Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

SZAKDOLGOZAT

Development of a server-driven single-page webshop application

Ghiurutan-Bura Péter
Mérnökinformatikus BSc

2022

Témavezető: Naszlady Márton Bese

Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

1083 Budapest, Práter utca 50/a www.itk.ppke.hu

Szakdolgozat témabejelentő nyilatkozat

Ikt.: I/1290/2022

Név: **Ghiurutan-Bura Péter**	Neptun kód: **AA6UJX**
Képzés: **Mérnökinformatikus BSc (IANI-MI)**	

Témavezető

Név: **Naszlady Márton Bese**

Dolgozat

Dolgozat címe: Development of a server-driven single-page webshop application

A dolgozat témája:
Inertia is a new approach for building classic server-driven web applications. By leveraging existing server-side frameworks, the complexity of SPA-s become considerably smaller, the need for building API-s decreases almost entirely. With this pattern one can benefit from the best functionalities of classic serve-side applications: routing, controllers, data manipulation, authentication, authorization, as well as the best parts of single-page applications: JavaScript rendering and no full page reloads, Inertia's unique feature is implemented in the view layer. Instead of using server-side rendering of HTML, the views are JavaScript page components. Throughout the project the student will create the database and the core functionalities of a webshop, with the given technology stack: Laravel backend, React frontend and Inertia as the complementary between the two. The project is ought to contain layers of authentication and authorization. With the goal of deeper understanding, the student should examine the protocols and the patterns Inertia is using.

A hallgató feladatai:
The following tasks should be performed throughout the project:
-Study of Inertia.js
-Initial project setup: Laravel-Inertia-React
-Representation of the database schema, configuration of the database
-Laravel Eloquent implementation: migrations, models, seeders
-Frontend stucturing: pages, layouts
-Application of server side routing, controllers
-Implementation of a authentication and authorization system
-Realization of core webshop functionalities

Kelt: Budapest, 2022. 09. 15.

Kérem a szakdolgozat témájának jóváhagyását.

Aláírva a Space rendszerben
2022.09.15. 18:09:38
Ghiurutan-Bura Péter
hallgató

A témavezetést vállalom.

Aláírva a Space rendszerben
2022.09.18. 18:04:25
Naszlady Márton Bese
témavezető

A szakdolgozat témáját az Információs Technológiai és Bionikai Kar jóváhagyta.

Aláírva a Space rendszerben
2022.09.28. 23:09:49
Cserey György
dékán

Declaration Of Authenticity

I, undersigned Ghiurutan-Bura Péter, student of the Faculty of Information Technology and Bionics at the Pázmány Péter Catholic University, hereby certify that this thesis was written without any unauthorized help, solely by me and I used only the referenced sources. Every part, which is quoted exactly or in a paraphrased manner, is indicated clearly with a reference. I have not submitted this thesis anywhere else.

Ghiurutan-Bura Péter

Abstract

Single page applications are being more and more frequently used in the world wide web. Whether it is a webshop, a grading system or a trading webpage, the speed and responsiveness of the pages become a necessity. The drawback is that often times single page applications bring an unwanted complexity: state management, routing or API building.

My thesis revolves around the creation of a server-driven single page webshop application, with the help of a monolith library, Inertia.js. Inertia is a new approach for building classic server-driven web applications. Throughout the thesis I emphasise the patterns used in Inertia to bypass the redundant complexity and connect the two worlds: back-end and front-end.

The webshop consists of three architectural parts. The server side functionalities are being solved by the PHP framework, Laravel. Thus the layers of routing, authentication and eloquent data manipulation can be accomplished simultaneously. Most of the web applications require a database behind the data layer. Throughout the thesis work I researched the pros and cons of the relational and non-relational databases, in the end opting for a MySQL database to fulfill the criteria of a webshop data management system.

The user interface is being implemented in React.js which serves as a perfect library to create interactive user interfaces or design simple views for each use case of the webshop. Moreover, an additional utility-first CSS framework, Tailwind is responsible for handling the UI niceties such as responsiveness on every device size.

During my thesis project, the webshop fully obtained the capabilities of a e-commerce application. By accessing the website, users are prompted to freely browse several categories and products, they are served by a customizable shopping cart as well as the possibility to completely select, order and ship a product. Furthermore the thesis required the examination and implementation of a real-time payment system, which choice lead to PayPal.

