

|  |
| --- |
| **VILNIAUS KOLEGIJA**  **ELECTRONICS AND INFORMATICS FACULTY**  **SOFTWARE DEVELOPMENT DEPARTMENT** |
| **PROFESSIONAL PRACTICE REPORT**  6531BX028 PI18E |
| |  |  | | --- | --- | | PRACTICE PLACE: | UAB Visma | |
| |  |  |  | | --- | --- | --- | | COMPANY ADVISOR | 2021 - 06 - 21 | AURIMAS CHOLOMSKIS | |
| |  |  |  | | --- | --- | --- | | STUDENT | 2021 - 06 – 21 | DŽIUGAS PEČIULEVIČIUS | | FACULTY ADVISOR | 2021 - 06 - 21 | DR. ROMANAS TUMASONIS | |
| 2021, VILNIUS |

**TABLE OF CONTENTS**

[1. PRACTICE ASSIGNMENT FORMULATION 5](#_Toc74943552)

[1.1. Deep problem description 5](#_Toc74943553)

[1.2 Functional requirements 5](#_Toc74943554)

[1.3. Non-functional requirements 6](#_Toc74943555)

[2. PRACTICE ASSIGNMENT ANALYSIS 8](#_Toc74943556)

[2.1.1. Analysis on functional requirements for eCommerce CRUD application. 8](#_Toc74943557)

[2.1.2. Analysis on functional requirements for a wizard stepper application. 11](#_Toc74943558)

[3. SOFTWARE 15](#_Toc74943559)

[3.1 Main software files for CRUD eCommerce application 15](#_Toc74943560)

[3.2 Main software files for Wizard Stepper application 20](#_Toc74943561)

[4. USER GUIDE 27](#_Toc74943562)

[4.1 User guide for CRUD eCommerce application 27](#_Toc74943563)

[4.2 User guide for CRUD eCommerce application 32](#_Toc74943564)

[CONCLUSIONS AND FUTURE STEPS 36](#_Toc74943565)

[LITERATURE 37](#_Toc74943566)

**TABLE OF FIGURES**

[Figure 1 - Use case diagram for functional requirements of CRUD application 8](#_Toc74943588)

[Figure 2 - Use case diagram for wizard application 11](#_Toc74943589)

[Figure 3 – Folder structure for CRUD application 16](#_Toc74943590)

[Figure 4 - App component for CRUD application 17](#_Toc74943591)

[Figure 5 - Routing file for CRUD application 17](#_Toc74943592)

[Figure 6 - Homepage component hyper text markup language file 18](#_Toc74943593)

[Figure 7 - Homepage typescript component 19](#_Toc74943594)

[Figure 8 - Wizard routing component 20](#_Toc74943595)

[Figure 9 - Main wizard component typescript file 21](#_Toc74943596)

[Figure 10 - Main wizard component html file 22](#_Toc74943597)

[Figure 11 - Wizard component typescript file 23](#_Toc74943598)

[Figure 12 - Wizard component html file 24](#_Toc74943599)

[Figure 13 - Stepper component 25](#_Toc74943600)

[Figure 14 - SASS files 25](#_Toc74943601)

[Figure 15 - SCSS variables 26](#_Toc74943602)

[Figure 16 - SCSS mixins 26](#_Toc74943603)

[Figure 17 - Homepage of a CRUD application 27](#_Toc74943604)

[Figure 18 - Product cards 27](#_Toc74943605)

[Figure 19 - Edit product form. 28](#_Toc74943606)

[Figure 20 - Edit product form. 28](#_Toc74943607)

[Figure 21 - Success alert 29](#_Toc74943608)

[Figure 22 - Edited product 29](#_Toc74943609)

[Figure 23 - Delete product alert. 30](#_Toc74943610)

[Figure 24 - Deleted product example 30](#_Toc74943611)

[Figure 25 - Add new product form. 31](#_Toc74943612)

[Figure 26 - Added product example. 31](#_Toc74943613)

[Figure 27- Wizard Shipping Form 32](#_Toc74943614)

[Figure 28 - Wizard shipping form filled. 33](#_Toc74943615)

[Figure 29 - Wizard payment form 33](#_Toc74943616)

[Figure 30 - Wizard payment form filled. 34](#_Toc74943617)

[Figure 31 - Wizard review page 34](#_Toc74943618)

[Figure 32 - Wizard review page confirmed. 35](#_Toc74943619)

[Figure 33 - Wizard success page 35](#_Toc74943620)

**INTRODUCTION**

Professional practice was performed at UAB Visma that took a total of 8 weeks from May 3rd 2021 to June 25th 2021 as a front-end developer role. The company is one of the Europe ‘s leading software companies and offers software/services that simplify core business processes in the private and public sector. Visma has diversity of projects – from small applications to large J2EE (Java 2 Enterprise Edition) systems.

During the internship time at the company, the focus was to successfully complete given tasks, learn how to write clean, tested and maintainable code and learn what are the best practices are for developing front-end, more specifically Angular (front-end framework) applications. For the first two weeks of the internship all of us were assigned a task to make plain front-end application without using any frameworks and test it using Jest. At Visma, front-end developers either use Angular or Vue. After analyzing our competence, we were allocated to a specific team. Our goal was to learn best practices and make a several small applications to become familiar with the framework and become job ready in 6 weeks.

Given internship tasks:

* To create an eCommerce CRUD application using only HTML, CSS, and JavaScript.
* To become familiar with Jest and Jasmine.
* To recreate an eCommerce CRUD application using JavaScript framework – Angular. Also, to test the application and strive for at least 80% of total application code coverage.
* To create wizard application which handles user forms, tracks user progress, and saves entered user information.

# 1. PRACTICE ASSIGNMENT FORMULATION

## 1.1. Deep problem description

So, every front-end developer’s role is to be responsible for implementing visual and interactive elements that user engages with through their web browser. Developers write code and create components from very simple UI elements like buttons to complex user forms. Front-end developers are also responsible for connecting front-end code to back-end by handling HTTP requests to the server and displaying responses in the user interface of the application.

During this internship, the focus was to become familiar with JavaScript framework – Angular, so we would be comfortable working with it further on the job after the internship. Basically, Angular team at the beginning assigned interns to do a CRUD application to see how well we manage working with Angular and test the application to get familiar with Angular testing syntax and tools which are Jasmine and Karma in this case. Later in the internship, Angular team assigned us some tasks that needed to be done for their codebase. One of them was a form wizard application, which handles user forms, tracks user progress, lets user review entered data and later when submitted, it makes a request to a backend with saved user information. The team seeks for at least 80% code coverage in their codebase, so every component on this application had to be tested with minimum of 80% coverage.

# 1.2 Functional requirements

Almost half of the time we were testing our applications, other part of the time we were writing code and building our apps. Since there were few applications we built, I am going to write requirements for all of them.

Further on, these are functional requirements for our first CRUD application that we developed while learning Angular:

1. The system should allow users to view products available.
2. The system should allow users to create a new product and add it to a list.
3. The system should let users delete any chosen item and show a confirmation dialog if user really wants to delete selected item.
4. The system should allow users to edit any given item information.
5. The application should also display a carousel component showcasing trending items.

And now these are functional requirements for a wizard application that we did later in at the internship:

1. The system should allow user to enter shipping details information.
2. The system should allow user to enter card details information.
3. The system should not allow user to continue to the next page if there are empty non-filled fields or entered data is invalid.
4. The system should display a stepper component and show user where in the process exactly user currently is.
5. At the last step of the process, system displays entered user data. To submit, user must check the checkbox and confirm that entered information is correct.
6. At the last step of the process, system should display user all the entered information and once confirmed, it should send a request to the backend and redirect user to a success page.

## 1.3. Non-functional requirements

Non-functional requirements for a CRUD application:

1. The application should be built with JavaScript framework – Angular.
2. The application should use TypeScript.
3. API should be mocked with json-server to create mock endpoints and the website should run using http-server with a proxy parameter to allow for same domain access to the database running on a different port.
4. All data should be fetched from the API.
5. Editing item should be done on a different page.
6. Deleting items should have a confirmation dialog.
7. Application should be responsive and should show screen size accordingly.
8. Text should be readable for all screen sizes.
9. Application should have at least two primary and secondary colors, should also have colors for backgrounds, borders, and for buttons such as error or success buttons.
10. It should display user a spinner (loading indicator) if any data is being fetched or page is being refreshed now.

Non-functional requirements for a Wizard application:

1. Application should be written in Angular.
2. Application should use TypeScript.
3. Application should use reactive forms which allows to track changes with minimal effort.
4. Should track user steps and how many there are in total using observables (observables are a technique for event handling, asynchronous programming, and handling multiple values).
5. The application should be responsive.
6. Text should be readable for all screen sizes.
7. Application should have a success page to redirect user once the process is complete.
8. It should have a button to go back if a user needs to tweak the information a little bit.

