Profitability Calculator

Technical Documentation

Author: Tharindu Ranaweera

Version Number: v1.0

Date: 04/10/2023

Table of content

Introduction

Project Overview

Profitability Calculator is a software solution aimed at delivering reliable calculations for the transport industry. The software will take away the overhead of manual calculations during the tendering process. The Profitability Calculator application will calculate the total distance based costs, total time based costs, total costs and the profitability of the transport. The web application is implemented to take inputs from the user and for the result display purposes.

Purpose of the document

Purpose of this technical documentation is to give an in-depth idea of how the software solution is designed, implemented and tested in all the ends.

Objectives

The objectives of this project are to calculate the following metrics

- Total distance based costs.
- Total time based costs.
- Total costs.
- Profitability of the quotation.

The above metrics are calculated based on the following inputs.

- Cost per kilometre.
- Number of kilometres.
- Cost per hour.
- Number of hours.
- Profitability.

The user inputs will be recorded from the web application interface and the calculations will be done in the web server. The results will be displayed in the same web application.

Scope

In Scope

- Functionality to create user accounts.
- The user accounts should be stored in a database securely with hashed passwords.
- Ability to log into created user accounts with credentials.

- Ability to calculate total distance based costs, total time based costs, income and profitability when the inputs are provided correctly.
- The users should not be able to access the calculator without proper authentication and JWT authorization.
- Backend calculate API should not be able to be used without proper authorization.
- Proper input validation in both frontend and backend.
- The web application should support localization.
- The application should be properly tested in the frontend and the backend.
- Unit tests and end to end tests should be conducted.
- The web application should be responsive to the screen it is viewing.
- A proper documentation should be provided along with the swagger documentation.
- An Android app should be developed and the same backend must be used.
- A development build will be provided as per this revision.

Out Scope

- Support for an iOS application.
- Setting a profit margin will not be supported under this revision of the application.
- Caching will not be supported under this version of the application.

Assumptions

- The user inputs (cost per kilometre, number of kilometres, cost per hour, number of hours, income) to be calculated accurately.
- The calculation formulas the application uses are accurate (Refer the following formulas).

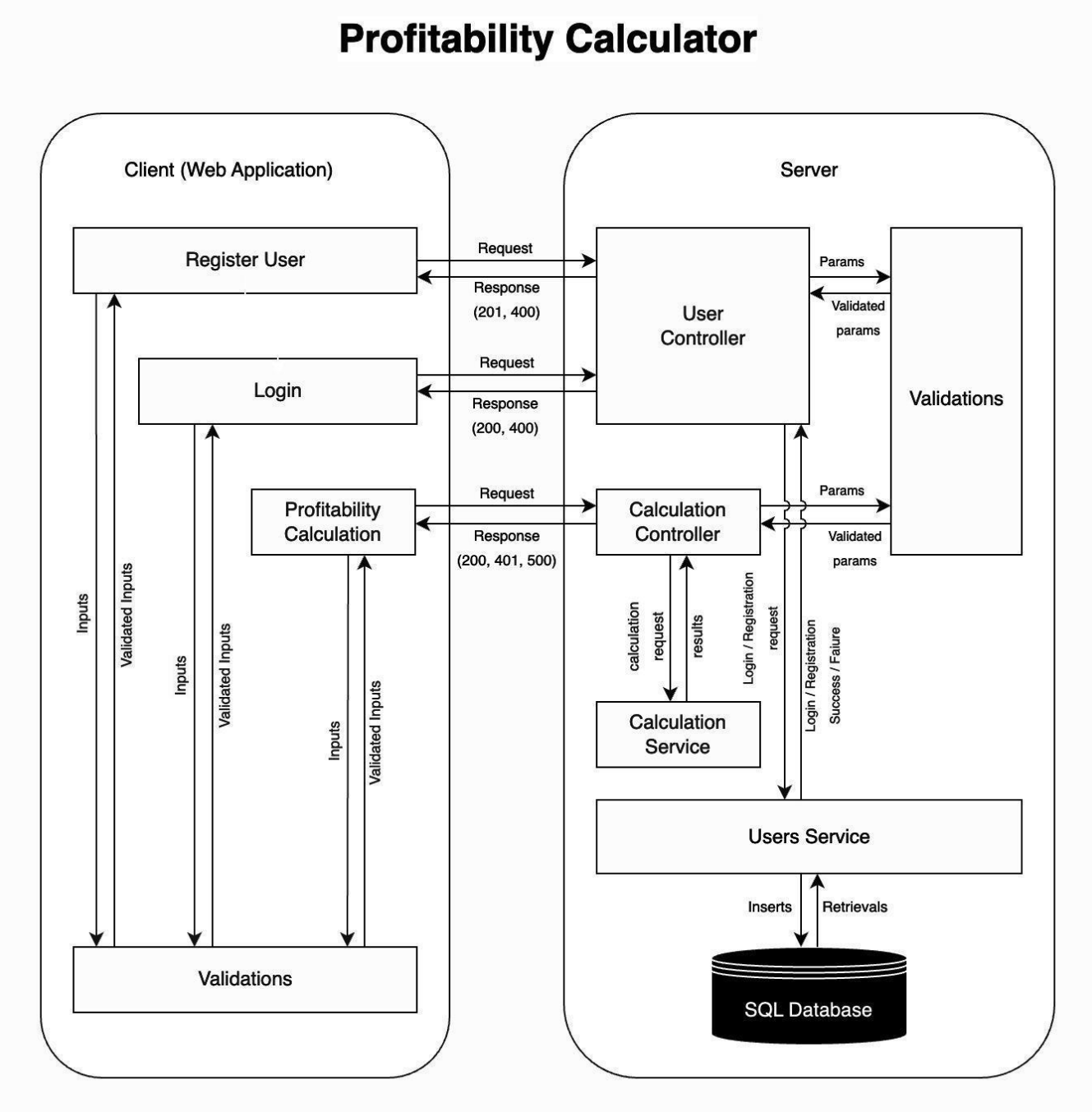
1) Total Distance Based Costs = (Price Per Kilometre) x (Number of Kilometres)
2) Total Time Based Costs = (Price Per Hour) x (Number of Hours)
3) Total Costs = (Total Time Based Costs) + (Total Distance Based Costs)
4) Profitability = (Income) - (Total Costs)