In conclusion, Inertia.js proved to be a completely adequate solution for solving the complexity issues faced in common single page applications, while keeping the common design patterns. By dodging the requirement of maintaining two separate code repositories, Inertia influenced the project in a productive and enjoyable way to develop full-stack applications.

Acknowledgement

I would like to express my deepest gratitude to my supervisor, Naszlady Márton Bese for his invaluable feedback and expertise throughout the whole project.

Special thanks go to everyone at Riptos, for always inspiring me to improve and providing me with the perfect environment to grow.

I would be remiss in not mentioning my mother, my family and my friends, whose patience and kindness impacted me tremendously.

Lastly, this journey wouldn't have been possible without Adrienn, her presence and never ending support is the sole reason I was able to arrive where I am today.

Contents

Chapter 1

Introduction

1.1. Web protocol

In 1989, Tim Berners-Lee handed in a proposal to build a hypertext system over the internet, later known as the World Wide Web. Described in his whitepaper, a hypertext system consists of four building blocks: [1]

- A HyperText Markup Language known as HTML to respresent the documents
- In order to transfer these documents, there is a need for a transfer protocol: HTTP
- A client to display these documents
- A server to store the documents

Clients and servers establish a request-response message transfer pattern. The route of exchanging messages is initiated by a client when they send a request, whereas the server receives it, processes it and finally returns a response. This is called inter-process communication as well. [2]

This messaging pattern can be implemented in both synchronous and asynchronous fashion. While in the first the connection is being held until the response is received, messages sent asynchronously can be received with a delay. [2]

The HTTP protocol acts as the foundation of any web service in which the data exchange is fulfilled within a client-server architecture. The request-response cycles consist of the browser requesting documents with an HTTP request, and as their task - the server responding to these documents and handling possible errors. [3]

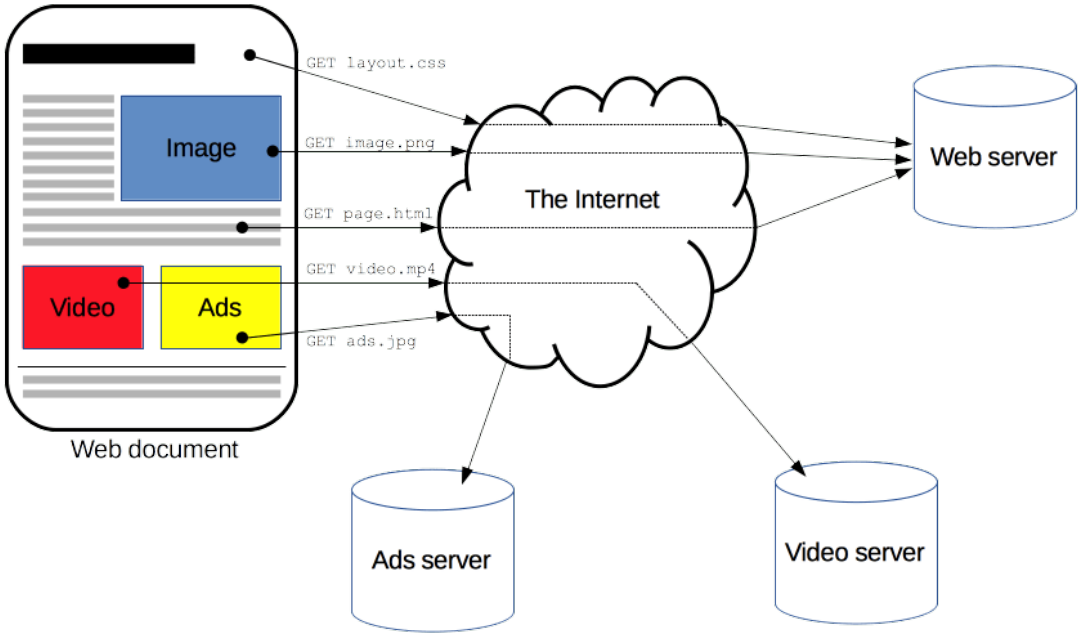


Figure 1.1: Steps of reconstructing a whole document by fetching the resources. Source: [3]

