with the MOSCOW list, we were provided with 2 mock-up designs, one for the FP's page and the other for the Tests page. The first mock-up, Figure 2, highlights the following features:

1. Ability to filter/search FPs in the daily summary to create a more clear overview
2. Team specific FPs that the team has ownership of
3. Status of the FP
4. Smart delta pill showing amount of new failures/successes compared to previous day (+/- for better or worse regression)
5. FP name and abbreviation information
6. FP test information
7. Status bar for amount of succeeded/failed/missing tests
8. Other FPs which have been tested but the team has no ownership of

When you select a FP card, an overview will be displayed about the tests and their current status. The second mock-up, Figure 3, highlights the following features:

1. Overview of the total amount of test cases and the amount of pass/fail/unknown test cases

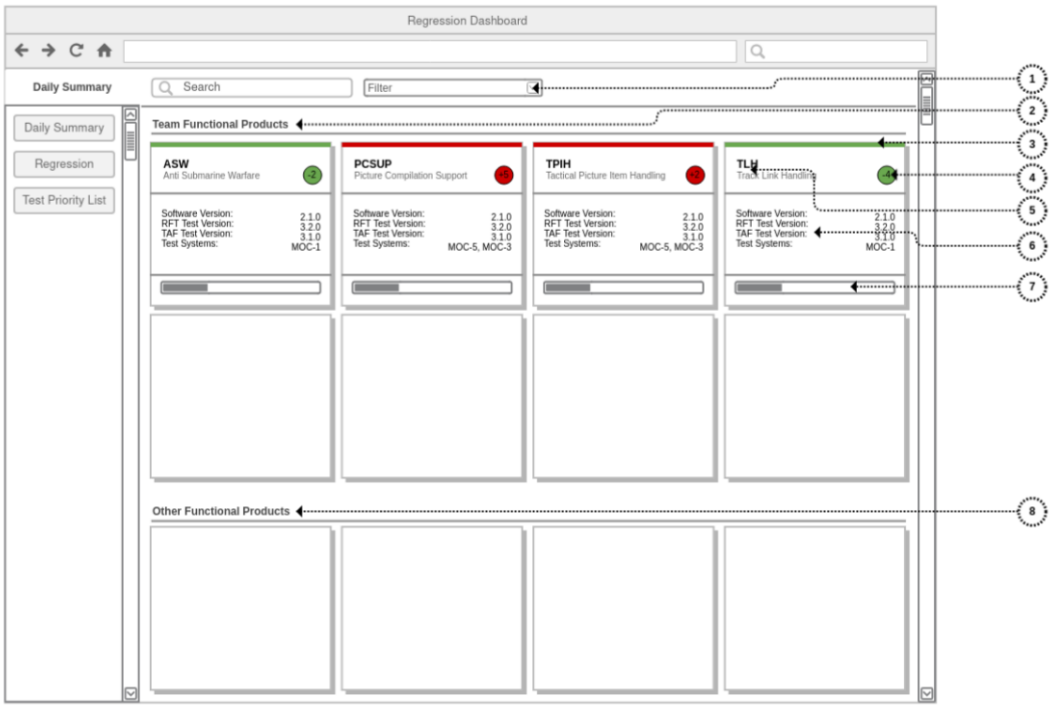


Figure 2: Mock-up from Thales, showing the FP's page and highlighting 8 different features.

2. Test case name with link to detailed test information
3. Jira link
4. Type of test case (RFT or TAF)
5. Status of the test case (Fail, Pass or Unknown)
6. Resolution of the test case (Broken, Unstable or None) (later called by the name *flakyness*)
7. Last time since the test case failed
8. The rate of success of the test case based on the history that is kept
9. Directly put the Jira ticket back to the test lane

1.3.5 Refined Requirements

After one week we received the the refined requirements from our supervisor at Thales. They were the following:

- FP separation based on ownership of the team. Being able to assign specific FP subset to a team.

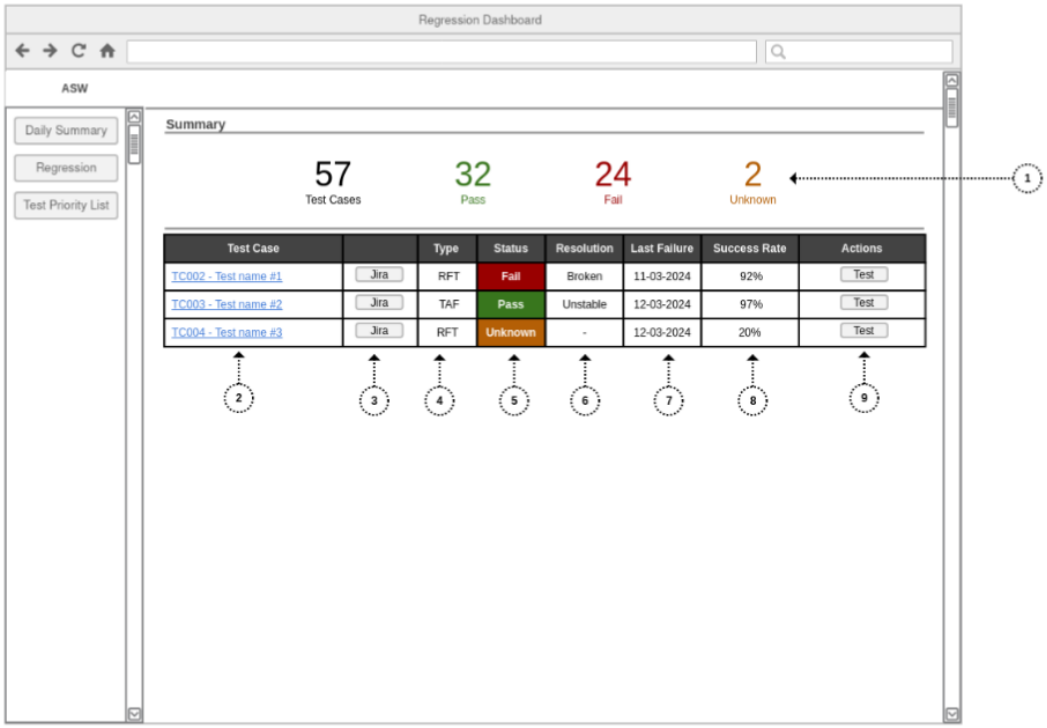


Figure 3: Mock-up from Thales, showing the Tests page and highlighting 9 different features.

- Team overview page. A page where you can have an overview of all teams in the system
- Functional Product overview page. A page where you can have an overview of all FPs assigned to a team
- Test case overview. A page where you can have an overview of all individual test cases for a specific functional product
- Detailed test case overview (exact information present still to be decided) (at least have the option for showing a stack trace window that includes possible picture from the test case failure). A page where you can have an overview of the detailed information about that case
- Zoom level navigation towards more detailed overview (i.e. Team Overview → Functional Product overview → Test case overview → Semi-Detailed test case). You should be able to select a team, then select an FP from their subset, then select an individual test case
- It is possible to provide a status to a test flaky/new/broken (possible options yet to be defined). In the "Test case overview" level you are able to select a status for an individual test case

- Functional product overview page should have a delta status show +/- results based on a pre-selected timeline. You should be able to select for example the last two weeks and see the delta results on the failures/successes
- Add filtering to test case table (RFT only/ TAF only) (Success/Failures). You should be able to at least filter on either RFT only or TAF only as well as the ability to filter on success/failures
- A central place to store the results of the daily regression per team. Such as a database that stores the data per team
- Test case overview should be pre-sorted by failing tests first, then passing after that it should be pre-sorted on RFT or TAF first to seperate the different type of tests. The test case overview table should be pre-sorted on either TAF on top or RFT since we want to separate their results as much as possible
- Test case overview should be sortable based on the different columns. You should be able to sort based on the columns available in the test case overview (test case number, status, last failure, fail count, etc.)
- Test case overview page should contain the product versions displayed somewhere. You should be able to show the versions of RFT and TAF that were used to run the regression shown on screen.
- Functional Product overview page should contain the product versions displayed somewhere. You should be able to show the versions of RFT and TAF that were used to run the regression shown on screen per FP next the FP name. (This depends on your implementation and how you see the best fit)
- Test case overview should have an option to select the date/s you want to look at. You should be able to select for which date/s you want to see the regression.

1.3.6 Extra Requirements

The goal of the project was to make a baseline working project, but because of our team's high effort and teamwork, we managed to take on a lot of extra requirements that evolved throughout the project. These requirements were:

- Possibility to add comments on test cases with a name.
- Date range filter, allowing the user to go back to specified date or date range.
- History of a test case's previous results on Test Case overview page.

