

PROTOTYPE

DAT255 - GROUP TUGLIFE



Åkesson, Jonas	Lundgren, Emil
<code>jonasak@student.chalmers.se</code>	<code>emilundg@student.chalmers.se</code>

Nordenhög, Kevin	Nilsson, Sebastian
<code>nkevin@student.chalmers.se</code>	<code>sebnilss@student.chalmers.se</code>

Mrkjonc, Luka	Rasku, Kevin
<code>mrkonjic@student.chalmers.se</code>	<code>rasku@student.chalmers.se</code>



CHALMERS

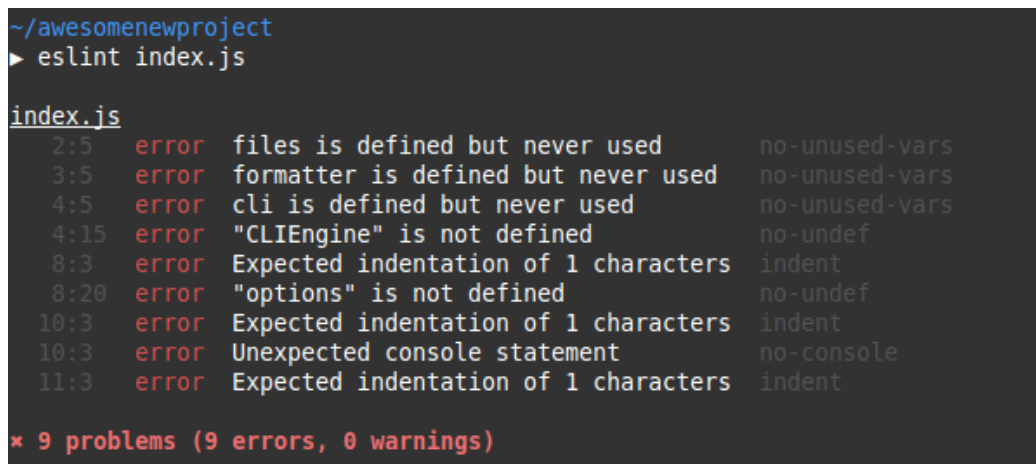
Contents

1	Code Quality	1
2	Unit, Integration & System tests	1
2.1	Unit testing	2
2.2	Integration Testing	2
2.3	System Testing	2
2.4	In Practice	2
3	Design rationale	3
3.1	General	3
3.2	In Practice	3
4	Overview	4
4.1	Behavioural & Structural	4
4.2	Protocol (client/server)	6
4.2.1	General	6
4.2.2	In Practice	6
5	User Stories	6

1 Code Quality

To maintain a high level of code quality is essential. If not maintained it could lead to unnecessary waste of money, time and resources dealing with the faulty code. Characteristics of good code quality include robustness, efficiency and reliability. Efficiency importance lies in the fact that slow software is not something desirable and it can be used as a means for measuring the software quality. To have a reliable code includes the fact that the software should work the same every time you use it and not crash at unexpected times. Robustness is quite similar to reliability but comes in hand when dealing with errors that shows.

There is no doubt that these factors are of crucial and essential meaning. The project was initialized with a Vue-CLI NodeJS template, with its base in Webpack. When initializing a project like this questions are asked on how you want to deal with certain criterias. An important criteria is code quality. We decided that we wanted ESLint to run with our code. ESLint is a pluggable linting utility similar to JSHint for JavaScript. Linting is the process of running a program that will analyze code for potential errors. This means that it has the ability to locate possible logical errors in the program. In practice this means that ESLint runs every time the code runs and if it finds any errors in syntax, unused variables or possible logical errors, the program will not run until the problems are fixed. This was used during the whole project and played a huge part in maintaining a high level of code quality. An usage example from their webpage can be seen in Figure 1.



```
~/awesomenewproject
> eslint index.js

index.js
  2:5  error  files is defined but never used      no-unused-vars
  3:5  error  formatter is defined but never used  no-unused-vars
  4:5  error  cli is defined but never used        no-unused-vars
  4:15 error  "CLIEngine" is not defined           no-undef
  8:3  error  Expected indentation of 1 characters  indent
  8:20 error  "options" is not defined              no-undef
 10:3  error  Expected indentation of 1 characters  indent
 10:3  error  Unexpected console statement          no-console
 11:3  error  Expected indentation of 1 characters  indent

✖ 9 problems (9 errors, 0 warnings)
```

Figure 1: Results from ESLint being used.

2 Unit, Integration & System tests

During the course a Software Quality lecture was held by Håkan Burden. This concluded that testing could be broken down to smaller parts, these are:

- Verification - Are we building the software right
- Validation - Are we actually building the right software
- Static - E.g Loop conditions, indices names and argument order
- Dynamic - E.g Build & Run, test-to-pass/test-to-fail and automatic/manual

The order of this is often unit testing, integration testing and finally system testing. Unit testing is the most local to the code and system testing is testing the final software, not looking at the code.

