

DESIGN REPORT

Intelligent Shunting Solution for the Rail Industry

-----o0o-----

Authors:

Alexia Balotescu – s2920166

Khanh Nguyen – s2950944

Miroslav Atanasov – s2999897

Nhat Vy Dinh – s2803100

Thomas van der Boon – s2939347

Supervisor:

dr. Tiago Prince Sales

Client:

Roel Westenberg

Faculty of Electrical Engineering, Mathematics and Computer Science

Technical Computer Science, Design Project

April 2025

Abstract

This report outlines the design, development, and evaluation of an Intelligent Shunting Solution developed for Strukton Rail, with the objective of transforming marshalling yard operations by means of real-time sensor-based monitoring. Strukton Rail, being one of the largest infrastructure providers in the Netherlands, and Strukton Systems (their innovation branch), have recently developed sensors that can detect wheel passages and switch positions. While these sensors provide raw data, there was a need for an integrated, smart system that would translate said data into a visualization environment. This project answers that need by creating a web application that not only shows wagon movement and switch positions, but also provides historical playback, hazard notification, and system health monitoring. Our solution is based on technologies used by Strukton: Vue.js for front-end, C-sharp (C#) alongside the .NET framework for back-end, and SQL for the database. Our platform also has role-based access for yard operators, maintenance staff and administrators. Moreover, it also supports configuring yard layouts through a canvas-based UI. This project demonstrates that web and Internet-of-Things technologies can be combined to provide operational visibility in environments as complex as marshalling yards. With its flexible architecture, and interactive visualization platform, SmartShunt provides a foundation for future innovation. Through this project, not only do we offer a proof of concept for the sensors developed by Strukton, but also contribute to the broader digital revolution of the rail logistics sector.

Table of Contents

Introduction.....	4
1.1 Project Description.....	4
1.2 Domain Analysis.....	4
1.2.1 Introduction to the domain.....	4
1.2.2 General knowledge of the domain.....	5
1.2.3 Clients, users, and interested parties.....	5
1.2.4 Software environment.....	5
1.2.5 Current operational procedures.....	5
System Specification.....	7
2.1 Requirements.....	7
2.1.1 Stakeholders.....	7
2.1.2 User-level requirements.....	8
2.1.3 System-level requirements.....	9
2.2 Risk Analysis.....	11
2.3 Design Choices.....	12
2.3.1 Programming languages.....	12
2.3.2 Libraries and frameworks.....	12
2.3.3 Architecture.....	13
Implementation.....	14
3.1 Database.....	14
3.1.1 Overview.....	14
3.1.2 Database architecture.....	14
3.1.3 Database design analysis.....	16
3.1.4 Key database procedures and triggers.....	17
3.1.5. Advanced features.....	18
3.2 Front-end.....	19
3.3.1 Structure.....	19
3.3.2 Canvas-based yard rendering.....	19
3.3.3 Yard configuration interface.....	19
3.3 Back-end.....	20
3.3.1 Structure.....	20
3.3.2 API and routing.....	20
3.3.3 Real-time communication.....	21
3.3.4 Security and authentication.....	21
Testing.....	22
4.1 API Endpoints Testing.....	22
4.2 Frontend testing.....	23
Evaluation.....	25
5.1 Planning.....	25

5.2 Team Evaluation.....	25
5.3 Communication With Client.....	25
5.4 Final Product.....	26
Appendix.....	27
Database Schema.....	27

Chapter 1

Introduction

1.1 Project Description

Strukton Rail is a company which develops, installs, and maintains the railway systems within the Netherlands, with the scope of creating tracks that are optimally available, safe, dependable, and measurable. Their business unit, namely Strukton Systems, is based in Hengelo and focuses on developing innovative solutions for the rail systems with a powerful, dynamic team of system and software engineers. One of their latest developments is an Internet-of-Things (IoT) sensor, ‘Wheel Passage Sensor’, which collects data on train movements: wheel passages, direction, speed, time. Moreover, they are energy-efficient, as they can remain operational on a singular battery for years. Another innovative sensor is the ‘Switch Position Sensor’ which determines the direction of rail switches, together providing valuable insights and data for tracking routes and optimizing yard operations.

Marshalling yards are high-traffic environments where wagons are parked, assembled, and prepared for further transport. Considering these sites are extremely complex, with dynamic movement and limited space, there is a need for proper real-time monitoring and smart solutions that ensure operational efficiency and safety.