# 2. PRACTICE ASSIGNMENT ANALYSIS

## 2.1.1. Analysis on functional requirements for eCommerce CRUD application.

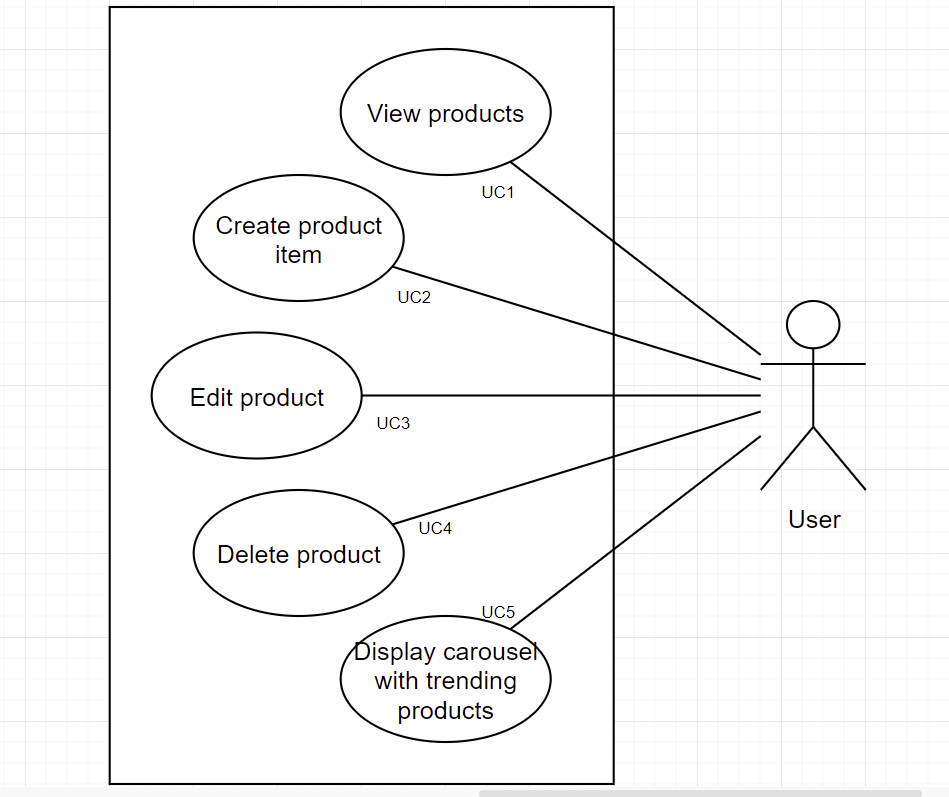


Figure 1 - Use case diagram for functional requirements of CRUD application

Use case diagram for functional requirements of CRUD application.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **FUNCTIONAL REQUIREMENT** | **USER REQUIREMENT** | **USE-CASE** | **PRIORITY** |
| FR1 | View products | User | UC1 | 1 |
| FR2 | Create product item | User | UC2 | 2 |
| FR3 | Edit product | User | UC3 | 3 |
| FR4 | Delete product | User | UC4 | 3 |
| FR5 | Display carousel trending products | User | UC5 | 4 |

Make sure that the functional requirements are formulated correctly: they are easy to understand, clear, unambiguous, implementable, verifiable.

Describing the use cases:

|  |  |
| --- | --- |
| **Function** | View Products |
| **ID** | UC1 |
| **Description** | User can see existing product |
| **User role** | User |
| **Pre-conditions** | There is an internet connection |
| **Inputs** | Go to the screen where data is fetched and displayed |
| **Outputs** | Product list |
| **Action** | 1. The use enters the website |
| **Post-conditions** | After entering the website, user should see loaded products |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Create product item |
| **ID** | UC2 |
| **Description** | User can create a product |
| **User role** | User |
| **Pre-conditions** | There is an internet connection |
| **Inputs** | Product name, price, image URL and description. (Product ID is automatically generated) |
| **Outputs** | Adds item to the product list |
| **Action** | 1. The use enters the website. 2. Fills create product input fields. 3. Confirms and clicks ‘Add’. |
| **Post-conditions** | After entering product data, the new item should appear just above the input form right next to all the existing components. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Edit product item |
| **ID** | UC3 |
| **Description** | User can edit a product |
| **User role** | User |
| **Pre-conditions** | There is an internet connection |
| **Inputs** | Product name, price, image URL and description. |
| **Outputs** | Adds edits existing item |
| **Action** | 1. The use enters the website. 2. Chooses the product user wants to change. 3. Makes changes to product data in the form. 4. Confirms and clicks ‘Edit’. |
| **Post-conditions** | After making and confirming changes, it should confirm that changes have been made and redirect user to the main page user was before. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Delete product |
| **ID** | UC4 |
| **Description** | User can delete a product |
| **User role** | User |
| **Pre-conditions** | There is an internet connection |
| **Inputs** | Click delete button |
| **Outputs** | Deletes product from the product list |
| **Action** | 1. The use enters the website. 2. Chooses the item user wants to delete. 3. Clicks delete. 4. Clicks confirm to delete the product. 5. Deletes product from the product list. |
| **Post-conditions** | After deleting the item, it should delete the item and make changes in the UI in real time. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Display popular products in the carousel |
| **ID** | UC5 |
| **Description** | User can see the popular products in the carousel |
| **User role** | User |
| **Pre-conditions** | There is an internet connection |
| **Inputs** | Go to the screen where product and carousel data is being fetched and displayed |
| **Outputs** | Displays carousel component. |
| **Action** | 1. The use enters the website. |
| **Post-conditions** | After entering the website, user should see loaded carousel data. |
| **Side effects** | None |

## 2.1.2. Analysis on functional requirements for a wizard stepper application.

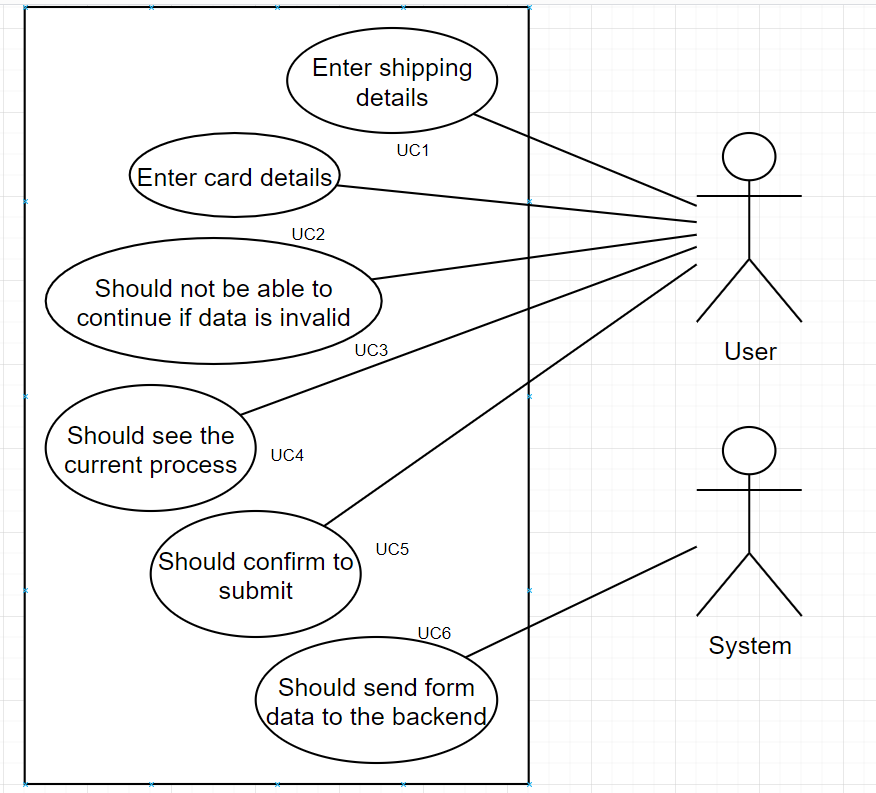


Figure 2 - Use case diagram for wizard application

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **FUNCTIONAL REQUIREMENT** | **USER REQUIREMENT** | **USE-CASE** | **PRIORITY** |
| FR1 | Enter shipping details | User | UC1 | 1 |
| FR2 | Enter card details | User | UC2 | 1 |
| FR3 | Should not be able to continue if data is invalid | User | UC3 | 3 |
| FR4 | Should see the current process | User | UC4 | 4 |
| FR5 | Should confirm to submit | User | UC5 | 2 |
| FR6 | Should send form data to the backend | System | UC6 | 2 |