A basic illustration of this, would be accessing the webshop's website deployed throughout the thesis. Assuming the domain is called www.mywebshop.com, whenever an user accesses the website, its browser becomes the client - which sends an HTTP request to the server for the resources. The server reconstructs the complete documents by fetching the different sub-documents. [4]

HTTP request methods, in shorter terms HTTP methods are verbs or nouns that signify the desired action of the client when making a request. The HyperText Transfer Protocol defines a set of request methods, from which the most common used ones are: [5]

- GET: The most used request method on the web, GET requests simply retrieve a representation of the specified resource
- POST: All POST requests are being sent when the goal is to create a new entity from the resource data
- PUT: The PUT method replaces all current representations of the target resource entity with the data received in the request body
- PATCH: PATCH requests are sent to update some already existing entity on the server. The data that needs to be refreshed has to be sent within the request body
- DELETE: Upon receiving a DELETE request, the server removes the specified resource. It is the opposite of GET, while GET retrieves data, DELETE verbs

eradicate the data.

1.2. Frameworks

With the introduction of JavaScript in 1996, the web has reached an unseen growth in interactivity - it became a dynamic place. By teams and developers constantly working on problems, the repetitive behavior of these tasks lead them to create tools, packages that tackled these issues in a well constructed manner. Each programming language, by shaping its own ecosystem made room for the appearance of frameworks and libraries. [6]

In computer programming, a software framework can be interpreted as an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, hence making it adequate for the given project. [7]

In other words, framework takes the role to fill the low-level functionality, while the user can fulfill the high-level functionality parts which make a project unique. [8]

A framework brings structure to the code: by providing ready-made components, easily understandable syntax and common design patterns - the framework acts as a guide throughout the journey of programming. By solving the problem of reusability, the development time is speed up immensely. [9]

While often times, in technical documents as well - the scope of frameworks and libraries are not entirely distinguished, their scope can be well defined by the principle of inversion of control (IoC) [8]

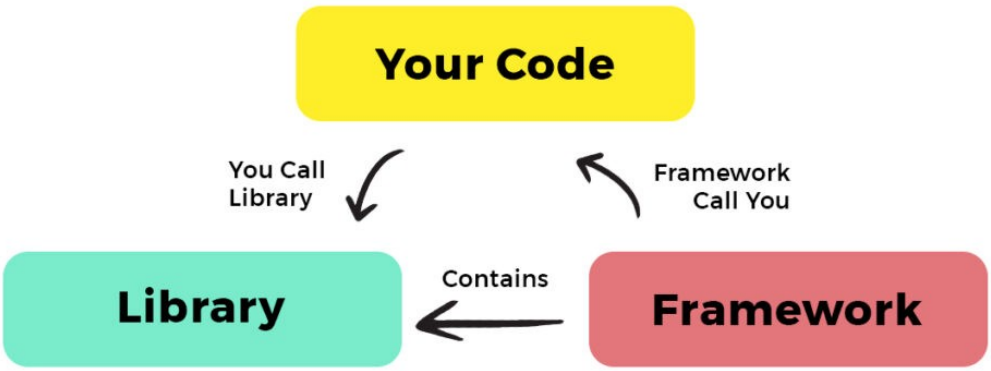


Figure 1.2: Inversion of control between libraries and framemworks. Source: [10]

The main difference between two relies in the code property: upon using a library, for example React, the control of the code is totally in the hands of the developer, who instructs the application when to use the library's functionalities.

On the other hand, the control is reversed in frameworks, the framework signals the developer where code needs to be provided and calls it as needed. [11]

However, as it was proven in my webshop project as well, the utmost goal of frameworks and libraries points to the same direction: by solving common problems and providing patterns, they help in reducing the room for error, provide security from the get go and assist sustainable code.

1.3. Single page applications

Web applications can be divided into two design patterns: multi-page applications (MPA) and single-page applications (SPA), with both of them having pros and cons on their side.