In order to make use of the full potential of Strukton’s IoT sensor technology, we developed an Intelligent Shunting Solution, a system that transforms raw sensor data into meaningful insights for a range of stakeholders. Our final product provides real-time wagon visualization, movement playback, sensor health monitoring, and comprehensive reporting. These functionalities assist operational teams in optimizing workflows and help maintenance crews validate sensor reliability.

1.2 Domain Analysis

1.2.1 Introduction to the domain

The domain of this project is monitoring and managing marshalling yards, which are critical in railway logistics. These environments are complex and dynamic, so we need correct monitoring for achieving operational efficiency and safety.

The Intelligent Shunting Solution we developed aims to solve the challenges in this domain. Some of them are: poor real time visibility of wagon position, inefficient manual tracking and limitations in how operational and maintenance teams are integrated. We are addressing these issues through IoT sensors which can modernize the management of marshalling yards, as well as contribute to digitalizing rail logistics.

1.2.2 General knowledge of the domain

Marshalling yards, particularly those used for cargo operations, are networks of parallel tracks, rail switches, and entry/exit points for trains. These systems get even more complex due to the movement of wagons and their limited space. As we discussed with Strukton, the current monitoring practices rely on manual labor and fragmented systems.

Strukton Systems developed two sensors, Wheel Passage and Switch Position, to improve this monitoring. They operate on battery power for years and have been designed to operate in specific places within the yard. Together they offer data on wagon movements, which can be used to digitally show train activities.

This domain is intertwined with growing technologies like real-time data visualization or AI decision making. Moreover, the system also aims to process operational information from these sensors, such as detecting when wagons carrying hazardous materials are next to each other.

1.2.3 Clients, users, and interested parties

The client of this project is Strukton Rail, more specifically their innovation branch, Strukton Systems. The users include yard managers, who are responsible for handling wagon movements, traffic controllers who oversee the routing done with switches, and maintenance teams who handle sensor functionality. Additional users can be logistics partners, cargo train operators and national infrastructure managers.

As an example, yard managers prioritize real-time monitoring while maintenance crews need access to sensor health and anomaly alerts. Moreover, there are end users such as freight operators and logistics companies who benefit from correct wagon localization. The system is designed to help with different user needs through appropriate dashboards and role-based access.

1.2.4 Software environment

Strukton Systems primarily utilizes a stack based on Vue.js for the frontend development, C# (.NET) for backend logic, and SQL Server for data management. This setup provides a scalable and maintainable foundation for our project and has been used in the development of the final product.

1.2.5 Current operational procedures

In the current situation, most marshalling yards do not operate with a fully digitized monitoring solution. The ways wagons are tracked, and switches are assigned often count on manual labor,

which can cause errors and delays. Moreover, maintenance teams usually handle sensor failures after they occurred, without constant diagnostics or predictive alerts.

The development of WPS and SPS by Strukton shows a movement toward automated monitoring. The system can show a real time overview of yard activity, playback and automated reporting by integrating these sensors, for operational and maintenance matters. This shows an important departure from conventional ways to handle such operations. It can lead to better accuracy, faster decision making and, overall, a safer working environment.

This domain of marshalling yard monitoring is complex but offers an opportunity for digital transformation. In this project, we are integrating IoT sensor networks, processing real time data and providing intelligent visualization. This contributes to a new, practical solution to long standing issues within rail logistics. We are aligning this system with the needs of multiple stakeholders, as well as building a scalable software environment. This Intelligent Shunting Solution can become a foundational tool for modern rail yard operations and future innovations within the field.

Chapter 2

System Specification

This chapter focuses on the overall system, considering the requirements elicitation during the project proposal phase, risk analysis, key design choices and the design of our user interface.

2.1 Requirements

The first phase of our project consisted of requirements elicitation. The initial list of requirements was sent to our client and validated; thus, the following lists represent the final requirements considered and incorporated in our final product.

2.1.1 Stakeholders

Before formulating any type of requirements, it is important to consider the stakeholders of the system – entities that interact with and benefit from our product.