Risks

- The learning curve of Vue.js, ASP.Net, xUnit, Cypress will have an impact on the delivery on time.

Product Overview

High level architecture



Requirement Analysis

Functional Requirements

FR-01: User registration

Requirement	The user should be able to create an account in the system providing full name, username and password.

Acceptance Criteria	- Users should be successfully registered in the system. - Proper validations must be performed in frontend and backend. - Multiple users shouldn't be created with the same username.

FR-02: User login

Requirement	The user should be able to log into the system with an existing account.

Acceptance Criteria	- Users should be successfully logged into the system allowing the user to access the calculator. - Proper validations must be performed. - Response should contain a jwt token which is later used for authorization.

FR-03: Profitability Calculation

Requirement	The user should be able to calculate total distance based costs, total time based costs, income and profitability.

Acceptance Criteria	- The calculation should be successfully done with correct calculations. - The user inputs must be validated in the frontend and backend. - Unauthorised requests should be failed.

FR-04: Internationalisation Support

Requirement	The application should support internationalisation.

Acceptance Criteria	- The application must be easily switchable between languages.

Non-Functional Requirements

NFR-01: Security

Requirement	- The user data should be securely stored. - Calculation API should be properly secured.

Acceptance Criteria	- The password should not be stored as plain text in the database. - The calculation requests should fail if a proper token is not included in the request header. - User's should not be able to access protected screens without login.

NFR-02: Responsive design

Requirement	The application should work in any window size.

Acceptance Criteria	- The application should be responsive to wide screens. - The application should resize respectively to mobile screens.

NFR-03: Usability

Requirement	The application should be user friendly so that people can use the application with a small amount of computer literacy.

Acceptance Criteria	- The required information should be clearly displayed. - The error messages should be descriptive. - Responsive error messages should be displayed. - Calculation results must persist on refreshes. - Page routing must be coordinated.