- Possibility to create quick filters, improving customization and efficiency. Quick filters combine different filter parameters into a single button.
- Include settings page to edit and add quick filters.
- Showing failed test cases on hover of FP container in Team overview page.
- Small UX improvements - copy-on-click fields, tooltips on test cases, full use of the screen space.
- Implement error screenshots and extra error message descriptions for a *Test Case*.

2 Design

2.1 Global Design Choices

Our main design choice was to keep everything simple; starting from the technologies we use to designing the actual pages. As insisted by Thales, we put extra focus on UX, but still kept UI looking nice and clean. The main goal was to have a working product, no matter how many extra features it has; having the database, backend, and frontend work in unison without bugs was our ultimate goal. In addition, incorporating stakeholders in the process was crucial to deliver a product that helps the developer. As we had the mock-ups, great requirements, and a rather clear overview of the pages we needed to build, we did not use Figma to design a flow, but rather used modern tools to quickly develop a working MVP and reiterate on that.

To be specific, our repository contained one folder for React and other packages were dedicated to the Spring Boot. This structure allowed the group to have rapid access to every part of the project.

2.1.1 Backend – Spring Boot Application

Due to previous experience of some group members with Spring Boot and general proficiency in Java, the group decided to use this backend framework for connecting to the database and deriving business logic.

To ensure simplicity for colleagues new to Spring and to maintain future client usability, we adopted an MVC-based layered architecture. The project is structured into four main directories: *controller*, *model*, *services*, and *exceptions*.

The **controller** layer handles HTTP requests, mapping them to specific request addresses and invoking the appropriate service methods. The **model** directory contains:

- Entity definitions to create database models and manipulate data as Java objects,
- DTOs (Data Transfer Objects) to handle incoming and outgoing data,
- Repository interfaces that abstract database operations using Spring Data JPA.

The **exceptions** package is responsible for raising meaningful exceptions across the API. Business logic is implemented in the **services** directory, where each main object — *Team*, *FP*, and *Test* — has its own dedicated service class.

Overall, this structure aligns with Spring Boot best practices, promoting clean code, single responsibility, and separation of concerns between layers.

2.1.2 Frontend – React + Vite

Similarly, as some team members had extensive experience with React, we chose this framework for developing the user interface. The final project is built on a **React + TypeScript + Vite** stack and uses the popular Tailwind CSS plugin **DaisyUI** for styling and components.

2.1.3 Interactions and Technologies

To accelerate the development process, the group initially used an **H2 in-memory database** for fast data transfer and easy access. Concurrently, the React project was run in development mode using the `npm run dev` command.

In later stages of the project, the clients decided that maintaining everything on a single server - without a separate Node.js server to run React - would be more convenient and maintainable. As a result, we provided an option to host the entire project using only the **Spring Boot server**. This was achieved by building the React project and placing the compiled files into the `static` directory of the Spring Boot application.

Additionally, we decided to validate that our application could scale to a standard **PostgreSQL** database. We created a PostgreSQL server and established a connection between our Spring Boot application and a PostgreSQL instance hosted on **Railway**. Finally, the entire application was deployed to Railway, allowing clients to track our progress via a shared link.

2.2 User-Centered and Value-Sensitive Design

Throughout the project, we applied principles from User-Centered Design (UCD) and Value-Sensitive Design (VSD) to ensure that the dashboard not only functiones technically but also serves the real needs and values of its users.

User-Centered Design (UCD) is a framework that emphasizes designing products based on the needs, preferences, and limitations of users at every stage of the design process [1]. Research shows that UCD leads to higher usability, better user satisfaction, and more successful adoption of technology solutions [2]. In our project, we followed a UCD approach by involving Thales stakeholders early and continuously. We had regular feedback meetings with the company, at least once every two weeks and provided them with an online hosted prototype where they could see changes in real time. Changes such as redesigning pages based on user preferences, simplifying workflows, and adjusting the breadcrumb navigation system were all direct outcomes of this close collaboration.

Value-Sensitive Design (VSD) complements UCD by considering human values such as trust, privacy, and accountability in the design of technology [3]. In our case, VSD principles helped when making decisions around features like the comment system (see Dashboard Overview). Although it would have been technically possible to allow comment editing, we prioritized maintaining the

integrity of shared information. By preventing comment edits and requiring confirmation before deletion, we aimed to protect user intentions and prevent the misuse of collaborative features.

Together, UCD and VSD helped us build a dashboard that is not only functional and usable but also aligned with the values and working practices of Thales.

2.3 Dashboard Overview

In total, we made 4 different pages. Here is a deep dive into each single one and the design process behind it.

2.3.1 Teams Page

Teams page, Figure 4, is the simplest of the pages and the least visited one, as an average developer would only visit this once to choose their team and then bookmark the URL to their Functional Products page. This page is simple and contains cards with the team name and number of FP's belonging to the team. Clicking on one of the teams brings you to the Functional Products page of that team.

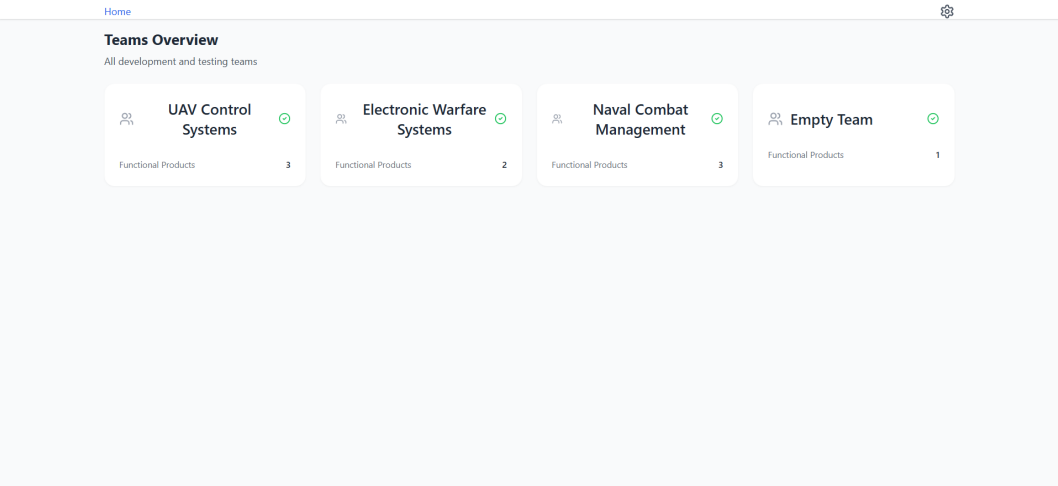


Figure 4: A screenshot of the Teams page.

2.3.2 Functional Products Page

Functional Products page, Figure 5, was intended to be quite simple in the beginning of the project, but as the project evolved, more extra requirements were added. Resembling the Teams page in its layout, it contains cards of FP's belonging to a single team. Each card has the FP name, latest software, TAF, and RFT version of the last run container. In addition, there is the time

and date of the last run and a pass/fail bar visualizing the ratio. For a quick overview, the whole card is color coded based on the results:

- Yellow - the container did not run the day before/missing container.
- Green - all of the tests in the container passed.
- Red - one or more tests in the container failed.

On hover over a card of an FP, a tooltip showing all the failed tests pops up, giving the possibility for the developer to have a quick insight without loading the whole FP. Clicking on one of the FP's brings you to the Test Cases page of that FP.

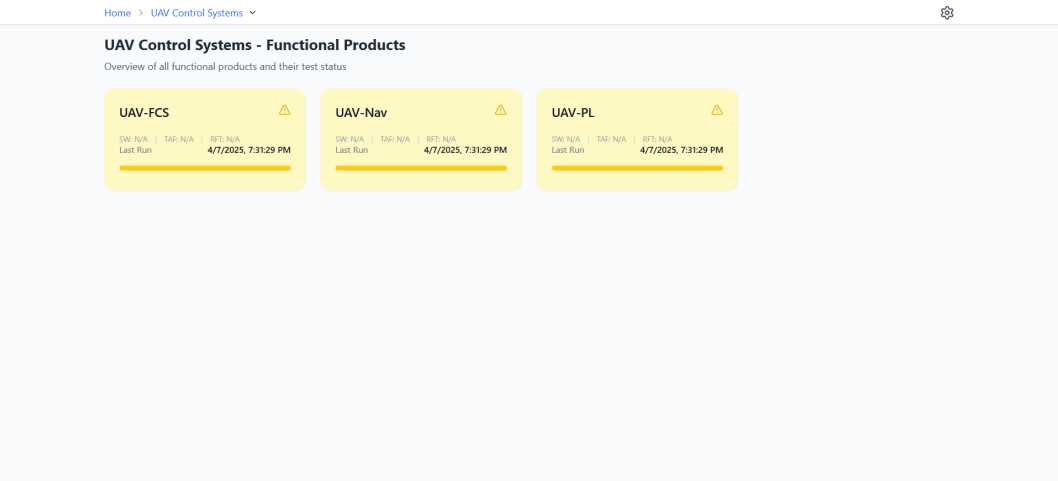


Figure 5: A screenshot of the Functional Products page.