- 1) Strukton - solution provider and innovator
 - a) Strukton Systems - developers of sensors and software
 - b) Strukton Rail Management teams – operation and management of railway networks
 - c) Project managers – oversee business and operations
 - d) Administrators – manage access control, configurations, reporting etc.
- 2) Railway owners and operators
 - a) ProRail - Netherlands' railway infrastructure owner
 - b) National railway operators - NS, Blawnet, Arriva etc.
 - c) Cargo trains operators - companies handling freight transport
- 3) Marshalling yard personnel
 - a) Yard managers - oversee train movements, scheduling, and track assignments
 - b) Traffic controllers - oversee switching of trains between tracks

- 4) External partners
 - a) Technology providers – suppliers of IoT sensors, software
- 5) Maintenance Teams
 - a) Strukton sensor management teams - monitor, maintain, and replace sensors
 - b) Rail management crews - handle physical infrastructure maintenance
- 6) End users – commercial beneficiaries
 - a) Logistics and Supply chain companies
 - b) Rail freight companies

2.1.2 User-level requirements

After completing the list of stakeholders, we formulated the user-level requirements that showcase how each will interact with our product. We thought of these in order to get more familiar with the project and consider multiple perspectives for implementation.

- 1) Strukton
 - a) As a Strukton developer, I want to:
 - i) see sensor data to analyze system performance
 - ii) make sure the system is adaptable to different yards
 - b) As a Strukton Rail Management team member, I want to:
 - i) monitor the overall system performance across different marshalling yards
 - ii) see reports on yard utilization and efficiency trends
 - c) As a Strukton project manager, I want to:
 - i) oversee the integration between software, sensors and yard operations
- 2) Railway owner and operators
 - a) As a ProRail employee, I want to:
 - i) monitor train and wagon movements across marshalling yards
 - ii) receive alerts on disruptions or safety incidents
 - b) As a national railway operator employee, I want to:
 - i) track passenger and freight trains within marshalling yards
 - ii) experience minimal delay by having optimized switch and sensor operations
 - c) As a cargo train operator, I want to:
 - i) make sure cargo with hazardous material is protected within a marshalling yard
- 3) Marshalling yard personnel
 - a) As a yard manager, I want to:
 - i) view train and wagon location within the yard

- ii) see reports on yard utilization
- b) As a traffic controller, I want to:
 - i) monitor switch positions for correct track assignments
 - ii) access previous events for logging and accident management
- 4) External partners
 - a) As a technology provider, I want to:
 - i) integrate my products with the marshalling yard platform
 - ii) receive feedback on system performance for improvements
- 5) Maintenance and Engineering teams
 - a) As a Strukton sensor management team member, I want to:
 - i) monitor sensor health and battery status
 - ii) receive alerts in case of failure or anomalies
 - iii) see when maintenance is needed based on sensor data/reports
 - b) As a rail management crew member, I want to:
 - i) locate malfunctioning sensors
 - ii) see the reports generated on performance
- 2) End Users
 - a) As a logistic and supply chain company employee, I want to:
 - i) see the cargo wagons within the marshalling yard
 - ii) see when cargo wagons might be misplaced
 - b) As a rail freight company employee, I want to:
 - i) monitor the exact location of my wagons
 - ii) see where hazardous goods are placed and take preventive measures

2.1.3 System-level requirements

The following system requirements are grouped based on the two types, functional and non-functional, and prioritized using the MoSCoW system: must have are essential for the system to function properly, should have are important but not essential for the initial deployment, could have are nice to have but not critical, and won't have are outside of the project's scope.

2.1.3.1 Functional requirements

Must have:

- 1) The program must provide real-time visualizations of wagons and train movements.
- 2) The program must have playback functionality.
- 3) The program must monitor sensor health, allowing maintenance teams to timely detect and resolve sensor issues.

- 4) The program must generate reports for different departments based on collected data.
- 5) The program must track and display the location of wagons within the yard.
- 6) The program must implement role-based access for different user levels (administrators, operators, and maintenance crews).
- 7) The program must include functionality to configure yards with in/out tracks, switches, and connecting tracks.

Should have:

- 8) The program should identify wagons carrying hazardous materials and trigger alarms when such wagons are positioned close together on parallel tracks.
- 9) The program should provide a confidence indicator for displayed data.
- 10) The program should validate data accuracy by cross-referencing different sensor inputs.
- 11) The program should set off an alarm when movement is detected on a track that was not expecting any.

Could have:

- 12) The program could implement AI-assisted wagon layout optimization to improve efficiency.
- 13) The program could optimize sensor placement for better accuracy.
- 14) The program could support real-time data streaming from the sensors.

Won't have:

- 15) No additional sensors will be integrated beyond the existing WPS and Switch Position Sensors.
- 16) The program will not set off an alarm when a sensor is damaged/bent by the force of a train coming from the wrong direction.

2.1.3.2 Non-functional (quality) requirements

Must have:

- 1) Performance
 - The program must provide visualization updates within less than 5 seconds when real-time sensor data is used.
- 2) Reliability
 - The program must accurately display wagon locations in at least 99% of cases when no sensor failures occur, and the data from the sensors is correct.