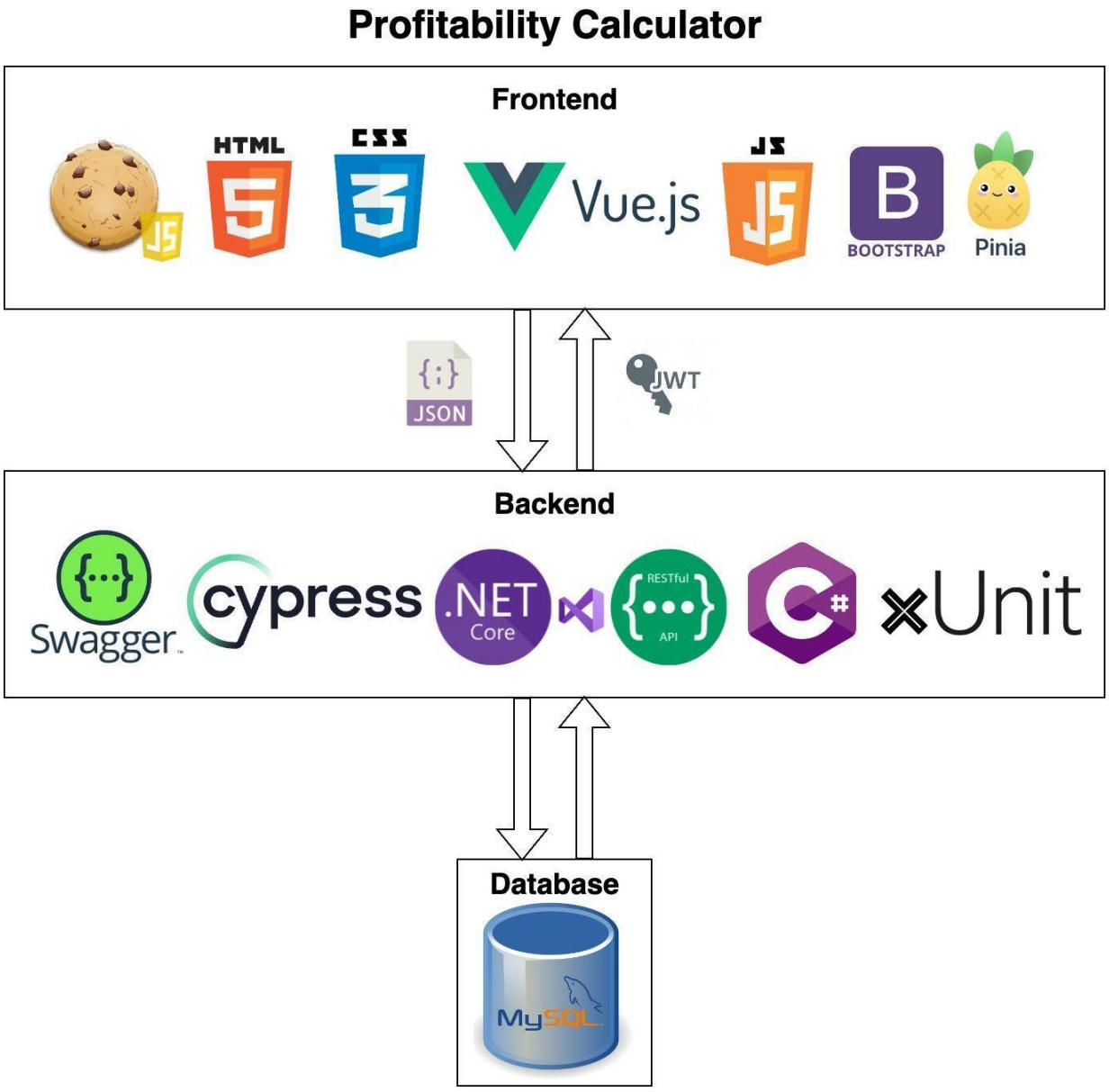
NFR-04: Availability

Requirement	Both backend and frontend applications should be always available.

Acceptance Criteria	- Proper exception handling must be done to avoid crashing.

Implementation

Technology Stack



Prerequisites

In order to run the development version of this software solution the following prerequisites are needed.

- Microsoft .NET version 6
- Node version 18
- npm version 9.5 or above
- MYSQL
- Xamarin Android 12.1
- Xamarin iOS 16.0

Implementation

User registration

Backend implementation

Request class

Class Name	`UserSignUpRequest`

Properties	`Name: String`
	`Username: String`
	`Password: String`

Entity class

Class Name	`User`

Properties	`Name: String`
	`Username: String`
	`PasswordHash: byte[]`
	`PasswordSalt: byte[]`

Response class

Class Name	`UserSignUpResponse`
Properties	`Name: String` `Username: String`

Controller Method

Method Name	`SignUp`
Method Type	`POST`
Parameters	`request: UserSignUpRequest`
Returns	`response: UserSignUpResponse`
Response Types	`[201: Created()],[400: BadRequest()],[500: InternalServerError]`
Response	`{ username, name }`

Service Method

Method Name	`async CreateUser`
Parameters	`request: UserSignUpRequest`
Returns	`user: Task<User>`

Util Methods

Method Name	`CreatePasswordHash`

Parameters	`password: String`
	`out passwordHash: byte[]`
	`out passwordSalt: byte[]`

Returns	`void`

Description

Once the request reaches the endpoint, the **`SignUp`** method should validate the input.