2.3.3 Test Cases Page

Test Cases page, one of the more feature-rich pages. When directed to this page, the default filtering is applied automatically, based on the user's customization. This will include a single day view (list view), Figure 6, or a day range view (table view), Figure 7.

This page used to include a test/pass ratio bar, and a system information island, but later in the development, the ratio bar was deemed unnecessary and system information could be fitted in the header, making more room for the test cases.

In the list view, the test cases of that FP are shown as rows, it includes the following columns:

- Status - icon showing if the test passed or failed, if a single scenario fails, the test is considered to be failed.
- Test ID - the id of the test.

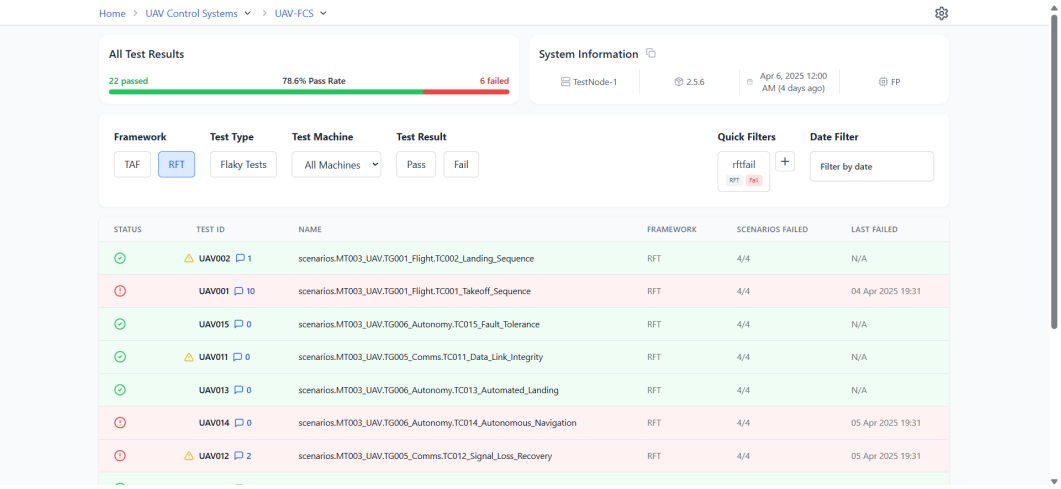


Figure 6: A screenshot of the Test Cases page in single day view.

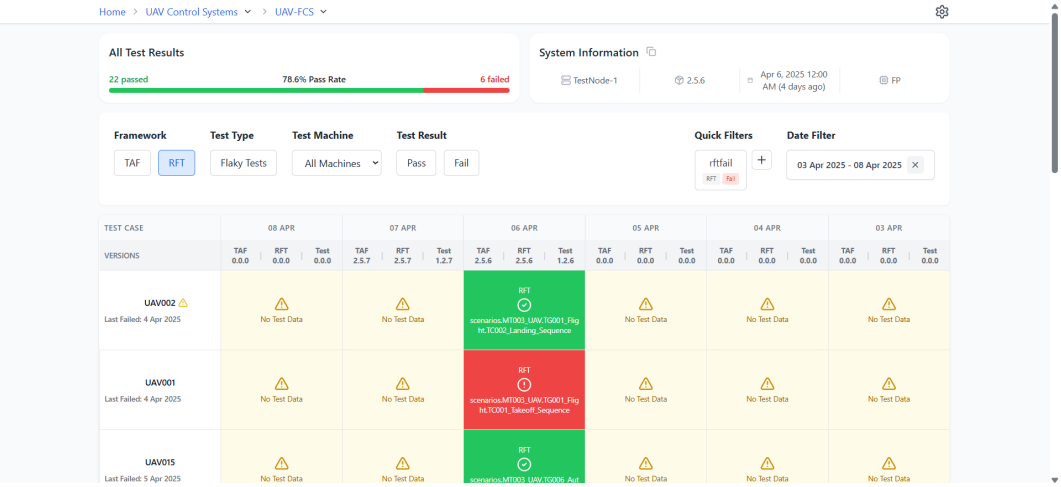


Figure 7: A screenshot of the Test Cases page in day range view.

- Name - the name of the test.
- Framework - the framework of the test, either RFT or TAF.
- Scenarios Failed - the amount of scenarios failed, as a single test can contain many scenarios.
- Last Failed - The date and time when the test last failed.

In addition to that, there are two flags: flaky flag, showing if the test has been deemed as flaky, and comments flag with a number, showing the amount of comments for that test case. The whole row is colored either red or green based on the status. Clicking on one of the test case rows brings you to the Detailed Test Case page for that test.

On the filter bar, the user can filter the tests based on framework, test type, test machine (the machine that the test ran on) and test result. In addition, the date picker allows the user to go back in history or even select a range, in which case the table view is shown.

In the table view, rows are test cases and columns are days, meaning each cell is a test case for a single day. Although containing less detailed information, this view gives the user a fast overview of the history without having to shuffle between single day views. Below the day, there are 3 version numbers for each day: software version, RFT version and TAF version. Similarly to the list view, each test case cell is colored red or green based on status, and contains the framework type and name. Clicking on one these cells brings you to the Detailed Test Case page for that test.

2.3.4 Detailed Test Case Page

Similarly to the Test Cases page, the Detailed Test Case page, Figure 8, is a rather feature-rich page. The top part of the page contains the general information about the test case. Right under the header is the name, last failed date and time, status, and flaky flag, that can be toggled. Once pressed it will trigger a confirmation pop-up to ensure the toggle was not accidental, as anyone access to the page can change it. The first container has all the information about the test case, including names, version numbers, etc. These are made all easily copyable by clicking the value. In addition, the user can copy the whole block of information by clicking the small icon in the corner. This makes it easy to share it to other developers in case something unusual has happened.

The next container holds the comments, showing the recent comments first, but giving the option to load earlier comments. Furthermore, there is a plus button allowing the user to add a comment. When pressed a dialog is prompted, Figure 9, asking for a name and description. Once the name is inputted it is saved in the LocalStorage of the browser therefore next time when adding the comment the name is already prefilled.

The run history container has the information about the same test case on previous days, giving a quick overview to the developer about the history. When one of the cards is clicked, the scenarios for this test case are expanded, shown in Figure 10.