3) Maintainability

- The program must be designed to allow the maintenance team to identify and resolve issues promptly.

Should have:

4) Usability

- The program should have a user-friendly and intuitive interface so that non-technical users can operate it effectively.

5) Security

- The program should not allow third parties to have access to sensitive information.

Could have:

6) Compatibility

- The website could be able to run on different operating systems (i.e., Windows, Mac and Linux).

Won't have:

7) Portability

- The program won't be designed to function on both desktop and mobile web platforms.

2.2 Risk Analysis

During the development of the project, a number of risks were identified. One of the primary risks was misunderstanding the client's requirements, which could result in a system that has critical functionality missing, or that does not meet user expectations. To mitigate this, we organized weekly meetings with our client, validated the requirements we came up with, always asked for feedback and maintained detailed notes of every meeting. This way we made sure that the required features and use cases were mutually understood.

Another major risk is how the system relies on accurate sensor data from the WPS and SPS. Delayed or incorrect messages could compromise wagon tracking and visualization. We try to mitigate this by monitoring sensor health and use simulations to validate functionality when real-time data might be incomplete or missing.

Moreover, errors in yard configuration (such as malformed JSON) could also lead to incorrect visualizations. This can be mitigated by adding input validation and a visual editor when creating yard layouts.

2.3 Design Choices

In this chapter, we analyze the different choices we made for the overall system and its implementation. More detailed usage is explored in the following chapter.

2.3.1 Programming languages

The system is built using a combination of C#, JavaScript, and SQL. They are the languages suggested by the client, because they are suitable in different layers of the application:

- C# is used for the backend. It supports the development of our REST API using ASP.NET Core and integrates well with SQL Server via the Entity Framework. The language has strong typing and support for asynchronous programming, which help create a scalable and maintainable system
- JavaScript and CSS (specifically in combination with Vue.js) are used in the frontend for dynamic UI updates, real-time interaction, and event-driven features like yard visualization and playback animations
- SQL is used for defining and managing the relational database, including complex stored procedures and triggers to handle wagon movements, alerts, and system events

2.3.2 Libraries and frameworks

Several frameworks and libraries have been used to ease development and create a modular application:

- ASP.NET Core is the primary backend framework. It enables the development of secure RESTful APIs and integrates with the Entity Framework for database access
- Entity Framework Core is used for object-relational mapping (ORM), simplifying database operations by using C# objects to represent database records
- ASP.NET Identity Framework is used to handle both authentication and authorization in a secure and extensible manner. In this implementation, it manages user registration, login, role assignments, password hashing, two-factor authentication (2FA), and JSON Web Token (JWT) generation. This setup ensures that only authorized users can access protected resources and that sensitive operations, like user registration or role changes, are restricted to Admins.
- JWT (JSON Web Tokens) are used for authentication and authorization, integrated via ASP.NET Identity and custom user-role mappings

- SignalR provides real-time communication capabilities, used to push updates from the backend to the frontend without polling
- Vue.js is the frontend framework used for building the web interface. It has reactive data binding and component-based structure, which made it suitable for building our user interface
- Canvas API is used in the frontend for creating the yard layout, animating wagon movements, and supporting user interaction
- Swagger for testing and documenting our API

2.3.3 Architecture

Our application follows a three-level architecture, consisting of:

- Presentation Layer (Frontend): This layer is built in Vue.js and is the one interacting with the user, generating real-time visualizations of the yard and containing playback controls and administrative tools (such as yard editing)
- Business Logic Layer (Backend Services in C#): This is the layer which includes controllers (handling API requests), services (logic like sensor data processing and wagon movement), and SignalR hubs for real time communication. There are business rules integrated here such as wagon hazard validation and track occupation status
- Data Access Layer (Entity Framework & SQL Server): Responsible for data keeping and query execution. The database schema is normalized and includes support for user roles, yard and track configurations, wagon tracking, event logging, and safety alert rules

Such a layered approach creates separation of concerns, supports scalability and allows for independent development and testing of different components.

Chapter 3

Implementation

3.1 Database

This chapter focuses on explaining the structure of our database and the logic behind our choices, separated into two domains: the authentication part, involving users and their roles, and the yard management domain. The full database can be seen in Appendix.

3.1.1 Overview

The SmartShunt database implements a robust marshalling yard management system designed to:

- Track train wagons
- Monitor sensors
- Process events
- Maintain safety protocols

The system architecture consists of two main domains:

- Authentication management
- Yard operations management

3.1.2 Database architecture

3.1.2.1 Authentication domain

The authentication domain manages user access and permissions with a role-based security model.