Validation 1: username and password should not be empty.
Validation 2: username must be at least 4 characters long.
Validation 3: password must be at least 8 characters long. At least one capital character must be included. At least one number must be included.

Then the **`CreateUser`** service method should get called. This method will take cake of database communication with the database context class.
The method will check for existing users with the same username. If not it will create hashed passwords and save it to the database.

Successful user insertion will create a **`201: Created`** response with **`name`** and **`username`** **`(UserSignUpResponse)`** as JSON payload. Any error in the class would result in **`400: BadRequest`** or **`500: InternalServerError`**.

Frontend Implementation (Web)

View Name	`SignUpPage`

State Data	`name: String`
	`username: String`
	`password: String`
	`re-password: String`

Props	-

Components	`LanguageSelector`
	`LoadingButton`
	`InputBox`
	`Button`

Methods	`onSubmit(e)`
	`validateInputs()`
	`matchUsername()`
	`matchPassword()`
	`routeToLogin()`

Description

This is a form view which shows 4 input components with a submit button. All the fields are required fields. Validations are implemented in the frontend.

Validation 1: Checks if all inputs are non-empty.
Validation 2: **`username`** should be at least 4 characters.
Validation 3: **`password`** should be at least 8 characters. At least one upper case letter and a number should be included.
Validation 4: password and re-password values should be an exact match.

Once the inputs are successfully validated, the values should be included into a single **`JSON`** object which should be sent to the server via a fetch request after stringification.

Proper alerts should be displayed in case of an error.
The response statusCode should be checked and the user should be redirected to the **`LoginPage`** in a successful registration allowing the user to log in.

Mobile Implementation (Android)

The mobile screen for the registration feature is also similar to the above web frontend implementation. It consists of four Entry fields which accept the **`name`**, **`username`** and **`password`**. All the fields are properly validated under the same set of rules as above.

Once the validations are successful, a **`POST`** request will be sent with the validated date. If an error occurs during the whole process, **`AlertDialog`** should get displayed with the reason.

User Login

Backend implementation

Request class

Class Name	`UserLoginRequest`
Properties	`Username: String` `Password: String`

Entity class

Class Name	`User`

Properties	`Name: String`
	`Username: String`
	`PasswordHash: byte[]`
	`PasswordSalt: byte[]`

Response class

Class Name	`UserLoginResponse`
Properties	`Username: String,` `Token: String`

Controller Method

Method Name	`Login`
Method Type	`POST`
Parameters	`request: UserLoginRequest`
Returns	`response: UserLoginResponse`

Response Types	`[200: Ok()],[400: BadRequest()],[500: InternalServerError]`
Response	`{ username, token }`

Service Method

Method Name	`async SignIn`
Parameters	`request: UserLoginRequest`
Returns	`user: Task<String>`

Util Methods

Method Name	`VerifyPasswordHash`

Parameters	`password: String`
	`passwordHash: byte[]`
	`passwordSalt: byte[]`

Returns	`Boolean`

Method Name	`CreateToken`
Parameters	`user: User`
Returns	`String`

Description

The above endpoint should receive **`username`** and **`password`** which the user's trying to authenticate with. It should call the service method **`SignIn`**. The **`SignIn`** method should look for an existing user with the provided **`username`** in the database.

If found, the method will hash the provided password and match with the stored hashed password. If the process succeeds, The **`CreateToken`** method will create a JWT token. A perfect scenario will cause **`200: Ok`** response with a **`UserLoginResponse`** (**`{ username, token }`**).

Frontend Implementation (Web)

View Name	`LoginPage`
State Data	`username: String` `password: String`
Props	-

Components	`LanguageSelector`
	`LoadingButton`
	`InputBox`
	`Button`

Methods	`onSubmit(e)` `validateUserInputs()`

Description

The form takes 2 inputs: username and password. The **`validateUserInputs`** will validate the inputs.

Validation 1: None of the inputs can be empty.

Once the inputs are validated, the values will be objectified and sent to the server via a **`POST`** request after stringification.
Once the response is received, the **`statusCode`** is checked and if the status is **`200`**, the user should be navigated towards the calculator screen. If an error status is returned an alert should be displayed.

Persisting user login

`js-cookie` library is used to persist user data. Once a successful login is done, the response body should contain a **`JSON`** with **`username`** and **`token`** value.

Both these values must be stored in a cookie. The token value must be used in the authorization header as the token bearer in the upcoming calculation requests.