|  |  |
| --- | --- |
| **Function** | Enter shipping details |
| **ID** | UC1 |
| **Description** | User can enter shipping details |
| **User role** | User |
| **Pre-conditions** | 1. There is an internet connection. 2. The page is reached by being redirected from other page to /form component. |
| **Inputs** | Enter name, surname, email, phone number, and address details |
| **Outputs** | If everything is filled, ‘Next’ button is enabled so user can continue to the next step. |
| **Action** | 1. The gets redirected to /form URL. 2. User enters the data needed for the form. |
| **Post-conditions** | Next button is enabled, and user can continue to the next step which is card details page in this case. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Enter card details |
| **ID** | UC2 |
| **Description** | User can enter card (payment) details |
| **User role** | User |
| **Pre-conditions** | 1. There is an internet connection. 2. The page is reached by being redirected from other page to /form component. 3. And user needs to pass previous (shipping step) |
| **Inputs** | Enter name, surname, card number, expiration month, expiration year and CVV. |
| **Outputs** | If everything is filled, ‘Next’ button is enabled so user can continue to the next step. |
| **Action** | 1. The gets redirected to /form URL. 2. User enters the data needed for the form. |
| **Post-conditions** | Next button is enabled, and user can continue to the next step which is review details page in this case. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Should not be able to continue if data is invalid |
| **ID** | UC3 |
| **Description** | User should not be able to continue to the next step or submit the form if any data is invalid. |
| **User role** | User |
| **Pre-conditions** | 1. There is an internet connection. 2. The page is reached by being redirected from other page to /form component. |
| **Inputs** | No data is entered |
| **Outputs** | Since there is no data entered. Then the user is not able to continue. |
| **Action** | 1. User gets redirected to /form URL. 2. Does not enter any data into the input fields. |
| **Post-conditions** | Next or Finish buttons are disabled until forms are not filled. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Should see the current step of the process. |
| **ID** | UC4 |
| **Description** | User should see the current step at the top on the wizard application during the process. |
| **User role** | User |
| **Pre-conditions** | 1. There is an internet connection. 2. The page is reached by being redirected from other page to /form component. |
| **Inputs** | Views the page |
| **Outputs** | Stepper component is shown |
| **Action** | 1. User gets redirected to /form URL. |
| **Post-conditions** | User can see the stepper component showing the current step and completed ones if there is any. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Should confirm to submit. |
| **ID** | UC5 |
| **Description** | At the review page, user should check the checkbox that user confirms, that entered data is all correct. |
| **User role** | User |
| **Pre-conditions** | 1. There is an internet connection. 2. The page is reached by being redirected from other page to /form component. 3. Last step of the process is reached. |
| **Inputs** | Checks the checkbox |
| **Outputs** | Finish button is enabled |
| **Action** | 1. User gets redirected to /form URL. 2. User goes through the process. 3. User confirms that data is correct |
| **Post-conditions** | User can now click ‘Finish’ button and send out the form to the backend to complete the order. |
| **Side effects** | None |

|  |  |
| --- | --- |
| **Function** | Should send form data to the backend |
| **ID** | UC6 |
| **Description** | At the review page, once user clicks finish, system should handle the request, and send it to the backend. |
| **User role** | System |
| **Pre-conditions** | 1. There is an internet connection. 2. User fills and confirms forms are correct. |
| **Inputs** | Users clicks ‘Finish’ |
| **Outputs** | System handles the request and send out the form. |
| **Action** | 1. User confirms that data is correct and send out the form. 2. Systems handles the data and makes a POST request to the backend (server). |
| **Post-conditions** | User data is now sent and stored in the backend. |
| **Side effects** | None |

# 3. SOFTWARE

The hardest part of this internship coming from a JavaScript background, was trying to write TypeScript code. TypeScript is a strict syntactical superset of JavaScript, and it adds optional static typing to the language. Types provide a way to describe the shape of an object, providing better documentation, and allowing TypeScript to validate that your code is working correctly. Basically, TypeScript shows error in real time, and it helps to catch bugs before running the code in the browser, otherwise like with JavaScript you had to run the code on the browser and would only then see if something is not right with the code.

## Main software files for CRUD eCommerce application

Angular applications are unique not just because it uses TypeScript, has better tooling but it has a higher scalability. The way angular applications are structured, it makes it easy to get going and simple to maintain large applications. This is something how angular application folder structure looks like:

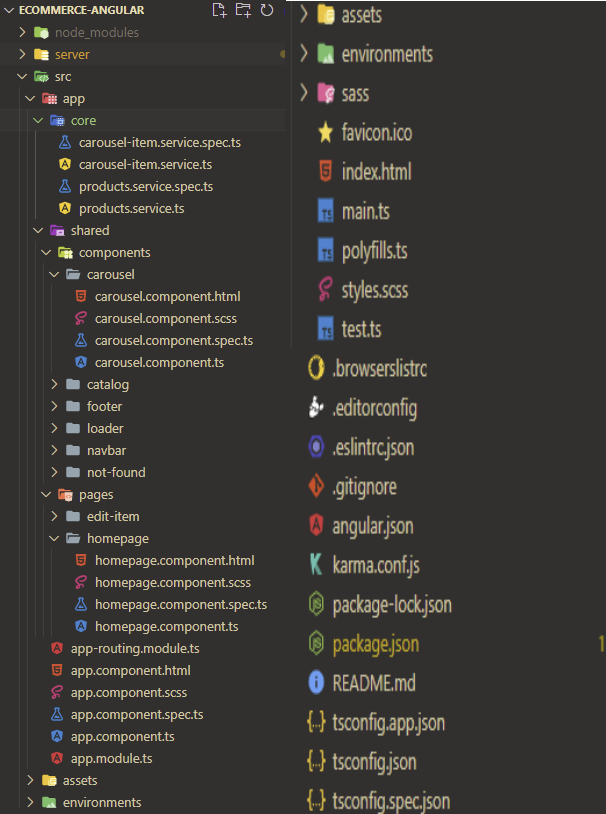


Figure 3 – Folder structure for CRUD application

This is how Angular eCommerce app folder structure looks like. On the left we can see our core/service files, components, pages, and main app component. I will go further into details later what each file is responsible for. On the right we can see a lot of styling and configuration files for angular, testing and typescript.

So, when we run our Angular application using a command `ng serve` or `ng serve -o` to open the project in new window, the application. Once the application is running on localhost, the first file that runs is the main app.component file.

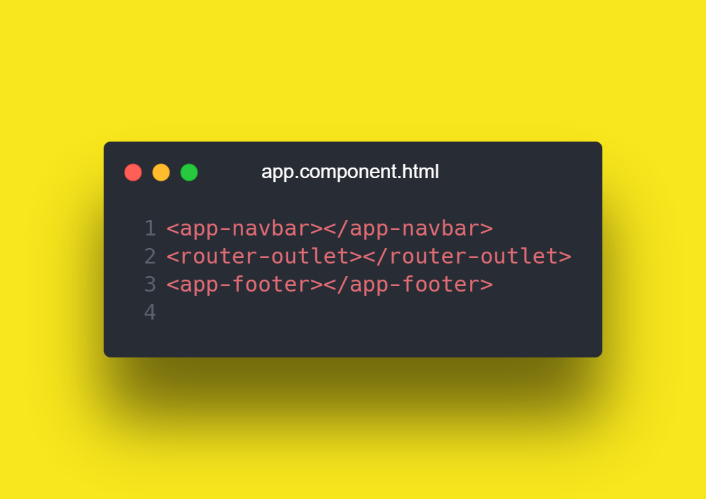


Figure 4 - App component for CRUD application

So since the app component is ran first and it is the root component of the application, we want to define components that are going to be replaced across the application like navbar and footer and everything that is going to change, we want to put them into the router outlet component.

What is the router outlet? It is basically a component where we can write our routes that is going to visit, and once the route is visited, the page renders the component accordinly, it none of the routes are found, then the application renders a “not found” component. This how “app-routing.module.ts” file looks like:



Figure 5 - Routing file for CRUD application

Once routes are loaded, we can look at our first component that is loaded and user sees.



Figure 6 - Homepage component hyper text markup language file

Basically, we have a homepage component, with all the necessary data here. At the top we can see our data loaded from the backend which are carousel and all the product items, then can see form container being used, this is a form below all the components that a user can use to add new products to a list, and at the bottom of the page we write our templates which in this case we use for a loading reference, so we can reuse it and load a loading component whenever it is needed. This is only the markup of the application, for an actual logic, we write everything in a TypeScript component file.



Figure 7 - Homepage typescript component

On the typescript file, at the top as always, we import all the necessary files and libraries needed. @Component let’s us define a selector how this component can be reached in our project by other components. Then inside the component, at the top we write our data fetch methods so we can pull needed data. After that we can write down our reactive form that user uses to enter new product data. Reactive forms let us easily manage form inputs. In the constructor we inject our needed dependencies and services. And below that we write down our methods responsible for each action.

## Main software files for Wizard Stepper application

Now that we moved on to the wizard app, we can have a look at how the application looks first. So just like the CRUD application, this app has a simple routing component:



Figure 8 - Wizard routing component

In this case we have additional line to tell the application to redirect to a form component if any of the written routes do not exist in the URL tab. So once again, the main route here is /form which brings user to FormPageComponent.



Figure 9 - Main wizard component typescript file

Form page component is responsible for all the forms that application contains, and a method that submits the data and redirects user to a success page. This application uses reactive forms so it is easy to manage forms in the application. This is a part of html code for this component:



Figure 10 - Main wizard component html file

We can also see that this component uses another component called „app-wizard“. That component contains all the logic for this wizard application. It also contains „wizardSumit“ button, but as we can see, we passed it to a parent component so it has a better reusability, because we don‘t need to pass form data anymore to other components and we can just send them easily as we saw in the example earlier.

So now to the logic itself. This is Wizard component itself.