2.1 Unit testing

Unit testing involves testing small and individual source code sets or modules. It is one of the lowest levels of testing done to a software and is done by testing these sets together with data to check if they are suitable to use in the software. The whole idea behind it is to test certain units of the code and verify that they are functioning as expected by returning proper values. It helps us identify possible faults in the code and prevents excessive bugs.

2.2 Integration Testing

Can be divided into two different parts that needs to be able to cooperate, *order system* that uses an *inventory system*. The order system in this case being the system provided by us in the group and the inventory system (the system that is going to be used) being the collective server-backend distributed by PortCDM. The integration test then comes down to actually checking that the order system gives correct data to the inventory system.

2.3 System Testing

System testing includes testing the complete full software. One of the major points of this is to actually test that the application meets the specified requirements without actually having to look at the code behind it. It often comes after unit and integration testing and is one of the final testing stages. The first part of this type of test can be divided into verifying that the input given to the "system" gives the desired output. The second part involves testing if the user experience is the desired one. There are several different methods of performing system testing but one widely used is black box testing that tests both valid and invalid input to the software and check if the output is desired.

2.4 In Practice

The testing in practice was not done in a particularly structured way. The overall testing was more of the trial and error type of way. But with that said, different parts of the testing goes into different types of testing.

A lot of the testing done can still be applied and described by the types mentioned above. The unit testing done was mostly done when testing the routes of the app. It looked at internal routes to see if they returned the correct data. Usage areas for this was e.g testing that the internal server gave back the relevant data from the correct route. To even begin to test the cooperation between different files and the internal client/server communication console.log of a string was used. This serves as a basic mean to check if the code actually reaches the expected point. After the code reaching the correct place a test was made to see if the expected data would be sent and returned. This was done by logging the data that was sent and verifying that it was correct. The unit testing was completely done manually.

During the whole project, integration testing has been a big part. A major problem that arose when trying to integrate our local solution to a existing collective server-backend was that our app did not work. To find out why our app did not work, integration testing was done. The order system (our system) was in need of communicating with the inventory system (PortCDM backend). The testing was done through both our app but also with Restlet client which is a REST API testing service available for Google Chrome. The reason for testing the API-calls through their service was that it was much faster than writing the code and parsing the input. After successful tests the API-call could easily be integrated and written in our app. The results from the integration testing concluded that the collective server-backend used a newer version of the API-calls. In this newer version the API-call that the group had previously used no longer existed and the group had to rewrite a lot of code to get the app working again.

After validating that the integration worked it was time to perform system testing. It was done by parsing

data from the frontend, sending it to the backend where the correct API-call was formatted and sent to PortCDM. The requested data was then returned to the frontend. The tests were performed by inputting both valid data and also invalid data to check that errors were presented.

3 Design rationale

3.1 General

The design rationale includes a detailed description of the decisions and motivations made when designing the system. It also includes reasoning behind certain decisions, as well as discussing other alternatives. The goal behind it is to make the designers of the system get a clear picture of choices made during the design process.

3.2 In Practice

When designing a system that should meet the requirements of any project it is important to consider what factors and design choices are actually needed to be done. Our group concluded that a separate database was not needed because the application was not going to store so much information. This due to PortCDM already having their own backend with all the corresponding data needed to be fetched.

At first the application used server side rendering which means that it is rendered at the server and then sent to the client. The benefit of using server side rendering is that it is often considered to be much faster than the opposite, client side rendering, where the application is rendered at the client. In practice this meant that our applications API calls was made at the server, then rendered and sent to the client already containing the fetched data. This worked fine with doing GET-API calls to the PortCDM backend, but a problem arose when trying to take user input from the client and making a POST-API call to their backend, as it would not work. This due to the fact that the group was not at all used to doing server side rendering and that VueJS works best with client side rendering. So in practice the app would have to be rendered again after the user input (POST-API call) had been done. This did not work as expected and the group then switched to client side rendering. The final design can be seen in Figure 2.

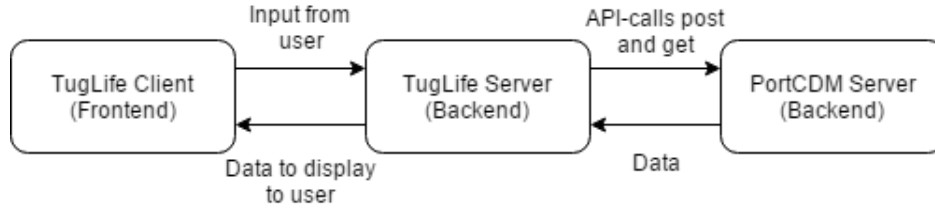


Figure 2: Design of TugLife application

The Tuglife client deals with displaying and retrieving information from the user. At startup, a GET API-call is made through the relevant routes to the PortCDM backend, where it retrieves information about the last portcalls made. The portcalls are then returned to the frontend where all mentions of tugboats will be filtered out and displayed in the table. The whole process is asynchronous which means that the app can work with other processes and is not waiting for this specific call to return anything.