Routing

`vue-router` is used for router implementation. Without a proper login the user should not be able to navigate to the calculator screen, even when using the specific route (**`<url>/calculator`**). It should be redirected to the login page.

Mobile Implementation (Android)

The login screen in **`Xamarin Forms`** was implemented the same as the web implementation. It consists of two Entries for **`username`** and **`password`**. The same set of validations were conducted for the mobile application to keep the consistency between the two.

After the validations are fully met, the application will send the request in order to log the user in. After a successful login request is made, a response with the username and the JWT token will get sent towards the client. The username and jwt token is saved in the Android secure storage. The saved token will get retrieved and used for the authorization in the calculator page.

Profitability Calculation

Backend implementation

Request class

Class Name	`ProfitabilityCalculationRequest`

Properties	`PricePerKilometre: Double`
	`PricePerHour: Double`
	`NoOfHours: Double`
	`NoOfKilometres: Double`
	`Income: Double`

Entity class

Class Name	`ProfitabilityCalculation`

Properties	`Id: String`
	`PricePerKilometre: Double`
	`PricePerHour: Double`
	`NoOfKilometres: Double`
	`NoOfHours: Double`
	`Income: Double`
	`TotalDistanceBasedCosts: Double`
	`TotalTimeBasedCosts: Double`
	`Profitability: Double`

Response class

Class Name	`ProfitabilityCalculationResponse`

Properties	`Id: String`
	`PricePerKilometre: Double`
	`PricePerHour: Double`

	NoOfKilometres: Double NoOfHours: Double Income: Double TotalDistanceBasedCosts: Double TotalTimeBasedCosts: Double Profitability: Double

Controller Method

Method Name	CalculateProfitability
Method Type	POST
Parameters	request: ProfitabilityCalculationRequest
Returns	response: ProfitabilityCalculationResponse
Response Types	[200: Ok()],[400: BadRequest()]
Response	{ id, pricePerKilometre, pricePerHour, noOfKilometres, noOfHours, income, totalDistanceBasedCosts, totalTimeBasedConsts, profitability }

Service Method

Method Name	CalculateProfitability
Parameters	profitabilityCalculation: ProfitabilityCalculation
Returns	response: ProfitabilityCalculationResponse

Method Name	CalculateTotalDistanceBasedCosts
Parameters	pricePerKilometre: Double noOfKilometres: Double
Returns	totalDistanceBasedCosts: Double

Method Name	CalculateTotalTimeBasedCosts
Parameters	pricePerHour: Double noOfHours: Double
Returns	totalTimeBasedCosts: Double

Description

The **`CalculateProfitability`** controller method acts as the endpoint. It's an **`[Authorize]`** endpoint. Which means the user must be properly authorised to use the API. An unauthorised request will result in **`401: Unauthorized`** just even before entering the endpoint.

Once a successful request lands there, the entity class will be initialised and will get passed to the service method.
Just before calculation, service methods will check for validation.

Validation 1: None of the fields can be empty.
Validation 2: None of the fields should contain negative values.

Once all the validation passes through, the values will get calculated and finally the response object will get returned.

Frontend Implementation (Web)

View Name	`CalculatorView`

State Data	`pricePerKilometre: String`
	`pricePerHour: String`
	`noOfHours: String`
	`noOfKilometres: String`
	`income: String`
	`isLoading: Boolean`

Props	-

Components	`LanguageSelector`
	`LoadingButton`
	`InputBox`
	`Button`

Methods	`onSubmit(e)` `routeToLogin()`

Description

The CalculatorView is also a form screen which records 5 above inputs from the user. The user inputs are validated from the InputBox component itself.

Validation 1: All inputs must be non-empty.
Validation 2: All inputs must be strictly numbers.
Validation 3: All inputs must be non negative double values.

Once all above fields are validated, they are sent to the server with a POST request for calculations. The auth token should be extracted from the pinia state and added as the authorization header before the sending the request. Then the response should be recorded and the statusCode should be checked.

In a scenario where the stored token is expired, the server will return 401: Unauthorized which will cause the web app to log out automatically.

Data Persisting

The calculated results should be available even after a browser window refresh. For that purpose pinia-persist npm package is used. The returned values are saved in the pinia store and also saved in the session storage.

Viewing Results

CalculationResultView is implemented to view the calculated results after successful calculation. The values should have already been saved in the pinia store and also the values must be stored in the session storage in case of tab refresh.
If the user goes back to the calculator, the old data should be cleared from the session storage. The results view should not be accessible if the results are not available.