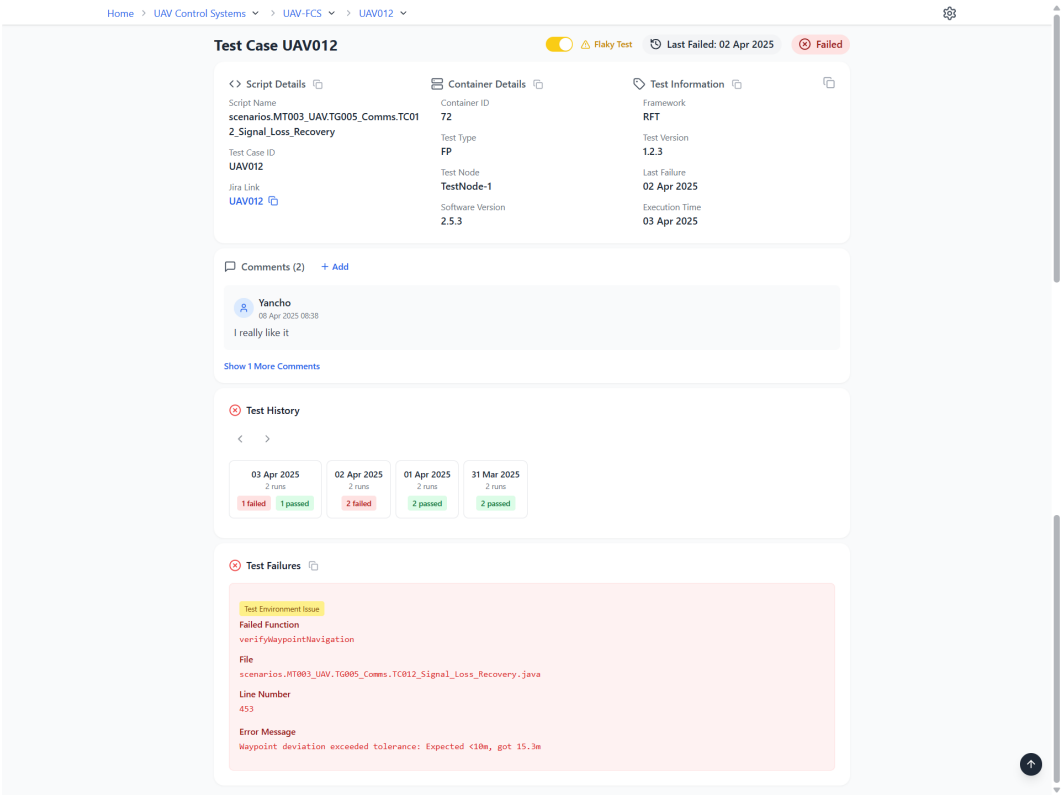


Figure 8: A screenshot of the Detailed Test Case page.

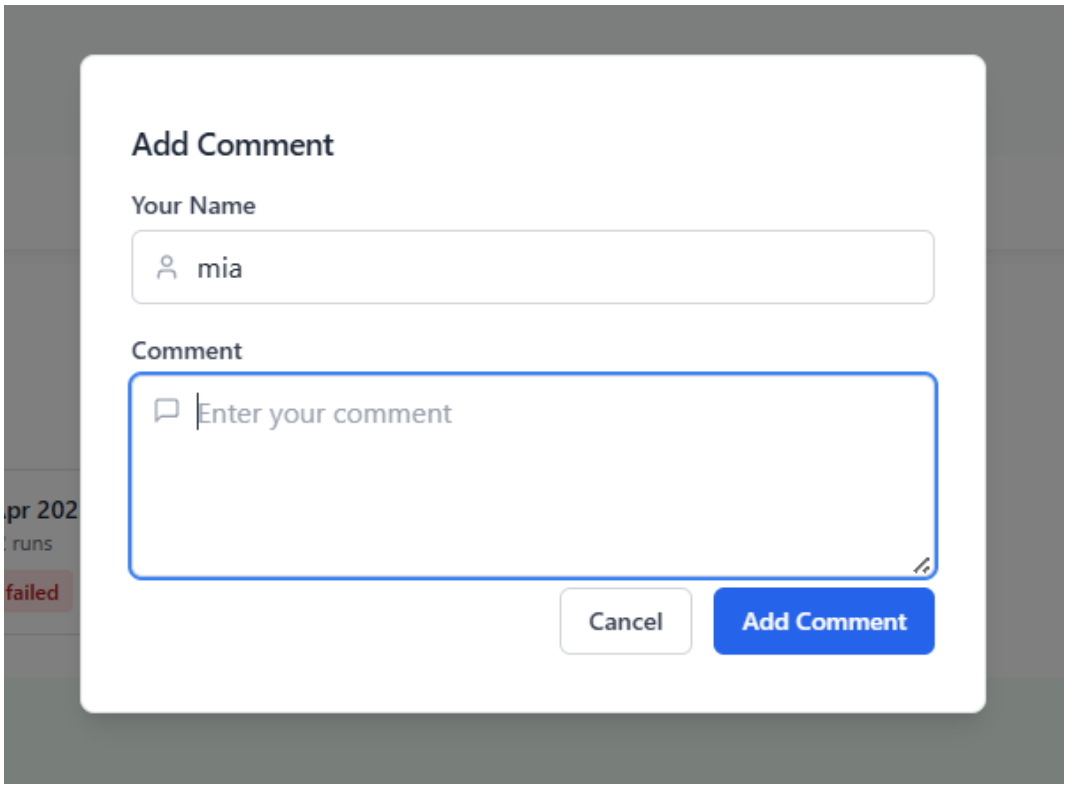


Figure 9: A screenshot of the comment pop-up of the Detailed Test Cases page.

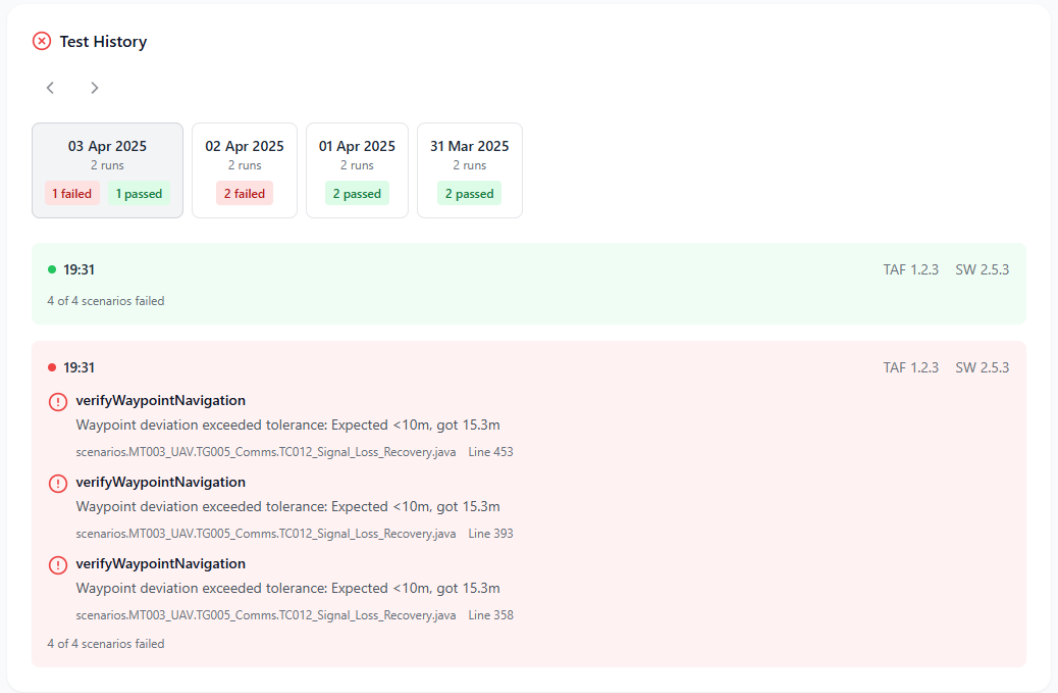


Figure 10: A screenshot of the history pop-up of the Detailed Test Cases page.

Lastly, in case the test case failed, there is a test failures container that holds information related to the error.

2.4 API

Our Spring Boot REST API follows a clear and modular structure, designed to remain maintainable and scalable. Most endpoints retrieve data from the database and return it in a format suitable for the frontend while fulfilling specific requests.

As stakeholders emphasized the need for performance and scalability, some APIs are designed to return **partial data** rather than full records. For example, we implemented **pagination** for certain endpoints - such as fetching comments - to optimize performance. When viewing a page with comments, the system initially loads the first 20 records. If the user wants to load more, an additional fetch is triggered to retrieve the next n comments.

Below is a full table of the implemented APIs. The placeholder `{{api}}` refers to the base address of the server.

Name	**Method**	**Endpoint**	**Description**
All Teams	GET	`{{api}}teams`	Fetches a list of all teams.
All Fps	GET	`{{api}}fps/ID`	Fetches FPs for a given team ID.
Breadcrumb	GET	`{{api}}teams/breadcrumb`	Fetches breadcrumb navigation data.
Recent tests By FP ID	GET	`{{api}}tests/ID/recent`	Returns recent tests for FP ID. Tests with containers for last timeframe.
Test Containers by FP ID and Dates	GET	`{{api}}tests/ID?startDate=""&endDate=""`	Fetches test containers for FP ID and within the given Dates.
Detailed Test Case	GET	`{{api}}tests/containerID/testID`	Detailed view of test case in specific container.
New Comment for TestInstance	POST	`{{api}}tests/ID/comments`	Posts a comment with the given user data and test ID.
Update IsFlaky	PATCH	`{{api}}tests/ID/flaky`	Updates flaky status of test ID.
Paginated Comments	GET	`{{api}}tests/ID/comments?page=""&size=""`	Returns paginated comments for the given test ID.
Paginated Failures	GET	`{{api}}tests/ID/failurespage=""&size=""`	Paginated failures by the given testInstance ID.