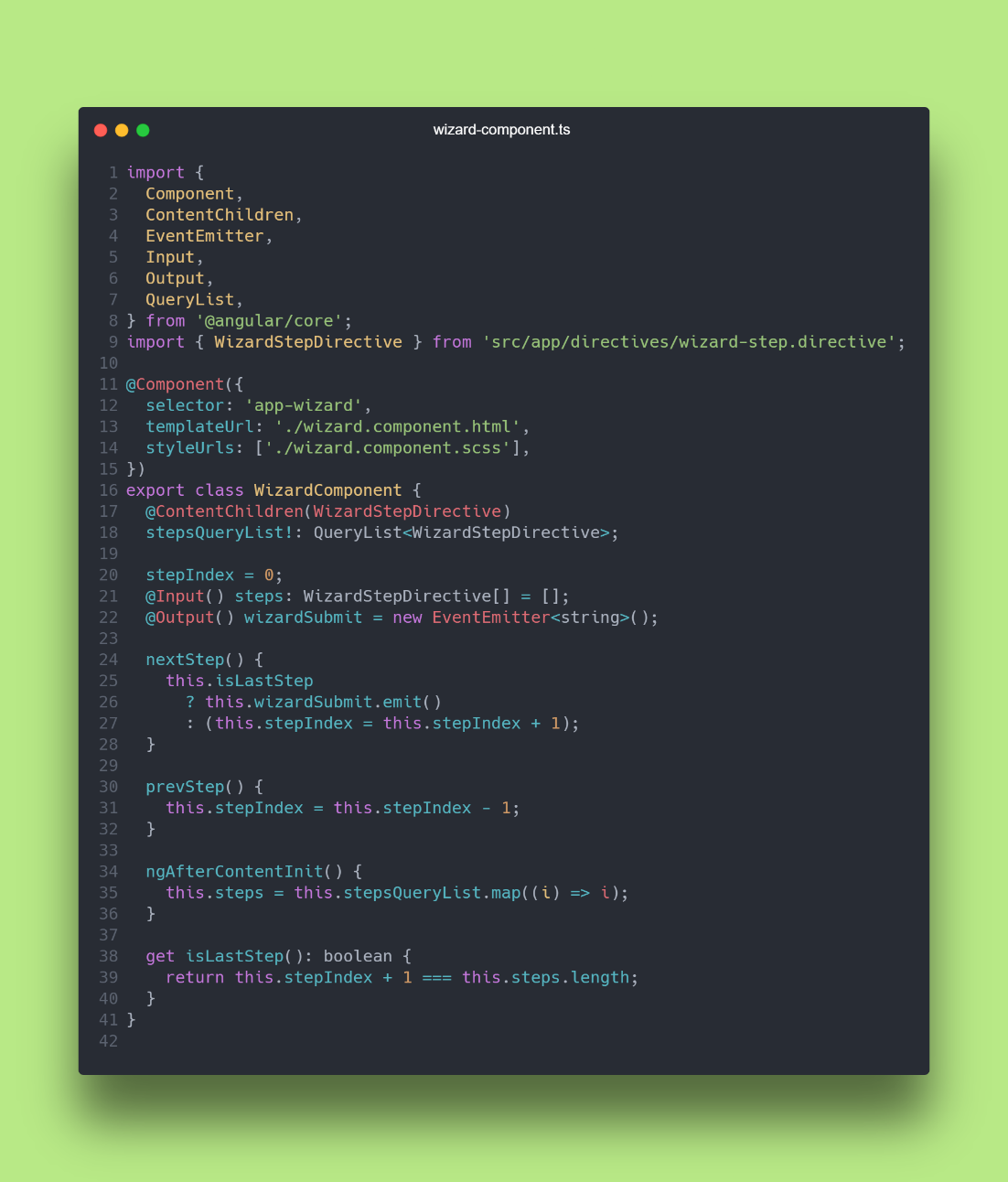


Figure 11 - Wizard component typescript file

In here we can see that all the logic. At the top we pull all the steps from form component, and define methods for buttons. Method ‚ngAfterContentInit()‘ loads all the steps into the component once it was loaded. And we use get methods such ‚isLastStep()‘ in this example to make calculations such as getting a step count through map or calculating something more complex. Now that we got the functionality out of the way, we can have a look how the html lokos like for the buttons that user can interact with:



Figure 12 - Wizard component html file

At the top of the component, we use another component which displays user steps on which step user currently is. Then going down we can use structural directives such as \*ngFor to loop through our list. In this case we look through our list and map each step. Basically ng-container displays our data within each step. And moving forward, we have our buttons that user uses. For previous button, we want to hide if user is on the first step, so we use \*ngIf directive and if user is on step 0, then we hide the component. And for the next button, we pass in an invalid disabled prop which disables continue button, if any of the forms are invalid, user missed entering. And also, we check if it is the last page then we set button label to “Finish”, else we display “Continue”.

Also, not to forget, this wizard component, uses a stepper component at the top, so let’s see what the stepper component holds in itself.



Figure 13 - Stepper component

Well, stepper component is basically responsible only for the UI element indicating the current step user is at. It will check what step user is at and will apply a CSS class for the component. It also displays a step index and a step name.

For application styling, I and Angular teams at the company, use SCSS, which is Syntactically Awesome Style Sheet. It allows us to write all kinds of stuff with CSS. We can define variables, do calculations, create animations and we can split all of that into different .scss files for easy management. This is what SCSS looks like in this project.

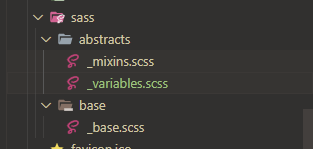


Figure 14 - SASS files

This is just a very few of those style files. Each component has one each scoped.



Figure 15 - SCSS variables

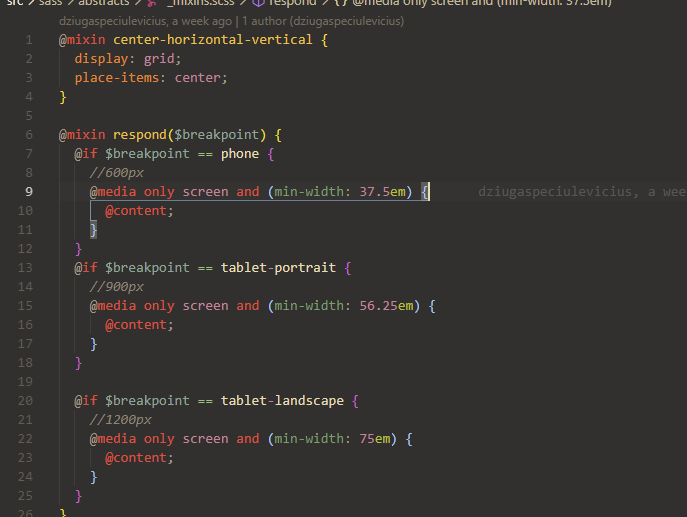


Figure 16 - SCSS mixins

Basically, we can see how easy it is to maintain the code. We have all our colors in one of the files, and all the CSS related to page size, we put into another component.

# 4. USER GUIDE

## User guide for CRUD eCommerce application

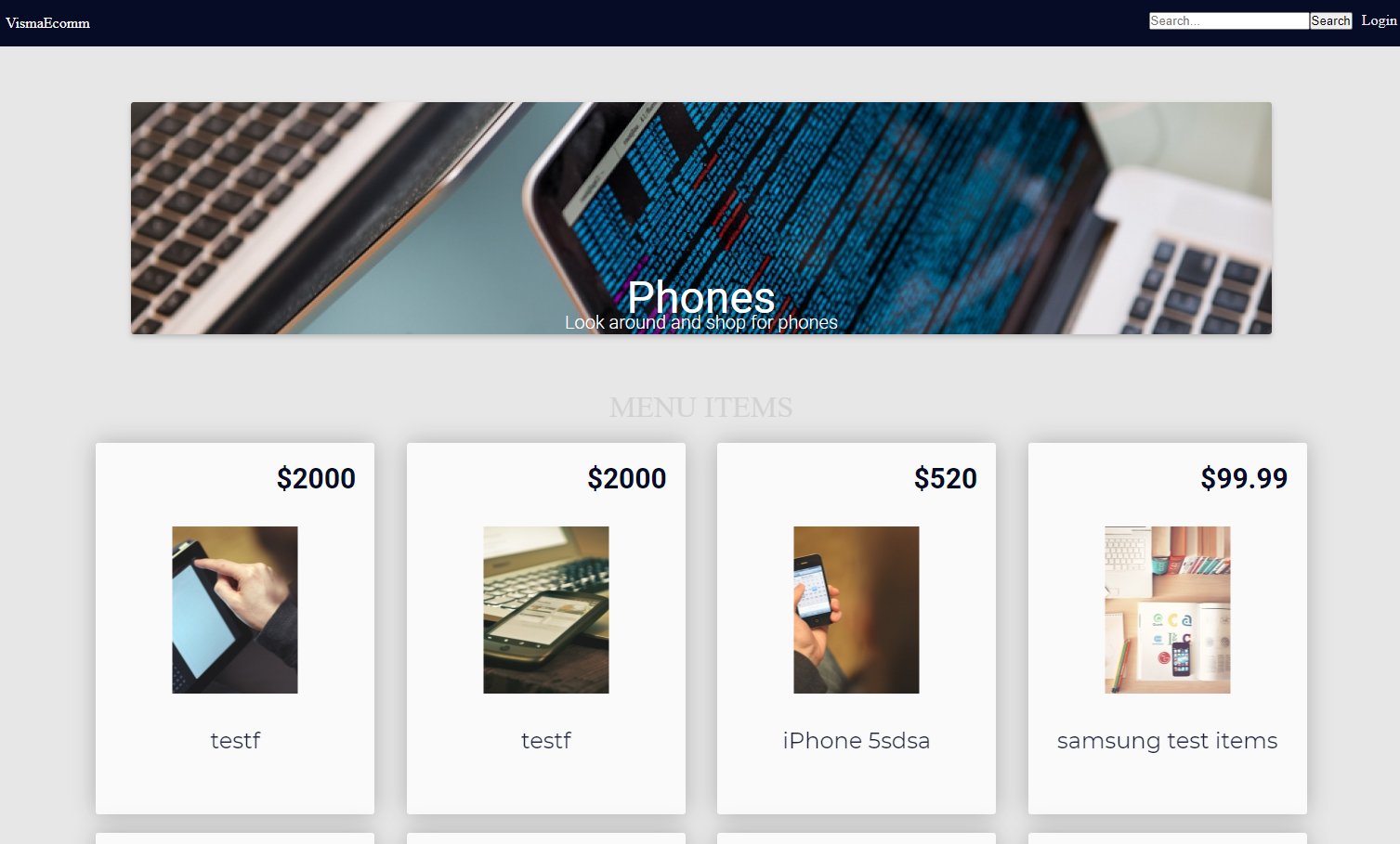


Figure 17 - Homepage of a CRUD application

While visiting the page, user is greeted with this kind of a homepage. User can see the carousel and the items. While hovering over the item, user can choose one of the two options. Either to edit the item or to delete it. Let’s see what the edit functionality does first.