Mobile Implementation (Android)

A form screen with 5 user entries is used to do the calculator implementation on Xamarin Forms following up with the same validations to retain consistency between the applications.

Once the inputs are validated, the saved jwt token is retrieved from the Secure Storage and added to the request header to authorize the endpoint. Once the response is received, the calculation results are saved to the Shared Preference and gets retrieved in the results screen.

Internationalisation support

The Profitability Calculation application supports internationalisation using **`Vuei18n`** library. As per this version it supports English and Finnish languages for all the screens and alerts. The default locale is set to English and the user is presented with a dropdown to select language. Once a language is selected, the screen should get re-rendered with the new locale strings.

Data persistence

In order to support data persistence during tab refresh for locale, the selected locale is stored in the session storage. If the locale is not present, the application will pick the default locale English.

Reusable components

InputBox component

This component represents the user input component and this component is highly customisable. It manages its own state and has the support for properties like

- Size of the component
- Validation support

PopUpAlert component

This component represents a reusable pop up modal component. It's also highly customisable. It takes the props like message, button text and a function which gets executed on button click. This component is placed at the root component and is getting triggered using pinia state actions.

LanguageSelector component

LanguageSelector consists of a dropdown component which helps the users to change the locale.

Header component

Header component consists of the log out button and language selector component.

Button component

Button component represents an HTML button which is styled according to the application theme. This accepts a function which gets executed on button click.

InputError component

This reusable component helps the developers to show an error on the form without using an alert box. This is the component used inside InputBox to show real-time input errors.

Testing and Quality Assurance

Unit testing

Backend unit testing

Backend unit testing was done using xUnit Nuget package. The ProfitabilityCalculator.Tests project consists of all the tests done for the profitability calculation. The tests are conducted for the following scenarios.

- Total distance based costs calculation

- Total time based costs calculation
- Total costs calculation
- Profitability calculation

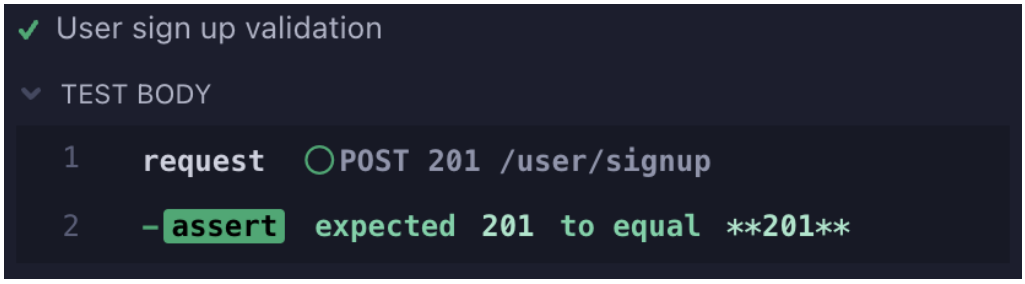
All the above tests have been conducted with possible scenarios and impossible scenarios which cause errors. All the exceptions are properly tested and handled.

End to end testing

The end to end testing has been conducted through the cypress node library. All the endpoints have been properly tested. Refer to the following scenarios and test results.

User sign up

Action: User sign up API is called with proper inputs.
Expected: User to be registered successfully and receive 201:Created response.
Response: User successfully created and received 201: Created.
Result: PASS

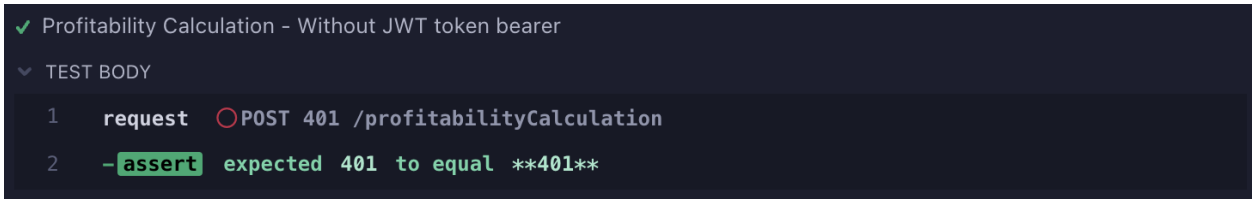


User login

Action: The login API was called with the above created user credentials.
Expected: User to authenticate successfully and receive 200:Ok with username and token.
Response: User successfully authenticated and received 200: Ok with username and token.
Result: PASS