- Companies: stores organizations that own/manage yards
- AspNetUsers: contains user details, credentials, and profiles
- AspNetRoles: defines system access roles (Admin, Operator, Maintenance)
- AspNetUserRoles: maps users to roles (many-to-many relationship)

Key features:

- Fine-grained permission management
- Security features like two-factor authentication and account lockout
- Association of users with specific companies

3.1.2.2 Yard management domain

The yard management domain handles physical infrastructure, sensor data, wagon tracking, and safety monitoring.

- Yard – defines a single marshalling yard, including a reference to its configuration and owning company
- Track – a physical segment of rail within a yard
- Sensor - represents the generalization of the two types of sensors
- SwitchSensor - the sensor on the switch
- WheelPassageSensor - the sensor of the wheel counting
- SensorMapping - creates associations between wheel passage sensors and switch sensors; therefore, enables the system to correlate data from different sensor types at the same location
- SensorEvent - generated automatically by triggers when conditions change and serves as the main event log for operational monitoring and notification
- Switch – determines routing options, includes coordinates and status.
- SwitchesAndTracks – maps how switches connect to tracks, including orientation and hierarchy
- Message – stores data from sensors including switch direction, wagon count, movement direction, and timestamp
- Wagon – represents an individual wagon in the yard, including its UIC code, location, and order
- SafetyAlertCombinations – encodes safety rules about which wagon types must be kept apart.
- History - contains data for playback functionality
- Notifications - stores information for alerts

3.1.3 Database design analysis

3.1.3.1 Table inheritance pattern

The database implements a form of table inheritance for sensors:

Sensor (base table)

└─ SwitchSensor

└─ WheelPassageSensor

Benefits:

- Common sensor properties defined once
- Specialized behavior for different sensor types
- Easy querying across all sensors or specific types
- Consolidated sensor health calculation using computed columns

3.1.3.2 Event processing architecture

The database incorporates a sophisticated event processing system:

Message (raw data) → trg_InsertSensorEvent → SensorEvent → SensorEventNotification
trigger → Service Broker

This creates a robust event pipeline that:

- Processes raw sensor data
- Updates system state based on sensor input
- Generates appropriate events and notifications
- Maintains historical records for auditing and playback

3.1.3.3 Wagon safety monitoring

- SafetyAlertCombinations table defines incompatible wagon types
- CheckSafetyAlert procedure evaluates wagon proximity hazards
- Safety severity levels (High, Mild, Low) are calculated
- Events are generated for safety violations

3.1.4 Key database procedures and triggers

3.1.4.1 Main procedures

CheckSafetyAlert

Evaluates wagon combinations on tracks for potential safety hazards:

- Examines adjacent wagons for compatibility issues
- References SafetyAlertCombinations table for rules
- Generates appropriate safety events with severity levels

UpdateTrackTrafficStatus

Monitors track capacity and updates traffic status:

- Tracks with < 15 wagons: green light
- Tracks with 15-25 wagons: yellow light
- Tracks with > 25 wagons: red light

3.1.4.2 Critical triggers

trg_InsertHistory

Creates historical records when wagon data is inserted, enabling:

- Complete wagon movement tracking
- Playback of historical yard states
- Audit trail of all wagon operations

trg_InsertSensorEvent

Complex trigger that processes incoming sensor messages:

- Updates sensor health metrics
- Processes heartbeat messages for monitoring
- Handles wagon movement between tracks
- Manages wagon positioning and order
- Handles terminal tracks and wagon removal
- Verifies axle counts for data validation
- Triggers safety checks and traffic updates

trg_SensorEventNotification

Implements Service Broker messaging for real-time notifications:

- Converts events to JSON format
- Sends messages through Service Broker queues
- Enables real-time client notification

3.1.5. Advanced features

3.1.5.1 Sensor health monitoring

The system calculates sensor health metrics automatically:

- Lifetime calculations based on installation date
- Last alive status tracking
- Failure rate monitoring
- Confidence score computation

3.1.5.2 Wagon movement algorithms

- Positional management based on switch orientation
- Source and destination track position recalculation
- Terminus track special handling
- Wagon order preservation
- Axle count validation

3.1.5.3 Service Broker integration

The database uses SQL Server Service Broker for asynchronous messaging:

- Queues for sending and receiving events
- Contract and message type definitions
- Trigger-based message generation

3.2 Front-end

3.3.1 Structure

Our front end is built using Vue.js due to its reactive component system, combined with JavaScript for logic and CSS for styling. The structure consists of two folders. The first one is the public folder, containing images used in design. The second one is the source folder which consists of the following directories, structured to be modular and respecting industry standards:

- Assets: contains json files which model yards
- Components: reusable UI components such as sidebars and headers
- JavaScript: contains the js files for logic, such as rendering and authentication
- Router: has a file that defines all route-to-view mappings using Vue router
- Views: contains full-page components that represent different screens within the app

Moreover, there are two files, main.js and index.html, which represent the starter files.