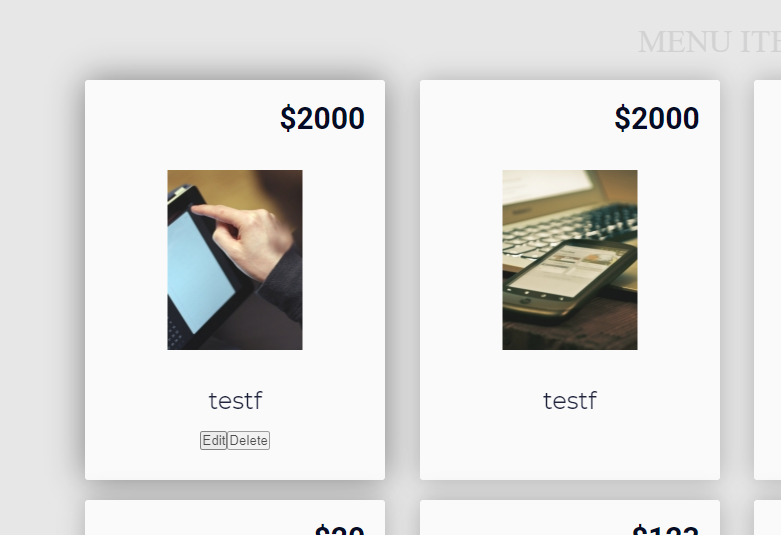


Figure 18 - Product cards

Once user click on the edit button, user is redirected to a page where you can see a form with existing product data. You cannot change it while data has not been changed.

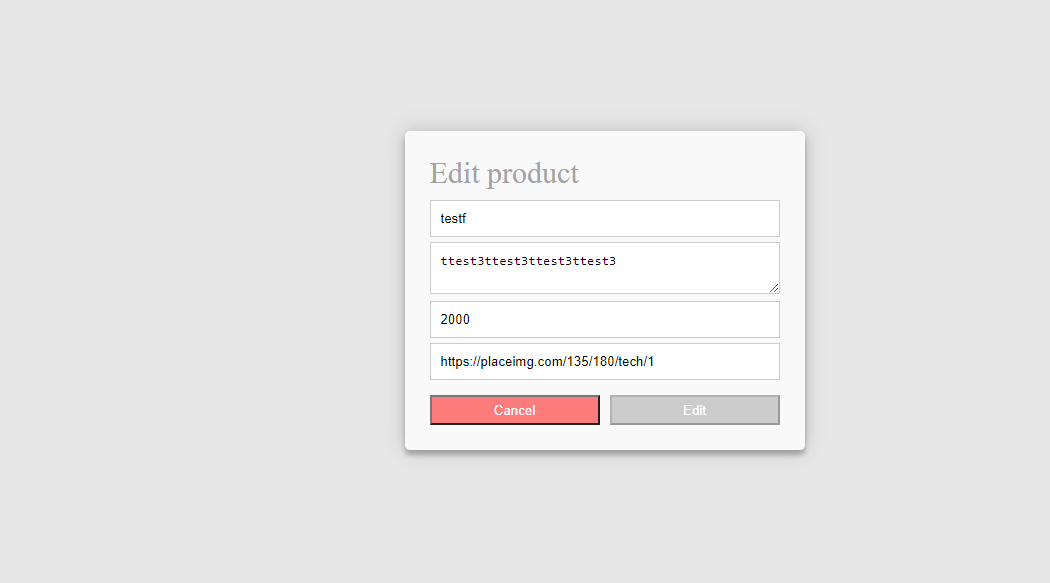


Figure 19 - Edit product form.

Once user changes the data, edit button activates.

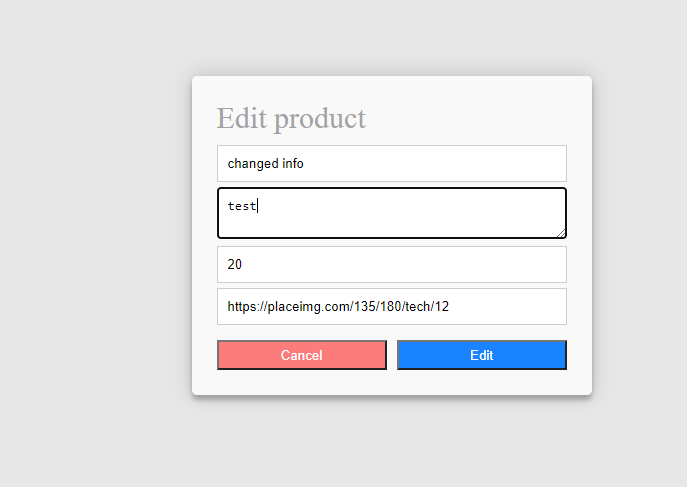


Figure 20 - Edit product form.

After confirming changes, user is shown an alert message that data has been changed.

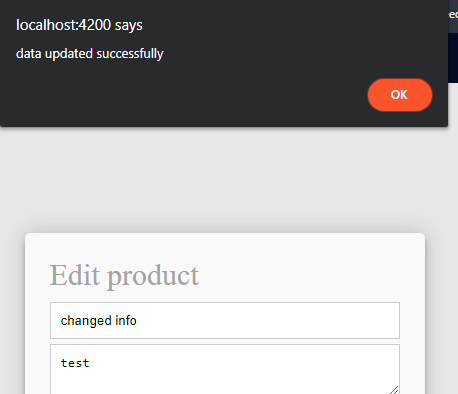


Figure 21 - Success alert

And here we can see the changes being affected.

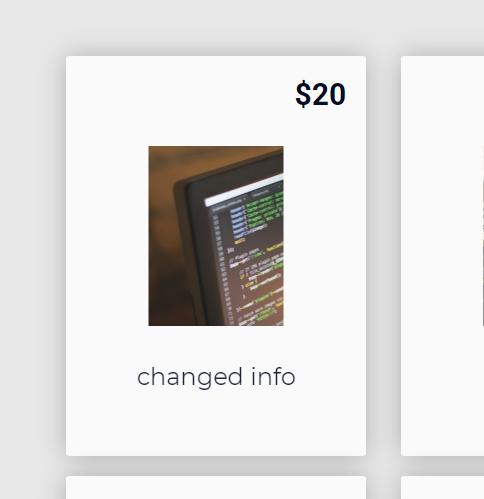


Figure 22 - Edited product

Now let’s see what does delete button do. Once user clicks the delete button, user is shown a confirmation dialog where user has to click OK to confirm. Once it’s confirmed it is deleted from the interface and the database.

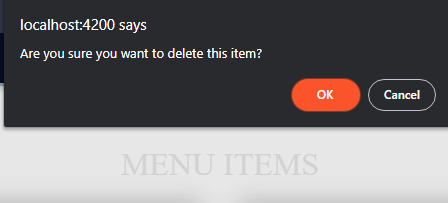


Figure 23 - Delete product alert.

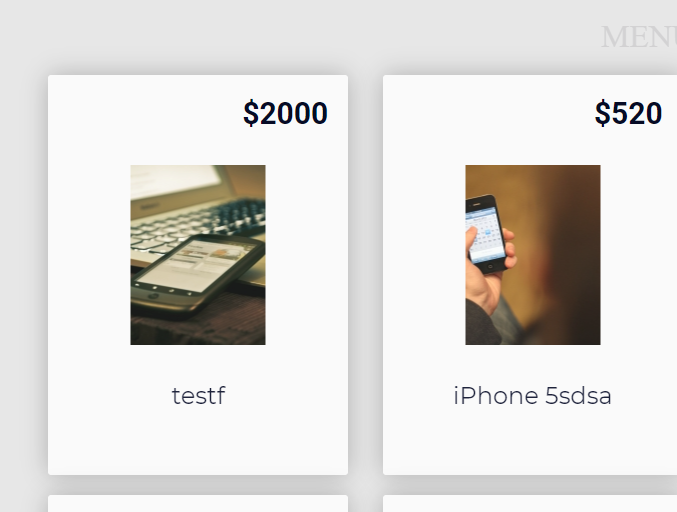


Figure 24 - Deleted product example

Now to Create functionality. User can create a product by entering its details. User can even use any image found on the interned. In this case we just use a simple Google image address found on Google.