The Tuglife server deals with incoming API-calls from the client-side with an `apiRouter`. The `apiRouter` does more than just routing API-calls. It also deals with parsing XML. The parsing is needed since the PortCDM backend is designed to receive requests in XML format. Own methods were created to deal with the parse the input correctly.

The chosen structure is common for web-applications and the benefit of using client-side routing to server-side routing to external API is that it provides a clear structure of the code and makes it easy to maintain a high quality of the code.

NodeJS contains a lot of external dependencies, so called Node Modules. The most relevant modules used for our application was `Axios`, which is an module for handling API-calls and furthered explained in Section 4.2.1, and `moment` which was used to convert and get correct time-stamps. The reason for using external dependency is clear when it comes to software development. It is unnecessary to reinvent the wheel.

4 Overview

4.1 Behavioural & Structural

Because of it being a single-page application the structure is quite simple. The initial parts that a user sees is a table of the tugboats that exist in the PortCDM backend. If no tugboat requests have been made from any party, the table is going to be empty. If there are tugboat requests in the backend, they are shown in the table with different relevant headers visible to the user as well as an update button at the side. An API call is made every five seconds to update the table continuously.

Below the table there are two areas. One area for making Location state calls and one for making Service state calls. The structure should be easy to follow for the user and highly relevant. At the final demo an employee at the tugboat company was present and reviewed the app and thought all relevant parts were visible and easy to understand. The main factor that differed from their current system was that their system did not have a function for retrieving requests and displaying them. The communication was instead held through VHF or phone and was then manually registered and answered to in their system. The final application can be seen in Figure 3.



Press update on a tug boat to update its location or service state.

Vessels	Service Object	Performing Actor	Time Sequence	Time	Type	Location	Update
um.mrn:atm:vessel:IMO-9261401	ESCORT_TOWAGE		COMMENCED	03/09/2017, 08:24	ACTUAL		UPDATE
um.mrn:atm:vessel:IMO-9261401	ESCORT_TOWAGE		COMPLETED	03/09/2017, 09:40	ACTUAL		UPDATE

Change Location State

Vessel ID

Reported By

Reference Object

Time

Time Type

Arrival Location

Departure Location

Post

Statuscode:

Change Service State

Vessel ID

Service Object

Time Sequence

Time

Time Type

At Location

Between: To Location

Between: From Location

Post

Statuscode:

Figure 3: The main page of the application

4.2 Protocol (client/server)

4.2.1 General

The application uses a Node package for dealing with API-calls called Axios. It is used to make XMLHttpRequests from the browser, make http requests from node.js, transform data and and so on. The sole purpose of using it is to simplify the process of making API calls. XMLHttpRequests uses different objects to interact with servers. The data is retrieved from an URL and is often done without having to update the whole page. This means that it is highly reactive because it is only refreshing the relevant part. The requests are used to fetch any type of data, not only XML.

4.2.2 In Practice

As described above in the design rationale, the client makes the relevant and corresponding API calls to a client sided route. The API call is then routed to the server route which makes the external API calls to PortCDM. Routing like this makes the structure clear and easy to follow.

5 User Stories

Our prototype was built on predefined user stories which were set by the product owner. Our goal was to fulfill as many of these as possible in our final product, which we did, as all of the set user stories were implemented. But there were many problems on the way, one being that we misinterpreted one of our user stories and developed a whole new portal for booking tugboats, which actually was the ship agents job. This misinterpretation slowed down our overall development and took valuable time from our project.

The user stories in our project were:

- As a ship agent I want to be able to check if I can start booking a tug boat so that the vessel can use it
- As a tugboat I want to be able to communicate when towage is commenced so that other parties know the status
- As a tugboat I want to be able to communicate when I am booked so that others know
- As a pilot I want to call the tug boat on VHF to be able to decide where to meet so that the tug boat and vessel can meet at a specified location
- As a tugboat I want to communicate when I am approaching a berth so that the personell can be ready
- As a ship agent I want to be able to book a tug boat at least five hours before the vessel arrives so there's enough time to tow the vessel to the berth station
- As a tugboat I want to communicate when I disconnect so that the status can be set to complete and others know
- As a tugboat I want to be able to know what berth to approach so that personnel from the port and the moorers can be ready