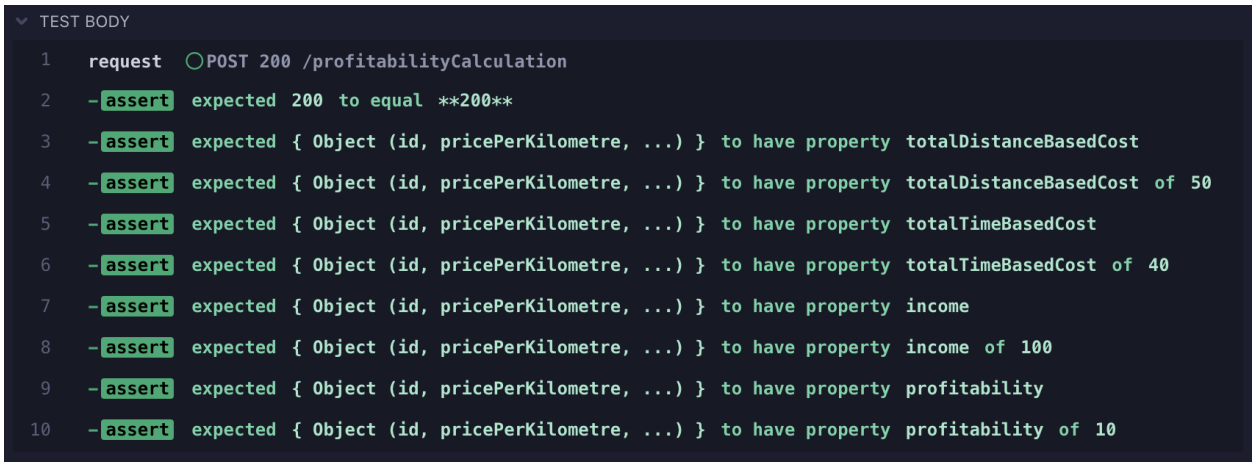
Profitability calculation without bearer token

Action: The login API was called with proper request body but without bearer token.
Expected: The request to fail with 401: Unauthorized
Response: The request failed with a response of 401: Unauthorized.
Result: PASS



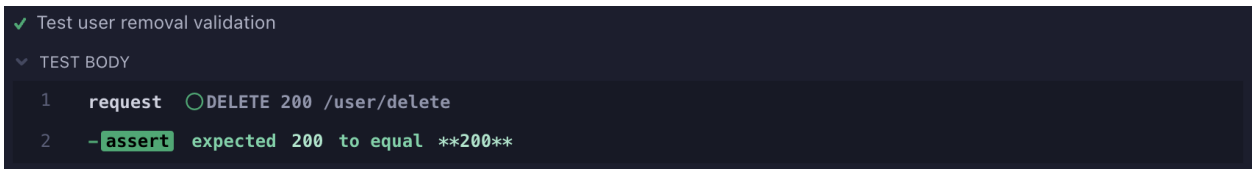
Profitability calculation with proper bearer token

Action: The login API was called with a proper request body with a valid bearer token.
Expected: The request to succeed with 200:Ok with proper calculated results.
Response: The request succeeded with 200:Ok and received correct calculation results.
Result: PASS



Removal of created user by end to end tests

Action: The login API was called with a proper request body with a valid bearer token.
Expected: The request to succeed with 200:Ok and the user to be removed.
Response: The request succeeded with 200:Ok and the user was removed from the system.
Result: PASS



Failure scenarios have been properly tested and added to the cypress.

Manual QA Testing

Test Case 1: User Registration

Scenario 1: Registration with empty values

Preconditions	-
Steps	- User navigates to signup screen - User presses Sign Up button - User fill everything except one field - User again presses Sign Up button
Expected Result	The operation should not be successful
Result	**PASS**

Scenario 2: Registration with weak username

Preconditions	-
Steps	- User navigates to signup screen - User enters “jan” as the username - User fills everything else perfectly - User presses Sign Up button
Expected Result	Alert should popup describing the issue and how to fix it.
Result	**PASS**

Scenario 3: Registration with weak password

Preconditions	-
Steps	- User navigates to signup screen - User enters “password” as the password - User fills everything else perfectly - User presses Sign Up button
Expected Result	Alert should popup describing the issue and how to fix it.
Result	**PASS**

Scenario 4: Registration with unmatched password and re-password

Preconditions	-
Steps	- User navigates to signup screen - User enters the password correctly (Password@123) - User leaves re-password as “Password@124” - User fills everything else perfectly - User presses Sign Up button
Expected Result	Alert should popup describing the issue and how to fix it.
Result	**PASS**