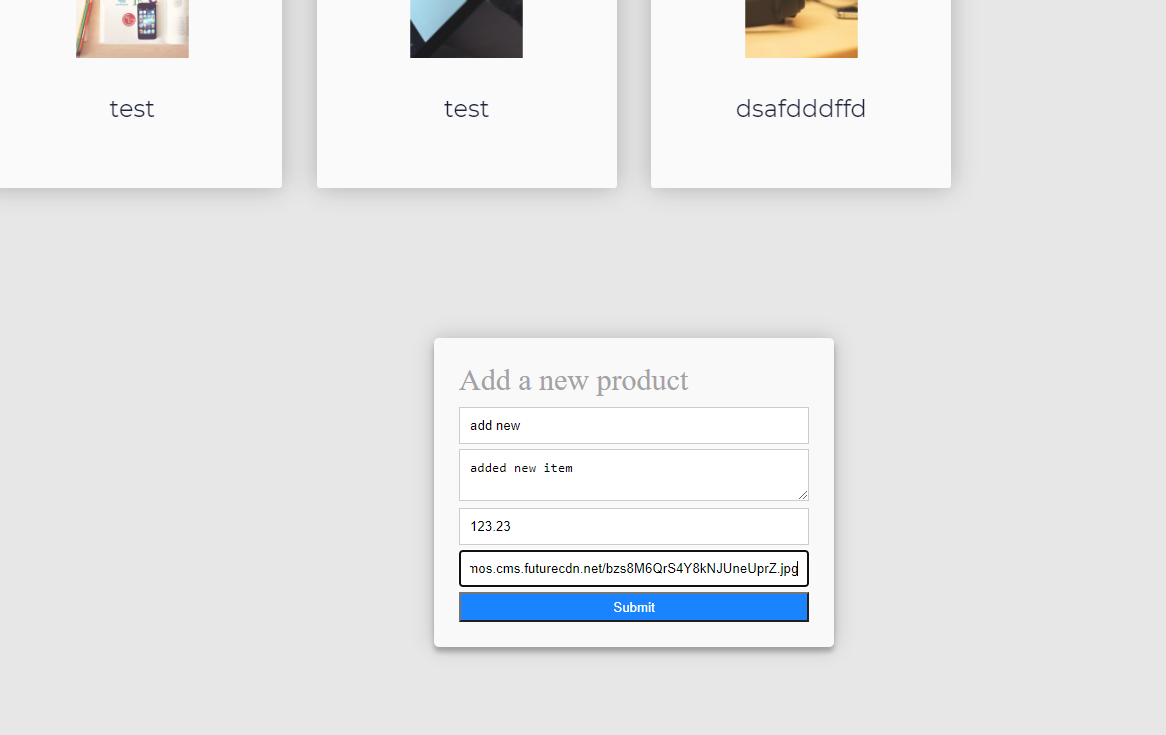


Figure 25 - Add new product form.

Once we click ‘Submit’, we can see our changes straight away in the user interface.

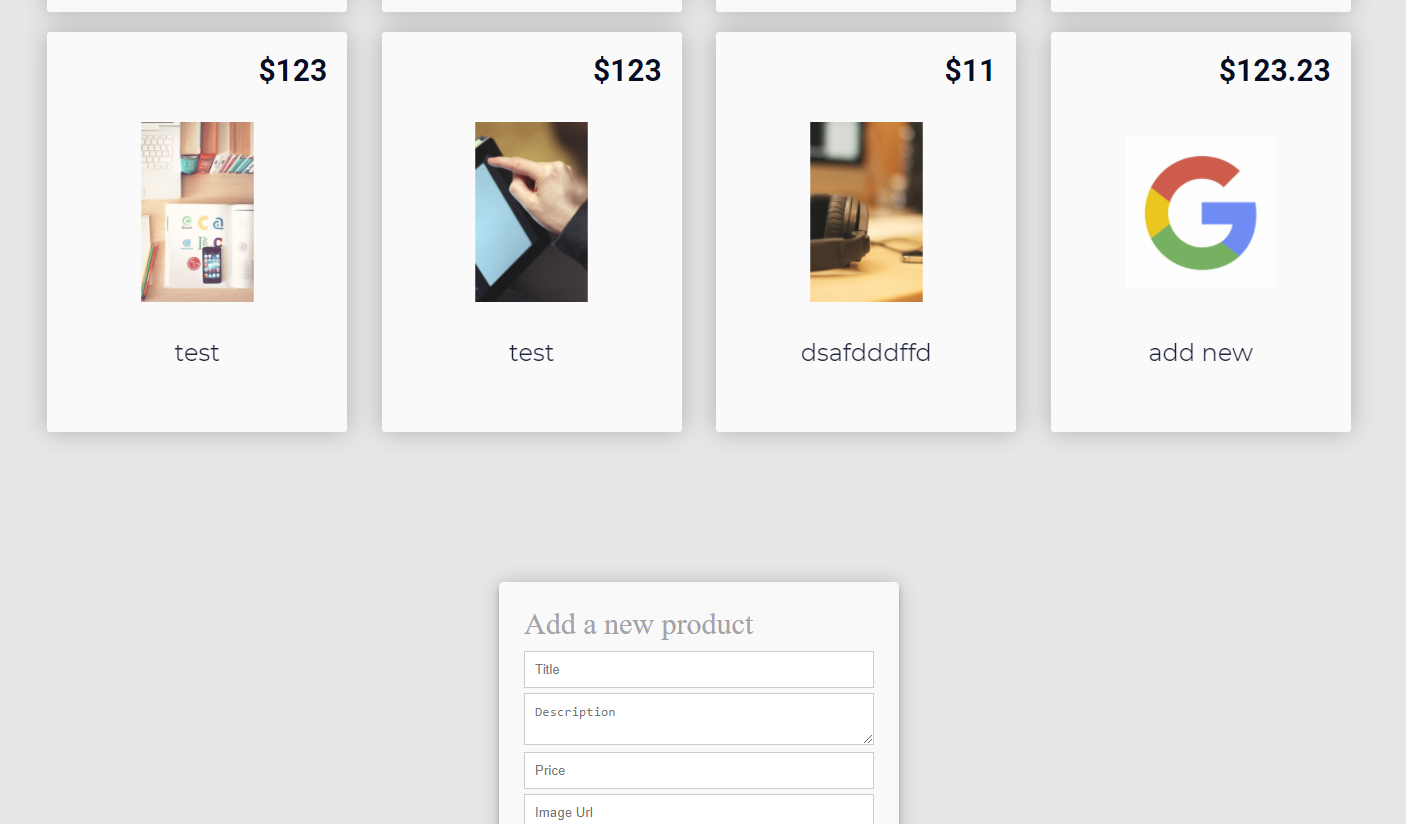


Figure 26 - Added product example.

This was a simple CRUD application to learn the basics of Angular how to create methods and do HTTP requests.

## User guide for CRUD eCommerce application

Wizard application can be mostly used while completing a process where the process takes more than one step, such as registering for a new account, or entering shipping details into the website. Well, this application is exactly that. First user is greeted with a shipping details page where user has to enter all the details to continue.

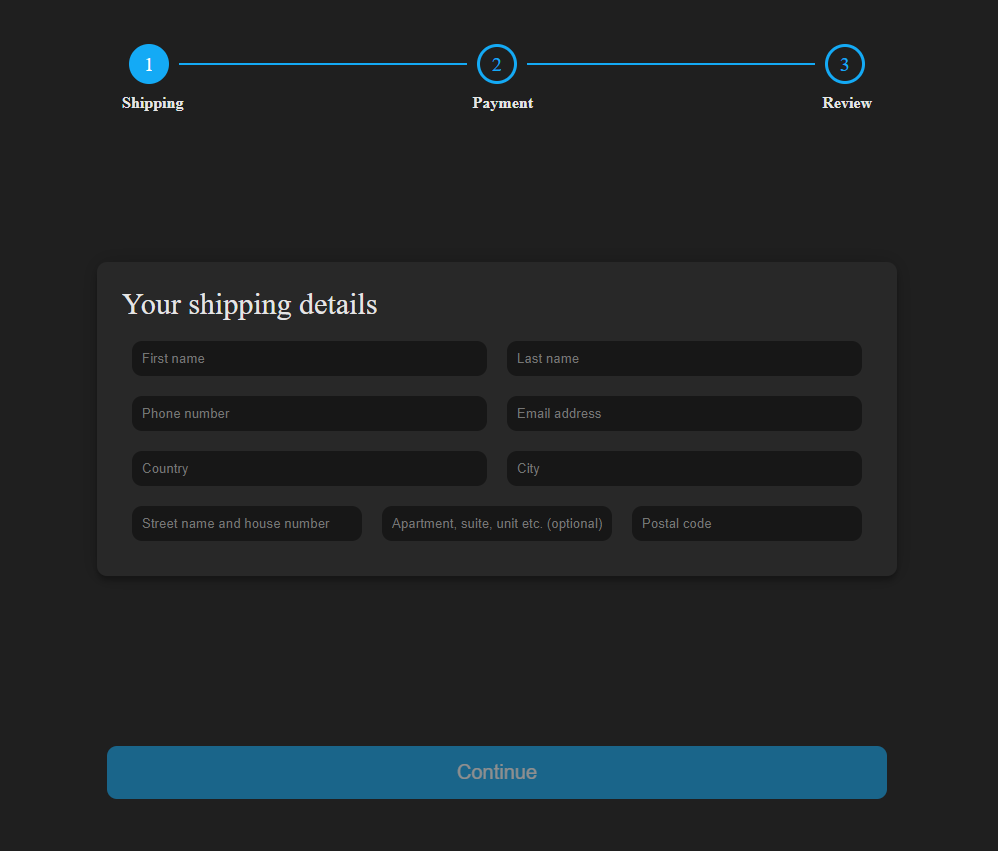


Figure 27- Wizard Shipping Form

We can see that the button is enabled one all the necessary data is filled.