Table 1: API TABLE

2.5 Database

2.5.1 Design Tool

To make the schema, we used a simple online tool called dbdesigner.net[1]. This allowed us to quickly iterate on the database design with an easy-to-use tool, which also allowed collaborative functionality.

2.5.2 Schema Evolution

First Schema

The database underwent many evolutionary steps due to constantly changing requirements. The initial schema design, Figure 11, was rather simple - **it centered on expecting 1 container per day** and did not include comments. As stakeholders stated they expected one container per day, our initial idea was to derive a flow-based database design, where each table had a direct reference to the next and previous tables.

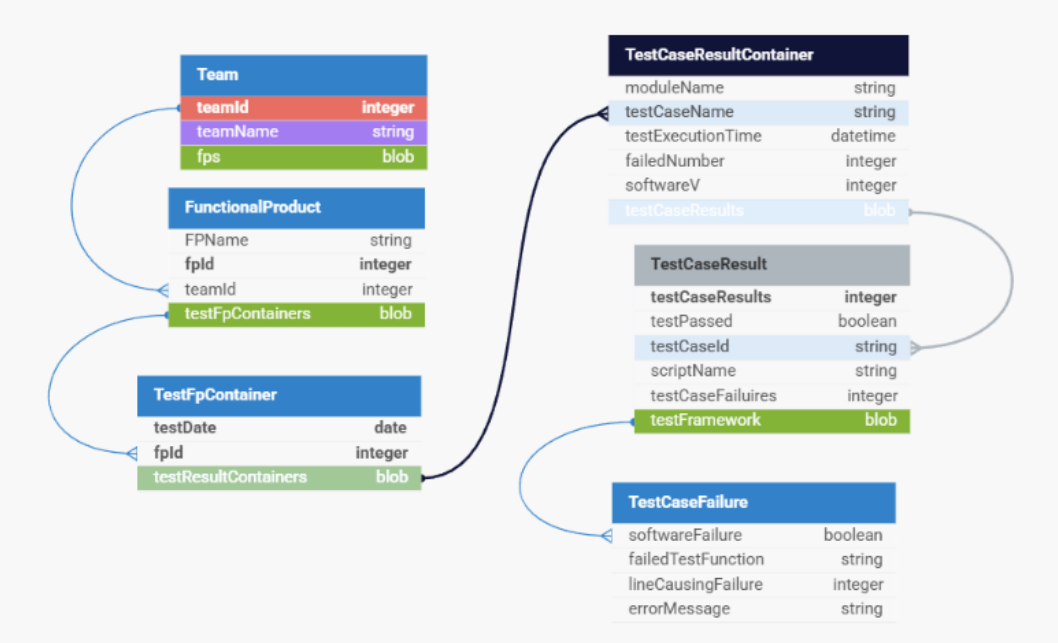


Figure 11: Our first database schema, designed and exported from dbdesigner.net.

The Figure depicts the class view of the Java model, where each *blob* field depicted on the Figure is a reference to a set of connected objects. These fields have a type of "blob" because of the limitations of the DbDesigner platform. For instance, the *Team* Java class has fields *teamId*, *name*, and a set of *fps*, while the *FP* table has a foreign key *teamId*, thus linking the two tables. Holding these sets allowed us to retrieve all the required information simply

[1] https://dbdesigner.net

and efficiently. Further database figures do not include Java model views but preserve a strict database view. Some of the classes no longer hold sets of objects for efficiency reasons.

Our first database design contained 6 tables:

- *Team* – representing each team within the Thales environment. The Java model also holds a set of FPs, allowing us to initialize a set of *FP* objects when retrieving all teams.
- *FunctionalProduct* – representing each FP and its belonging to a specific *Team*. The Java model held a reference to a set of *Containers* for the same reason as above.
- *TestFpContainers* – a table to link the specific data of a *Container* with the date and *FP*. Its Java model in Spring Boot contained a set of *TestCaseResultContainer* objects.
- *TestCaseResultContainer* – representing data of a container submitted on a specific day, where the testing machine or software could differ from day to day. Additionally, the Java model held a set of *TestCaseResult* objects.
- *TestCaseResult* – representing specific data of a test and its result. The Java model contained a reference to a set of *TestCaseFailure* objects, if they existed.
- *TestCaseFailure* – representing the description of a failure within the Thales environment.

Every table has a **One-To-Many** relationship with the next, except the *TestCaseFailure* table, as it was stated that there might be only one *TestCaseFailure* per **TestCaseResult**. In addition, our first design focused on simplicity and efficiency with respect to the requirements provided at the time. Having so many references within the Spring Boot Java models allowed us to eliminate many unnecessary queries and provided a more comfortable way to manipulate Java objects, rather than, for example, re-querying a *TestCaseResult* based on a given *TestCaseResultContainer ID*.

Second Schema

During our next meeting, we were informed that the system needed to expect **two containers per day** (1 TAF, 1 RFT), and that tests could have **comments**. The initial schema design was not prepared for these changes, but the new design, illustrated in Figure 12, remained simple enough and fulfilled the updated requirements.

The figure illustrates a clear database view with correct linking logic, preserving all unique primary and foreign keys. Because there could now be more than one container per day, our group decided to remove the *TestFpContainers*

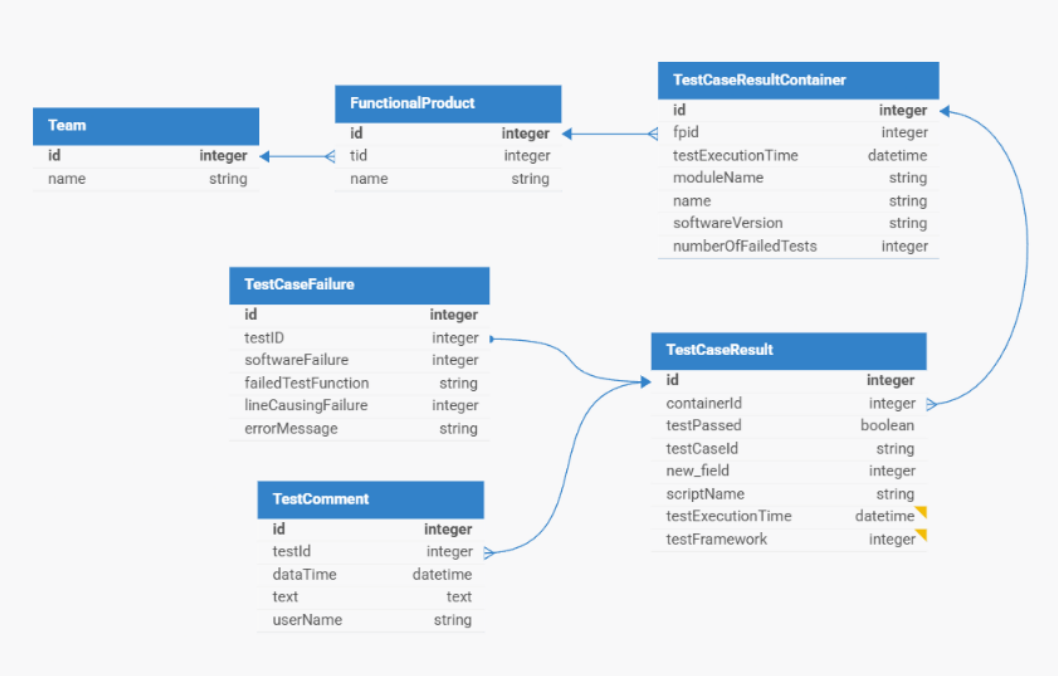


Figure 12: Our second database schema, designed and exported from dbdesigner.net.

table. Most fields remained consistent, except for a few minor changes. A new *TestComment* table was created to store comments.

In total, there were 6 tables:

- *Team* – representing each team within the Thales environment. **One-To-Many** connection to *FP*.
- *FunctionalProduct* – representing each FP and its belonging to a specific *Team*. **One-To-Many** connection to *TestCaseResultContainer*, and **Many-To-One** connection to *Team*.
- *TestCaseResultContainer* – representing data of a container submitted on a specific day, where the testing machine or software could differ from day to day. **One-To-Many** connection to *TestCaseResult*, and **Many-To-One** connection to *FP*.
- *TestCaseResult* – representing specific data of a test and its result. **One-To-Many** connection to *TestComment*, **One-To-One** to *TestCaseFailure*, and **Many-To-One** connection to *TestCaseResultContainer*.
- *TestCaseFailure* – representing the description of a failure within the Thales environment. **One-To-One** connection with *TestCaseResult*.