3.3.2 Canvas-based yard rendering

The core of the frontend's interactivity is the visualization system based on canvas. The yards are modeled through JSON configurations containing metadata (yard size, switch position, track layouts). Then, when a yard is loaded, the gridlines and coordinate rules are drawn and track and switches are placed based on their coordinates. Users can click on elements for more information, or for visual feedback.

Real-time events, such as wagon movement or switch toggling, are reflected on the canvas through SignalR push updates, which allows the system to react instantly to sensor changes.

3.3.3 Yard configuration interface

The app includes a yard creation tool in order to support admin control and configurable yards. It allows admins to:

- Upload a background image
- Overlay a grid
- Place switches and tracks using grid coordinates
- Edit and validate input
- Save the configuration as json

This configuration tool has the primary goal to simplify onboarding for new yards without needing to have access to the database, such that operators can update the infrastructure visually.

3.3 Back-end

3.3.1 Structure

The backend is structured based on industry practice, divided into logical components based on responsibilities, therefore adhering to the Separation of Concern principle. Below the folders are explained, including their purpose and files.

- **Controllers:** this folder contains files that define the API endpoints exposed to the frontend. Each controller handles HTTP requests by delegating the logic to services
- **Models:** this directory contains all the database tables representations. Each column is a property with its respective type and has getters and setters
- **Services:** All extra logic needed for the models is under services. Each service is a static class that has some logic for the model, it only contains methods that correspond to a model
- **DTOs (Data Transfer Objects):** This package contains classes that are used for the deserialization of the data contained within the incoming requests. They were used to take advantage of ASP.NET automatic deserialization which saved us time and let us focus working with the data rather than doing the deserialization ourselves
- **Hubs:** directory of SignalR hubs, used for real-time communication between the connected clients and the server
- **Data:** has the file of the database context to integrate the Entity framework with the models

Apart from the folders, there is the main program for configuration, the app settings file which stores runtime configuration among other things.

3.3.2 API and routing

Our backend contains RESTful API endpoints using ASP.NET Core controllers. Each controller handles a certain functionality such as user authentication, yard configuration or wagon management. They route and modify incoming HTTP requests to the corresponding services.

Routing is configured automatically through ASP.NET's attribute-based routing, example:

```
[ApiController]
[Route("api/yards")]
1 reference
public class YardController : ControllerBase
{
```

The API follows the conventional routing patterns, for example:

- GET /api/yards/{id} – fetch the metadata of a specific yard

- POST /api/notification – create a notification
- PUT /api/companies/{id} – edit a company name
- DELETE api/companies/{id} – remove a specific company

The project includes SwaggerUI that we used for testing and SwaggerDoc for documenting our API. When running the backend, Swagger UI is accessible at: <http://localhost:5194/> and the Swagger Doc can be found at <http://localhost:5194/swagger/api-docs>.

3.3.3 Real-time communication

We used SignalR, a real-time communication library built within ASP.NET Core, to support a responsive interaction with sensor data. This library allows the backend to instantly send updates to connected clients. We defined two main communication hubs:

- One for sensor events, related to wagon movement and switch updates
- One for notifications which are grouped by user role and company (so that messages are only delivered to the relevant users)

On the frontend, clients connect to the SignalR hub and listen for specific events, allowing for UI elements to update immediately when the backend state changes.

SignalR was needed to avoid polling, so instead of querying the backend every few seconds, the frontend receives data only when something changes. Moreover, we reduce network overhead and server load.

3.3.4 Security and authentication

The application uses ASP.NET Identity for managing user authentication and their roles. Users login with credentials and receive a JWT which is then used to authenticate following requests to the API.

The role based access control, meaning different endpoints and views are accessible based on the user's role, amplifies security. The system displays role-specific dashboards and restricts the permissions of each type of user. Moreover, each user is associated with a certain company via the company's id. This way, all tracks, wagons and sensor data are scoped to a specific company. Therefore, users can only access and interact with their organization's corresponding infrastructure.