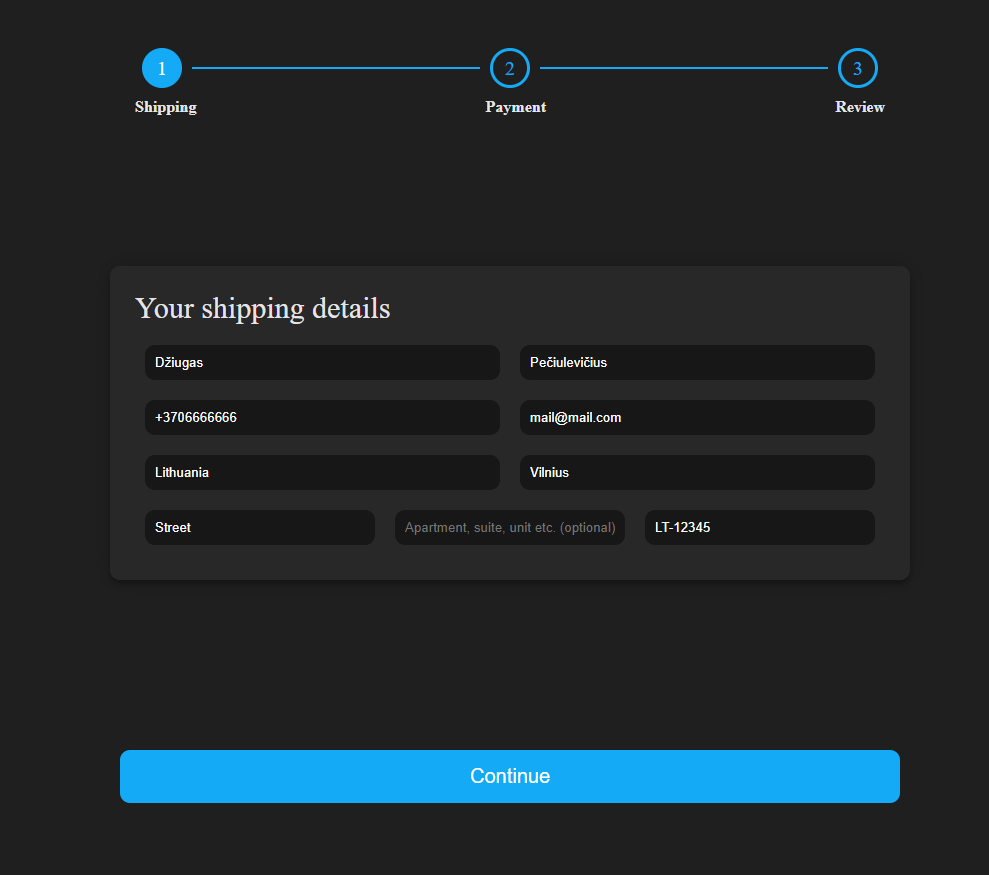


Figure 28 - Wizard shipping form filled.

When all the data is filled and user continues to the next step, user is greeted with another step where this time user has to enter payment information.

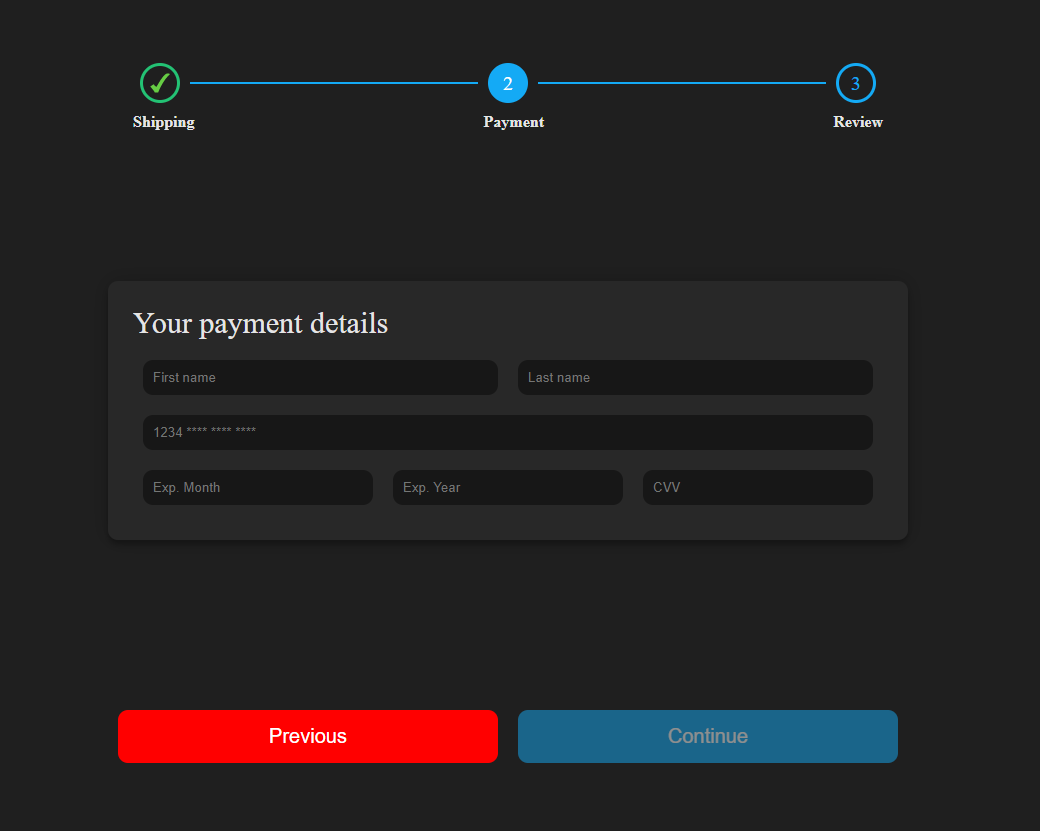


Figure 29 - Wizard payment form

And once again, once all the necessary data is not filled, user cannot continue to the next page.

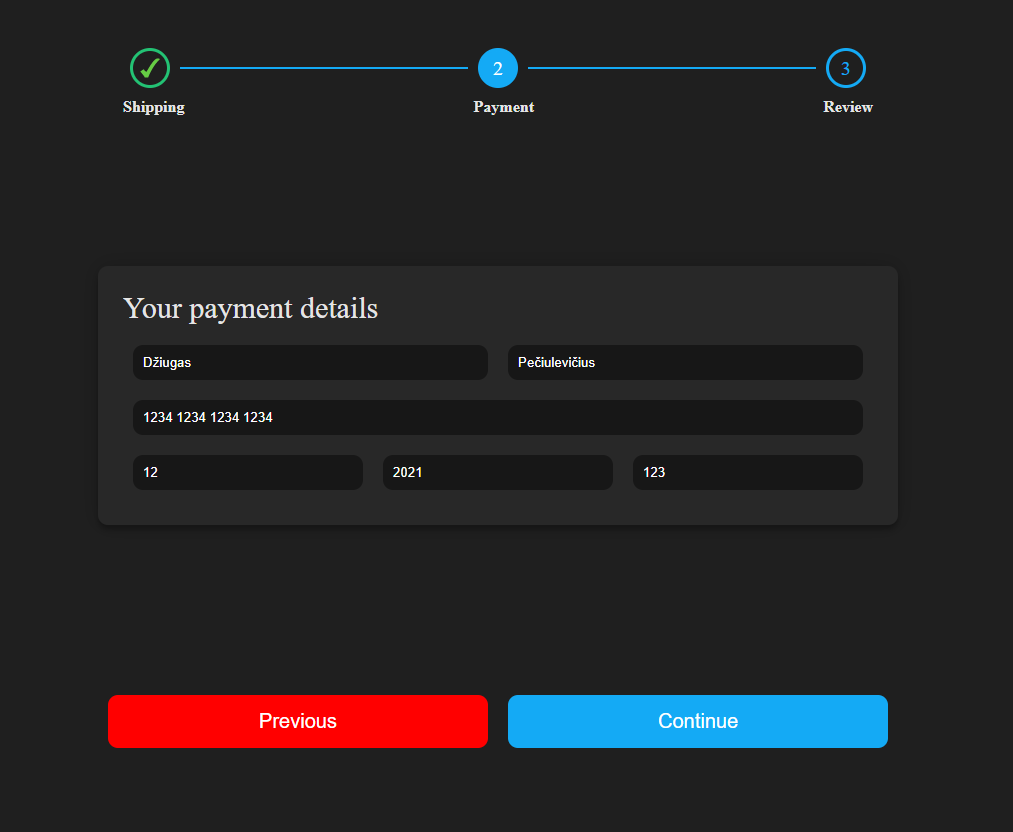


Figure 30 - Wizard payment form filled.