- *TestComment* – representing comments left by system users on specific test cases within the Thales environment. **Many-To-One** connection with *TestCaseResult*.

Final Schema

Further meetings with Thales supervisors provided a clearer overview of their internal architecture and introduced many new requirements. Several major changes included: the system should expect **n containers per day**, a single test could have multiple *TestCaseFailures*, comments should be globally connected to test cases, and some fields were moved or added. Previously, each test case result had a list of comments, which means that comments left on the same test case yesterday and today are different. As shown in Figure 13, test cases now support global comments and satisfy the new requirements. Compared to the previous design, where comments were linked to the *TestCaseResult* of a specific *Container*, the current system introduces a globally defined *TestInstance* table to store information about specific test cases executed on a daily basis. This new table is directly connected to the FP and lies at the same connection level as the *TestCaseResultContainer* table. This structure allows the system to rapidly query whether a specific test case belongs to a specific FP during parsing. The new table also eliminates test case data duplication, unifies comments by test case, and enables the use of flags, such as a *Flaky test* flag.

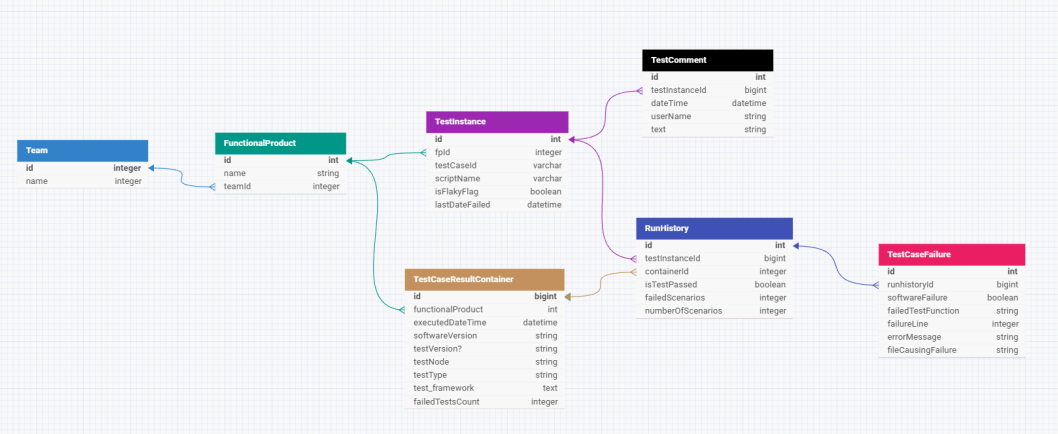


Figure 13: Our third and final database schema, designed and exported from dbdesigner.net.

The fields *dateTime*, *softwareVersion*, and *testVersion* are now stored only in the *TestCaseResultContainer* table, as per the client's new requirements. The new *RunHistory* table represents the result of a specific test case run in a specific container and records the number of executed and failed scenarios. This table allows us to store only changing information and avoid duplicating static data, such as *testCaseId* or *scriptName*. Each *RunHistory* entry is

uniquely identifiable by the *containerId* and *testInstanceId*, as there cannot be more than one run of a specific test case within the same container.

In total, the final design contains 7 tables:

- *Team* – representing each team within the Thales environment. Remains unchanged.
- *FunctionalProduct* – representing each FP and its belonging to a specific *Team*. Remains unchanged.
- *TestCaseResultContainer* – representing data of a container submitted on a specific day, where the testing machine or software could differ. Fields such as *softwareVersion* and *testVersion* were moved here. Connection changed to a single **One-To-Many** relationship with *RunHistory*.
- *TestInstance* – represents specific data of a test and its result. Some fields were removed; new fields such as *isFlakyFlag* and *lastDateFailed* were added. **Many-To-One** connection to *FP*, **One-To-Many** connections to both *RunHistory* and *TestComment*.
- *RunHistory* – represents the run of a specific test instance in a specific container and its result. **One-To-Many** connection to *TestCaseFailure*, **Many-To-One** connections to *TestInstance* and *TestCaseResultContainer*.
- *TestCaseFailure* – represents the description of a failure within the Thales environment. **Many-To-One** connection with *RunHistory*.
- *TestComment* – represents comments left by system users on specific *TestInstance* entries. **Many-To-One** connection with *TestInstance*.

2.6 Important Design Decisions

2.6.1 Breadcrumbs

An important feature that we put a lot of effort into is the breadcrumb[2] system. The idea is to make it possible for the developer to navigate around the website as easily and fast as possible. As seen on Figure 14, our breadcrumb bar has at most 5 levels, each of which is clickable and brings you back up the navigation tree. In addition, next to each of the levels is a down-facing arrow, which, once hovered over, allows the user to visit siblings of the navigation tree.

Each navigation level is also captured in the URL, therefore allowing the developer to use the browser's previous page and forward page navigation buttons. Each of the levels corresponds to one new page, except the 4th level, which on the Figure 14 corresponds to *moc-3*. This is a filter on the Test Cases

[2] https://en.wikipedia.org/wiki/Breadcrumb_navigation

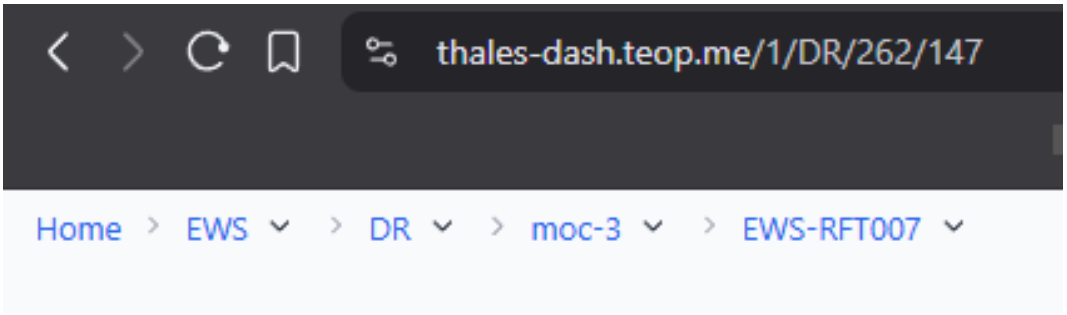


Figure 14: A screenshot of the breadcrumbs and URL bar.

page for one machine that the tests could've run on. This choice was made to group up test cases ran for one machine, so test cases could be separated by machine and be viewed with one glance.

2.6.2 Comments

One of the extra requirements was to include the possibility to add comments for a test case. Due to the nature of the system, where no authentication is present, several questions arise:

- Should it be possible to edit comments, given that anyone can edit any comment?
- Should it be possible to delete comments, since a user could delete comments that don't belong to them?
- Should comments be automatically deleted after some time, to prevent them from accumulating?

After extensive discussions with the client during our meetings, we decided not to include comment editing, as it could lead to confusion about the origin and intent of the comment. Additionally, implementing this feature would require storing and displaying an additional timestamp for each edit. Since comments could be edited multiple times - sometimes for grammar, other times for content changes - this might cause further ambiguity about the original creation date.

Therefore, we chose not to implement editing but opted to allow comment deletion instead. This, however, introduces the risk of malicious users, or simply careless developers, deleting comments that aren't theirs. To mitigate accidental deletions, we added a confirmation pop-up before a comment is removed.

Given the trust-based nature of the system and the absence of user authentication, we accepted the potential for abuse as an inherent trade-off and decided not to implement further restrictions.

2.6.3 Settings & Quick Filters

Similarly to comments, settings and quick filters were added as extra requirements during the development process. The settings pop-up, shown in Figure 15, became necessary once quick filters were introduced. It serves as a central place where users can create new quick filters or manage existing ones by activating or deleting them.