Chapter 4

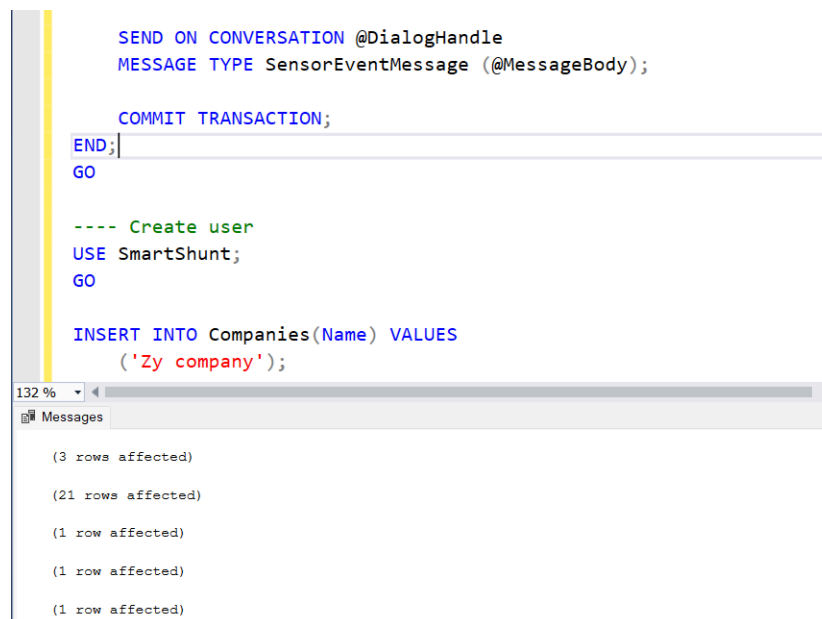
Testing

4.1 API Endpoints Testing

Since our backend contains various controllers that are in charge of establishing API endpoints for routing, it is essential to carry out automated testing to check if the API endpoints work and return values as expected.

Specifically, in our testing, we used RestAssure.Net (which is a framework built for C# .NET ecosystems)[<https://github.com/basdijkstra/rest-assured-net/wiki/Usage-Guide>] to test with every single endpoint that we have. Additionally, we used the Hamcrest framework to write matcher objects to check whether the returned values match with what we expected.

Since this is considered integration testing (because some endpoints relate to the others), we require some rows of already-existing data to compare and check. Thus, we first initialize the database with some predefined rows. After that, the tests are conducted in a particular order to show a simple procedure (or a workflow) of how things are operated in the application.



```
SEND ON CONVERSATION @DialogHandle
MESSAGE TYPE SensorEventMessage (@MessageBody);

COMMIT TRANSACTION;

END;
GO

---- Create user
USE SmartShunt;
GO

INSERT INTO Companies(Name) VALUES
('Zy company');
```

132 %

Messages

(3 rows affected)

(21 rows affected)

(1 row affected)

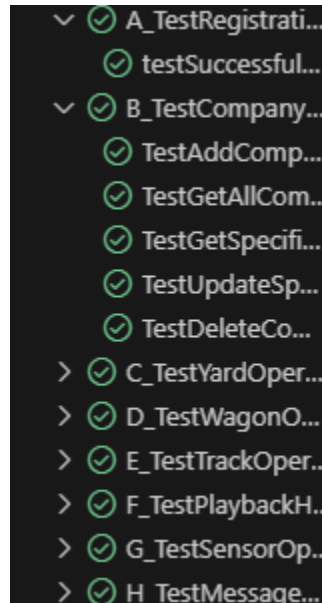
(1 row affected)

(1 row affected)

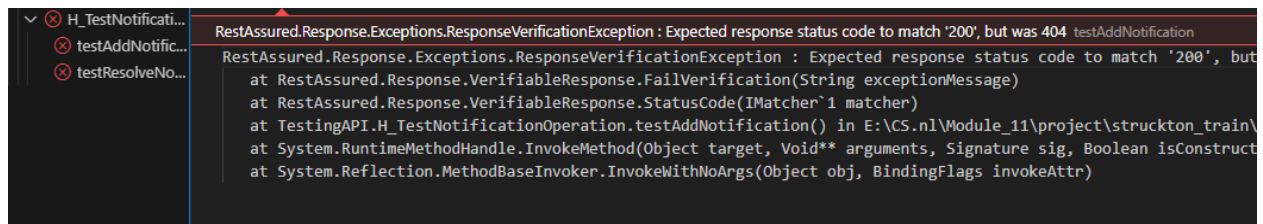
Initialization of the database

For instance, the RegistrationTesting will be first carried to create an instance of the user, where other operations can only be performed after this. In these tests, we will access the API endpoints

to fetch the data given some predefined parameters. After that, based on the returned status code, we will assess whether all the test cases passed or not.