When user reaches the last step of the process, user can review all the data entered, and to continue, user has to confirm that all entered data is correct. If user does not accept, then user cannot send out the form to the backend.

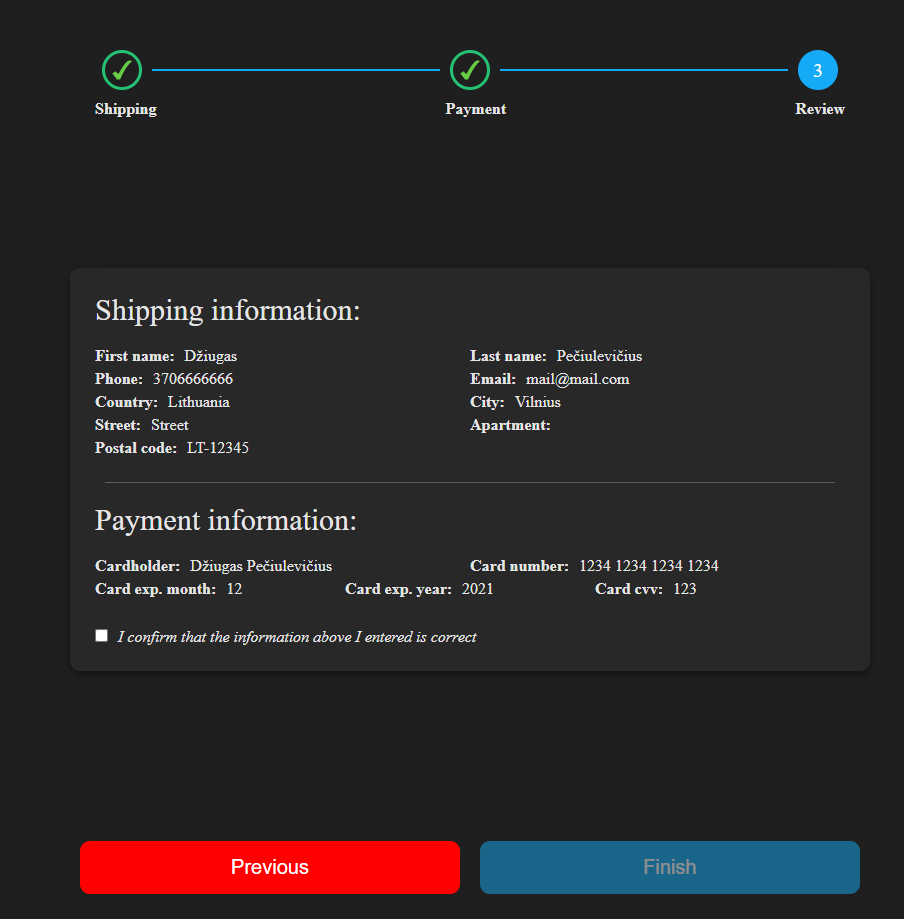


Figure 31 - Wizard review page

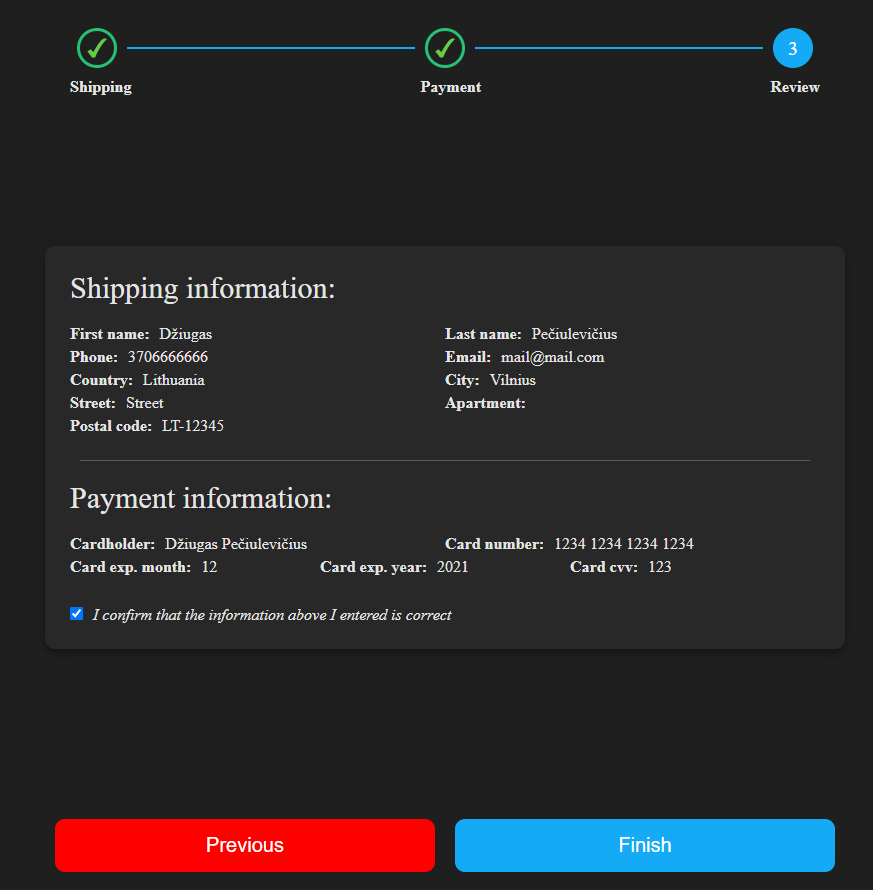


Figure 32 - Wizard review page confirmed.

When the form is sent out, user can see the confirmation.

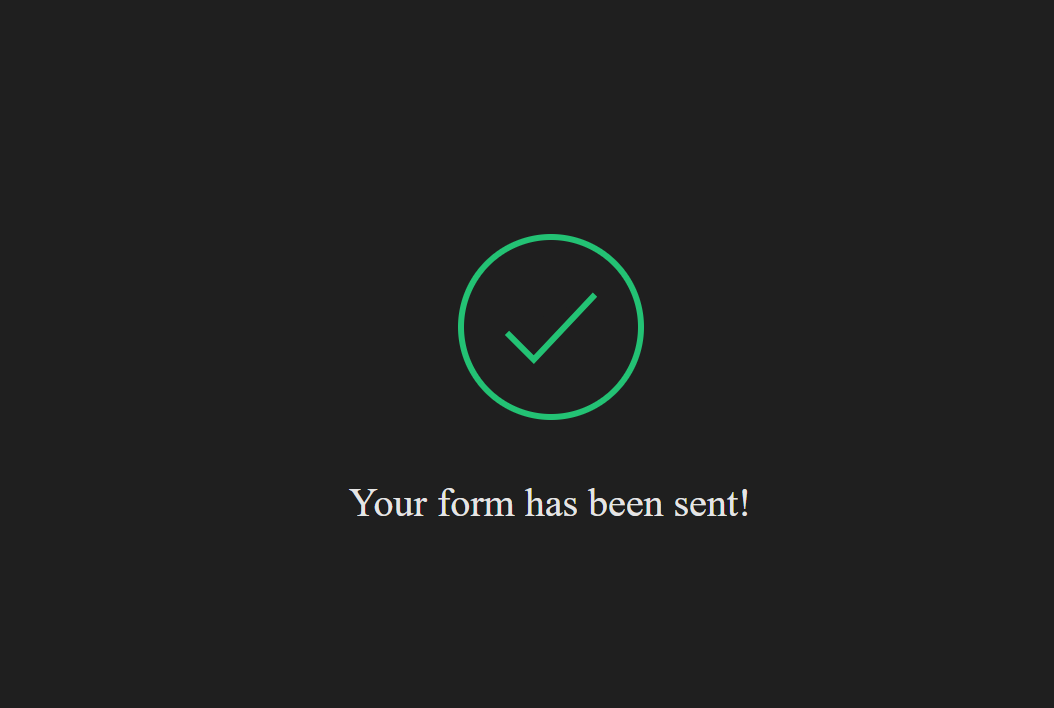


Figure 33 - Wizard success page

# CONCLUSIONS AND FUTURE STEPS

For the conclusions, I could say that the internship at Visma was a success. All assigned tasks were straightforward. Since interns were not allowed to work on a codebase, we had an opportunity to work and create parts for the application they are building and can’t tell us much about it until we have a working contract. Other than that, we learnt how to use new cutting-edge technology used by top companies in the world such as Google and best practices on writing clean and maintainable code.

To sum up the internship, we created a simple application using vanilla JavaScript and testing to see how everyone holds up, and then we began learning JavaScript frameworks. To make it easy, we built the same CRUD application just for a framework to see how it compares to a plain application written without any frameworks. And then later on in the internship they gave us applications to create that would benefit the projects in the company without letting interns touching actual codebase.

# LITERATURE

1. Angular documentation <<https://angular.io/docs/>/>
2. SCSS <[https://sass-lang.com/](https://sass-lang.com//)/>
3. TypeScript <<https://www.typescriptlang.org/>/>
4. Vanilla JS CRUD application <<https://github.com/pdziugas/Visma-E-Commerce>/>
5. Testing practice repository <<https://github.com/pdziugas/jest_node>/>
6. Angular CRUD application <<https://github.com/pdziugas/ecommerce-angular>/>
7. Angular wizard application <<https://github.com/pdziugas/angular-wizard>/>