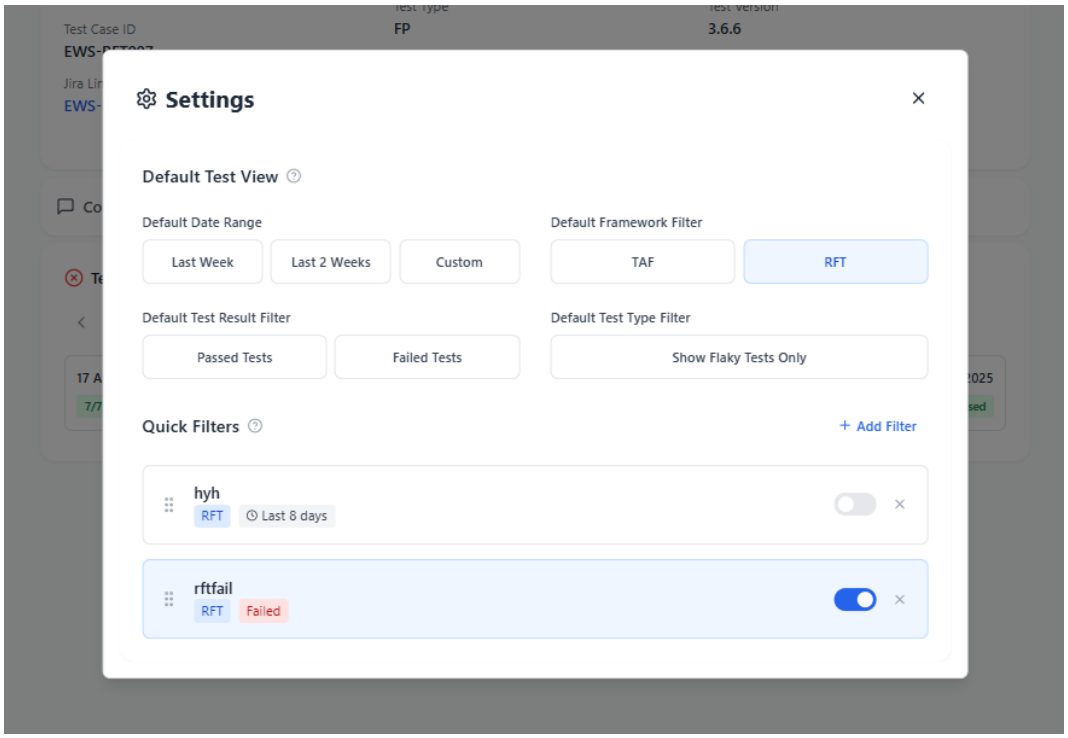


Figure 15: A screenshot showing the settings pop-up, with two quick filters, one of which is activated.

Quick filters allow users to group multiple filter values into a single reusable option. Once a quick filter is activated through the settings, a corresponding button appears on the filter bar. This makes it possible to apply several filters with one click - especially helpful when the same combinations are used frequently.

From a design perspective, the goal was to make the creation of quick filters both flexible and user-friendly. Users can create quick filters in two ways: directly from the *Test Cases* page by selecting filters and clicking the **plus (+)** button, or from within the settings pop-up itself. This dual approach supports both quick creation during workflow and more deliberate management in the settings view.

3 Implementation

This section outlines the technical implementation details of the dashboard system, including the technologies, tools, programming languages, and architectural decisions that we used to facilitate the project. While the design choices were discussed earlier, this part focuses more on code-level and infrastructure-specific decisions.

3.1 Technology & Tools

We chose to make the dashboard a web-based application because it was easy to integrate into the client's existing environment. The client already has a system in place for internal tools, so building a web app allows us to fit into their workflow without needing to install anything extra. This also makes it easier for different users within the company to access the dashboard from their browsers. We followed an API-based approach to keep the backend and frontend separated, which makes the system more flexible and easier to manage during development.

3.1.1 Tools

To manage the project, collaborate as a team and ensure a smooth development process we used the following tools:

- Jira[3]: Used for sprint planning, task tracking, and team coordination. It allowed us to break down the development process into manageable tasks and ensured a good way to track progress
- GitHub[4]: Our version control system and code repository. GitHub facilitated collaborative coding, issue tracking, and code reviews, making it easier to integrate features while avoiding conflicts.
- Postman[5]: Used extensively during development to test API endpoints. It helped us verify backend functionality before integration with the frontend, especially for testing edge cases and different inputs.
- DB Designer: Used during database development and further updates. It helped us to collaborate and share the current scheme seamlessly, especially in the middle stage of the project when we received specific requirements.

[3] https://en.wikipedia.org/wiki/Jira_(software)
[4] https://en.wikipedia.org/wiki/GitHub
[5] https://en.wikipedia.org/wiki/Postman_(software)

3.1.2 Languages & Frameworks

We used Java for the backend, together with the Spring Boot framework[6]. Java is a popular language that we were already familiar with, and Spring Boot made it easier to build and organize our API's. It helped us keep the code clean and focus more on the logic rather than setup. We chose this setup because it's reliable and it is well established as it is widely used and an industry standard. In addition, many of our teammates had previous experience with these technologies.

For the frontend, we used React[7]. It's a well-known library for building web applications and works well with API's. We chose React because it's easy to work with, fast, and lets us build reusable components. It also has a lot of support that helped us during development.

3.2 Frontend

The following subsections will outline how the frontend of the dashboard was structured and developed, including the project structure, key features and data handling mechanisms.

3.2.1 Project Structure

The frontend codebase is organized under 'frontend/src/' directory, following a modular architecture to ensure clarity, scalability and ease of maintenance, all important to our client as their testing framework is currently transitioning between two different software programs. Each directory within 'src/' has a specific responsibility:

- 'components/ui/': Contains base UI elements such as buttons, dialogs, cards, navigation menus, etc. These low-level components were designed to be reusable and consistent throughout the whole application.
- 'components/': Includes higher-level, feature-specific components used across the dashboard such as: "FilterComponent" - for selecting teams or filters; "TestResults", "TestSummary", "TestHistory" - to display various aspects of test data; "Breadcrumbs" for contextual navigation.
- 'views/': Responsible for rendering the complete pages and organizing components into views: "TeamsOverview.tsx" - overview of all the software development teams, "TeamView.tsx" - specific team overview including all the Functional Products that belong to that team, "FPView.tsx" - view of a specific Functional Product with all its associated test cases and "TestCaseView.tsx" - breakdown of individual test case.

[6] https://en.wikipedia.org/wiki/Spring_Boot
[7] https://en.wikipedia.org/wiki/React_(software)

- 'config/': Stored application-level configuration, in our case API endpoints
- 'services/': Logic for interacting with backend services via API calls
- 'types/': Centralized the TypeScript[8] type definition for different data models used across the application.

3.2.2 Data Handling

All API communication in the frontend is structured around a centralized configuration file: "apiConfig.ts". This file defines the base URL for backend endpoints, grouping them into categories such as "TEAMS", "FPS", "TESTS" and "COMMENTS", helping with keeping the code organized and easy to maintain. This approach promotes consistency and maintainability as it avoids hardcoded URL strings. It simplifies updates, as changes to the endpoint structures can all be made in the same location.

3.3 Backend

The following subsections will outline how the backend of the dashboard was structured and developed, including the project structure, API implementation and logging.

3.3.1 Project Structure

The backend was structured as a Maven project[9]. The code follows a layered architecture approach, separating responsibilities across configuration, controller, service, model and repository layers. This approach helped with keeping a clean separation of concerns, making the system easier to develop and test.

Package Overview:

- 'configuration/': Contains configuration related classes like "FileWatcherConfig" and general application settings. It also includes a "mockDataInjector" package used to generate some sample data during development for testing and showcasing purposes.
- 'controller/': Include the REST controllers that are responsible with handling the incoming HTTP requests and then routing them to the appropriate services. Each domain has its own corresponding controller: "FunctionalProductController", "TeamController", "TestController", "WebController", "XmlFileController"

[8] https://www.typescriptlang.org/
[9] https://maven.apache.org/

- 'exceptions/': Manages custom exception handling. The exceptions fall into two categories, domain-specific exceptions like: "XmlParsingException", "FileWatchRuntimeException", "InvalidDataException", and a centralized "GlobalExceptionHandler" to standardize error responses.

- 'model/': Contains all domain models and DTOs (Data Transfer Objects). It is structured into subfolders: 'dto/fps', 'dto/teams', 'dto/tests', 'dto/xml' contain the Data Transfer Objects used for API communication. Another subfolder is responsible for XML-specific models, such as "TestCaseResultXML" and "RegressionRunXML", that were defined separately to assist with parsing and serialization.

- 'repository/': Also a subfolder of 'model/', contains the Spring Data JPA repositories used for database interaction. Each repository is tied to its specific domain entity (e.g. "TestInstanceRepository", "FunctionalProductRepository", "TeamRepository"). Below the repository folder, the actual JPA entity classes are defined (e.g. "Test Instance", "FunctionalProduct", "Team")

- 'service/': Implements the core logic of the application. Services are again organized by domain: "TeamService", "TestService", "FunctionalProductService". A dedicated 'parsing/' subpackage is also present for mananging XML input and file monitoring logic: "XmlParsingService", "FileWatcherService" and "ContainerProcessingService".