Scenario 5: Registration with complete information

Preconditions	-
Steps	- User navigates to signup screen - User fills everything perfectly - User presses Sign Up button
Expected Result	The user should be registered successfully and prompted the success. And then the user should be directed to the login screen.
Result	**PASS**

Test Case 2: User Login

Scenario 1: Login with empty fields

Preconditions	-
Steps	- User navigates to login screen - User presses Login button without filling anything - User fills the username without password and vice versa. - User presses Login button
Expected Result	The operation should fail
Result	**PASS**

Scenario 2: Login with incorrect credentials

Preconditions	-
Steps	- User navigates to login screen

	- User fills incorrect user credentials. - User presses Login button
Expected Result	The operation should fail and prompt the issue with an alert.
Result	**PASS**

Scenario 3: Login with correct credentials

Preconditions	- User to be registered first
Steps	- User navigates to login screen - User fills correct credentials - User presses Login button
Expected Result	The user should be able to successfully login and be directed to the calculator.
Result	**PASS**

Test Case 3: Profitability Calculation

Scenario 1: Calculating with empty inputs

Preconditions	- User to be logged into the system.
Steps	- User logs into the system. - User presses the Calculate Profitability button without filling inputs. - User fills everything leaving an input empty. - User presses the Calculate Profitability button
Expected Result	The operation should fail.
Result	**PASS**

Scenario 2: Calculating with invalid inputs (using letters)

Preconditions	- User to be logged into the system.
Steps	- User logs into the system. - User fills all the inputs but with wrong values (Include letters). - User presses the Calculate Profitability button
Expected Result	The fields must show error. Even after pressing the Calculate button an alert should popup with the error.

Result	**PASS**

Scenario 3: Calculating with invalid inputs (using negative values)

Preconditions	- User to be logged into the system.
Steps	- User logs into the system. - User fills all the inputs but with wrong values (Include negative values). - User presses the Calculate Profitability button
Expected Result	The fields must show error. Even after pressing the Calculate button an alert should popup with the error.
Result	**PASS**

Scenario 4: Calculating with valid inputs

Preconditions	- User to be logged into the system.
Steps	- User logs into the system. - User fills all the inputs correctly. - User presses the Calculate Profitability button
Expected Result	The operation should succeed and the user should redirect towards the results page.
Result	**PASS**

Test Case 4: Data persistence test

Scenario 1: Refreshing results page

Preconditions	- User to be logged into the system.
Steps	- User logs into the system. - User fills all the inputs correctly. - User presses the Calculate Profitability button - <User is presented with the > - User refreshes the page.
Expected Result	The calculation results must persist.
Result	**PASS**

Scenario 2: Close tab and navigate to calculator in a new tab

Preconditions	- User to be logged into the system earlier.
Steps	- User logs into the system. - User closes the browser tab. - User reopens a new tab and navigates to the calculator. - The user should be automatically logged in.
Expected Result	The calculation results must persist.
Result	**PASS**

Test Case 5: Application routing test

Scenario 1: Routing to calculator screen without login

Preconditions	- User should not be logged in
Steps	- User tries to goto calculate page with url <baseurl>/calculator
Expected Result	The Calculator screen should not be available for the user. The user should be redirected towards the login page.
Result	**PASS**

Scenario 2: Routing to login/register screens after successful login

Preconditions	- User should be already logged in
Steps	- User tries to goto login/register page with url <baseurl>/signup or <baseurl>/login
Expected Result	The user should be redirected towards calculator
Result	**PASS**

Scenario 3: Routing to results screen before calculation

Preconditions	- User should be already logged in
Steps	- The user tries to goto results screen with url <baseurl>/results before doing a calculation.
Expected Result	The user should be redirected towards calculator
Result	**PASS**

Future implementation

- Support for an iOS app.
- Internationalisation support for Mobile.
- Improved testing for Mobile.
- Unit test support for Web application.
- Implementing unit tests for frontend application with Jest.
- Support for setting a profit margin.
- Caching for calculations on the server.

Conclusion

The course of this project is to develop a web application which can be extremely useful for the transport industry. The application can be used to calculate the costs based on distance and time as well as the total costs. The profitability calculation is also a key feature of this application.

The purpose of this document is to provide an overall high level understanding about the design and implementation of the project and the way it is tested. The project was successfully completed with all the core features as well as most of the optional features. The uncompleted tasks are displayed in the above future implementation section and will be completed in the next revision of this application.