An example of passed test cases

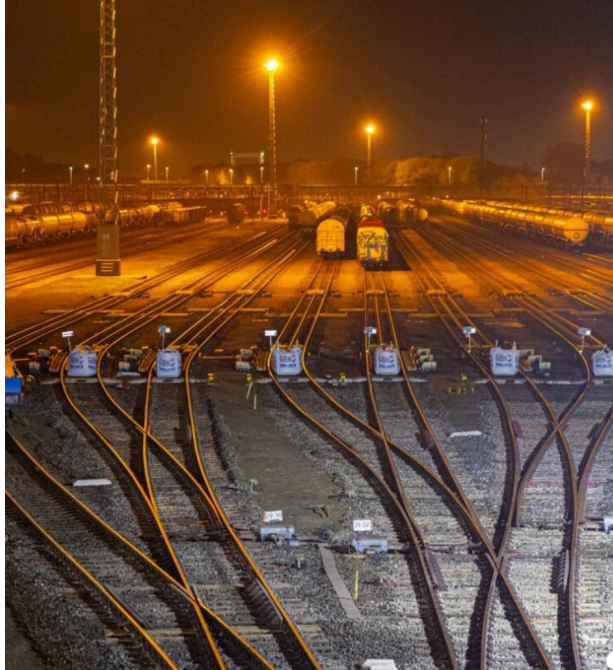


An example of failed test cases

As all API endpoints are used, we must ensure the test coverage is 100% (which means we tested all available endpoints) and the accuracy of every single test must be 100%, so that data can be posted, fetched and updated properly.

4.2 Frontend testing

For the frontend testing, we mainly conducted manual testing. Firstly, we run the command to start the front end. As soon as the index page shows up, we input the user account to test the log in.



Welcome Back

Email adress

user@struktio.nl

Password

Enter password

[Forgot Password?](#)

Sign In

Copyright©2025 | [Privacy Policy](#)

After successful login, we are able to view the main dashboard page with other different tabs. From then, we loop through each tab to test the functions manually.

Users are prompted to add a background image to overlay the yard for creation, which after that, the switches and tracks are also placed to make up a complete yard. We tested by adding the image and tried to add different switches and tracks to see if those components are properly added.

Regarding the analysis page, we fetched the data from the database and checked if those data are displayed completely on the frontend. Additionally, we also removed a part of data in the database to see if those contents are indeed removed from the frontend.

Chapter 5

Evaluation

5.1 Planning

Through the development of this project, we followed a structured plan in order to keep up with work. We spent the first two weeks getting in contact with the client and discussing requirements, after which we created and showcased the frameworks, which were approved. Weeks 4-9 were spent on development, and week 10 was spent on testing and polishing the final product. The team had stand ups once every two days, in order to check the progress and keep each other updated. Moreover, we met at least twice a week to work together. In the beginning, we lost some time due to miscommunication with the client and rather long reply times, but after we validated the requirements and had our questions answered, in week 3, the work ran smoothly.

5.2 Team Evaluation

We used Trello to keep up with tasks and tickets to explain the part we were working on and keep the others updated with the progress. There were some miscommunications as some teammates were working on the same task at times. In retrospect, we consider the communication rather poor, as we struggled to communicate our tasks effectively, separate concerns properly and keep up with the tasks at hand. Overall, it has been a learning experience, as we understood how improper time management and miscommunication can affect the workload and the final product. In the future, we will make sure to properly follow Agile and Scrum in teamwork, and always communicate with each other and mindfully divide tasks.

5.3 Communication With Client

In the beginning the communication was poor, as it took a while for our client to reach back to us to organize an initial meeting online, which we had in week 2, but did not fully answer our questions. However, we had an in-person meeting the following week, which explained most of our concerns and got the workflow going. We established then to have weekly meetings, on

Monday, with one every two weeks being in-person, and the other online. We met every week, showed our progress and had our questions answered. Apart from the setback in the first two weeks, communication went smoothly, and our client was pleased with the final product.

5.4 Final Product

Although our final product is less than expected, we still put in a lot of effort and our client is pleased with it. We hope the software team will manage to handle it properly and apply future improvements. Moreover, the proof of concept is adequate, and the company will be able to properly showcase their product to possible users.

Appendix

Database Schema