This structure follows common Spring Boot best practices, making use of annotations such as "@Service", "@Repository", "@RestController" to enforce loose coupling (reduces interdependencies among system and application components) and ensures that each layer performs a specific role in the application.

3.3.2 XML Parsing

Tests are run automatically within the Thales environment after 19:00 daily. Each machine generates XML files based on the results of specific tests and test scenarios. A collection of test results, together with the versions and machine information, is grouped in TestContainer. Each XML file consists of several TestContainers. In addition, every container has a field to outline belonging to the functional product, which can be used to connect a container and a specific functional product. However, at our first requirements iteration, we were not given information about establishing a connection to the team. Thus, it was not indicated to which Team a container belongs. Each of these XMLs has to be parsed and saved within the system. Moreover, some information must be entirely created during parsing, like a new Team or Functional Product. As the parsing process needs to be fully automated, establishing connections or creating new Team models manually would be the worst choice. Therefore, when a new Team or Functional Product appears in the XML, our parsing

service also creates it within our system; otherwise, the parsing service searches for existing models and manages the connections.

Our stakeholders aimed to present a requirement that would be as close as possible to the final system, allowing them to make only a few adjustments for the deployment on their side. Many different approaches achieve the desired functionality. Nonetheless, stakeholders have agreed to have a "data" directory to watch where new XML files can be loaded and automatically saved in the system. The question that remained for a couple of meetings was, "How to get the Team to which a container belongs?". At the further requirements iteration, it was decided that the best design choice would be to have separate directories for separate teams.

The full parsing functionality can be separated into three services: File Watcher, XML Parsing and Container Processing services. File Watcher functionality monitors the main "data" directory and all "team's" directories within the primary directory. This service passes the freshly uploaded files to the XML Parsing service. Intuitively, the XML Parsing service concentrates on reading XML files and creating Java objects. Specifically, we use an external library called "Jackson" to parse XML files in DTOs on our side. After converting all XML data into Java Objects as DTOs, these objects are processed using the Container Processing service. The last service simply stores and, depending on the logic of the fields, creates all the required instances within the database.

Lastly, according to the client's needs our team has created additional API to save the list of XML files.

3.3.3 Logging

Every service involved in parsing should statelessly work and produce some results. Catching exceptions is extremely handy when discussing a logging system and having an always-working parser. The starting point of our parser lies in File Watcher Service, where we launch a separate thread to monitor each team's directory. If this thread at some point catches an exception, it is logged on the server, saved in the database, and a new thread is spawned to monitor this directory. Further services have specific catches that depend on logic. If a required field is missing in some container, our system logs it in the database, stops processing the current container and reverts all changes made in the database. Logging an expected error can ease the fixing process and instantly tell the system administrator the cause of this error. Additionally, all unexpected errors are logged and properly isolated to circumvent data loss.

3.3.4 Documentation

Together with the source code, our team will provide Thales with supportive documentation generated by Javadoc. The most important documentation within the code lies in the Repositories and Services. Additionally, we aim

to present a complete manual of the system that will cover the implemented solution in detail and provide a clear reasoning on the given design choices.

4 Testing

This section describes the testing strategies, methods and tools used to validate the dashboard throughout development. Testing has been done at multiple levels from API testing to end-user validation, making sure that both functional correctness and user satisfaction are met.

4.1 Test Plan

Our testing approach combined iterative feedback from stakeholders with technical validation through unit tests and API testing. After each development iteration, we presented the updated dashboard during meetings with Thales. This allowed us to gather valuable feedback and adjust the product to better meet expectations.

For backend reliability, we implemented unit tests targeting the most critical components, ensuring the system behaved as expected and making it easier to detect regressions during development.

In addition, we used Postman to test all our API endpoints.

4.2 User Testing

Our user testing strategy centered around continuous and direct feedback from real users. Each meeting included at least four stakeholders from Thales, often involving different people depending on availability. This diversity of participants ensured we received feedback from multiple perspectives, all of whom were familiar with the previous dashboard and had specific needs.

To support this process, we maintained an always up-to-date version of the dashboard hosted online. This allowed stakeholders to test new features between meetings and report any issues or suggestions as soon as they noticed them. This ongoing feedback loop was essential in refining the interface and ensuring the product aligned with user expectations.

4.3 Unit Testing

To ensure backend reliability, we wrote unit tests. We created tests that verify the behavior of individual components in such a way that they are easy to repeatedly run during development.

Our approach focused on writing tests around critical parts of the system logic. By testing components individually, we were able to identify issues during development without relying on the full application context. This made it easier to refactor and progress through development with more confidence

Unit tests were regularly run as part of our development workflow and updated as the code evolved. This helped us catch bugs, ensured expected behavior, and contributed to a more robust backend.

4.4 Continuous API Testing

For continuous API testing, we made use of Postman, a popular tool for developing, testing, and documenting APIs. Postman allowed us to define a collection of test requests that could be run automatically to validate the functionality and stability of our backend.

We created a Postman collection that included all the relevant endpoints of our API. These tests helped us catch problems early in development and ensured that new features did not break existing functionality. With Postman we could quickly test our APIs without writing much code.

Overall, Postman proved to be a valuable tool in maintaining the quality and reliability of our API throughout the development process.

5 Discussion & Conclusion

5.1 Discussion

In conclusion, we are very satisfied with the outcome of the project. Our teamwork was efficient and highly collaborative. The final state of the product is something we can be proud of. A few areas for improvement include:

- **Large-scale testing** – Conduct testing with each of the 20 teams. This would allow the final product to be refined according to everyone's specific needs.
- **Screenshot and error message integration** – Implementing screenshot capture and detailed error messages in the Test Case page. This would require parsing additional file formats and modifying the database schema.

Our collaboration with Thales was excellent. Meetings were efficient and productive, and communication with company representatives was prompt and helpful. Thales expressed satisfaction with our product and was proud of what we achieved. From Thales' side, some potential improvements include:

- **Clearer requirements list** – The project could have been completed more quickly if we had received finalized and well-defined requirements from the beginning. A significant amount of time was spent trying to clarify the initial expectations.
- **Accurate and up-to-date data** – Much of the work from the first two weeks had to be redone, including modifications to the database schema, as it was originally based on an outdated XML file. The file did not match the current output format being generated.

5.1.1 Meetings

In total, we had six meetings with the client, as well as a final presentation. All meetings and the presentation took place at the Thales campus in Hengelo. On average, each meeting lasted around two hours. They typically began with us demonstrating our progress and discussing each page in detail. We talked about both the completed features and the upcoming ones to be implemented. By the end of each meeting, we had a to-do list, which Thales sometimes refined afterward by assigning urgency levels to each item.

Several different representatives from Thales attended the meetings, all of whom were stakeholders and users of the old dashboard. This diversity of input helped us gather multiple perspectives and refine the product to better suit everyone's needs.

5.2 Distribution

The work distribution during this project was the following:

- **Mihai Buliga** - main focus on writings and assignments like the report, the project proposal and reflection part. Also contributed partly to the backend.
- **Hanno Remmelg** - main focus on writings and assignments like the report, the project proposal, poster and reflection part. Also contributed partly to the backend.
- **Teodor Pintilie** - main focus on frontend development.
- **Rudolfs Neija** - main focus on frontend development.
- **Volodymyr Lysenko** - main focus on backend development, but also contributed to the report. Also contributed partly to the frontend.
- **Sviatoslav Demchuk** - main focus on backend development. Also contributed partly to the frontend.

5.3 Conclusion

In conclusion, we successfully completed all of the refined requirements, as well as all but one of the additional requirements. The remaining requirement was:

- Implement error screenshots and detailed error message descriptions for a *Test Case*.

This requirement involved significantly more work than a typical feature. Thales would need to provide additional files containing screenshots and metadata for failed *Test Cases*. We would then need to develop a dedicated parser for these files, modify both our database schema and filesystem to store the data, and integrate the results into the frontend - though the frontend implementation would have been the smaller part of the task.

References

[1] D. A. Norman, "The design of everyday things (revised and expanded edition)," *Basic Books*, 2013.

[2] J. Gulliksen, B. Göransson, I. Boivie, S. Blomkvist, J. Persson, and Åsa Cajander, "Key principles for user-centered systems design," *Behaviour & Information Technology*, 2003.

[3] B. Friedman, P. H. K. Jr., and A. Borning, "Value sensitive design and information systems," *Human-Computer Interaction in Management Information Systems: Foundations*, 2006.