While multi-page applications often time can become very complex, with the introduction of AJAX, the data transfer between server and browser became significantly more simpler. AJAX calls made it possible to refresh only particular parts of the application. [12]

Single-page applications (SPAs) have slowly become the modern way of creating a web application, and the primary reason is that they provide a more native user experience combined with an increase in loading speed. [13]

An SPA (Single-page application) is a web application that in its core loads only a single web document, and then proceeds to update only the body content of that single document. [14]

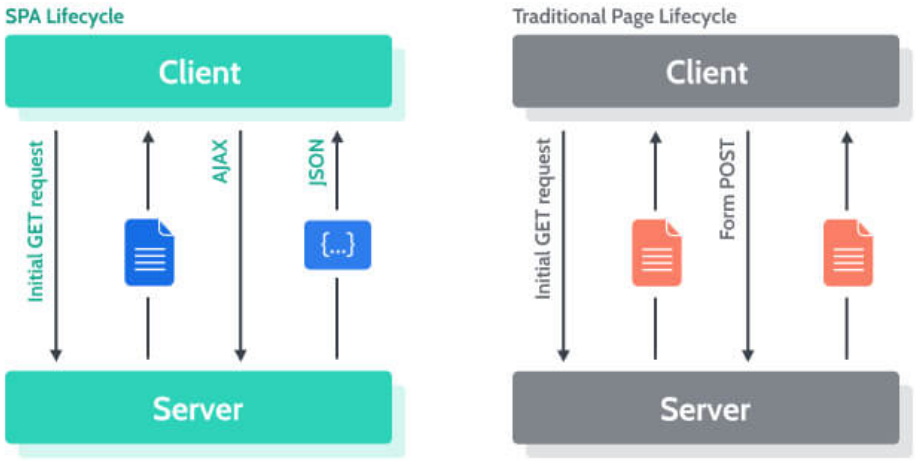


Figure 1.3: Lifecycle differences: while SPA's communicate with AJAX and only swap the changed components dinamically, MPA's require a full page reload on each request [15]

The main advantage of Single-Page Apps is that they achieve the redrawing of any part in the UI without requiring additional HTML from the server. [16]

After the initial page load, the server does not transfer any more HTML to the browser. The server initially sends the browser the app shell which makes the user interface renderable. [17]

Upon navigating, the application sends request for additional data and after receiveing the appropiate resources - the browser updates the user interface, interchanging the components without the need for a full page refresh. [18]

In essence this allows users to visit and browse websites without loading whole new pages from the server, thus resulting in several performance gains and a more dynamic and native-like experience. [14]

Web applications that need to function in browsers with limited or no JavaScript support should opt for the creation of traditional, multi-page application workflows (or be able to provide a way to fall back to classic functioning). The problem could appear from the fact that single page apps require client-side JavaScript in order to function. In case of older browsers, it becomes cumbersome, but luckily the vast majority of today's browsers run SPA's smoothly. For instance if a person is using the lower versions of a Internet Explorer browser than IE 11 then single page applications might not render

properly. [19]

1.4. Project structure

The core idea of the the project is to build a server-side driven single page webshop application, where users can access, browse and purchase several products real-time. The products are enrolled in different categories, and one can purchase these items. The webshop has a database based shopping cart, as well as order and shipment functionalities. The payment integration is done real-time with a PayPal driver. With the help of the server-side utilities, notifications and emails are being delivered after successful purchases.

From an abstract layer the webshop can be structured in three larger parts, which are constantly communicating and transferring data between each other: a Laravel backend, a React frontend - as well as Inertia.js - the bridge between the two. For the sake of data storage, collection and manipulation the project uses a relational database called MySQL.

Chapter 2

Inertia.js

With the rise in popularity of single page applications, more and more questions arised: what should be the next step when development teams want to replace their server-side rendered views with a modern JavaScript-based single-page application front-end?

Architecture wise the most common answer is that the system has the need to build and consume a GraphQL or REST API. Additionally in a React based project the necessity of handling client side state management (Redux, Recoil, Zustand) as well as client side routers and cachers (React Router, React Query) increase the complexity. One of the solutions looks to be **Inertia.js**. [20]

2.1. Core concept

Inertia.js acts as a monolith allowing developers to create SPAs with both client-side and server-side rendering, essentially a fully JavaScript-based single-page app without all the complexity that surrounds it, such as building APIs or managing state.

Inertia.js can not be comprehended as a framework, its role as a library is to combine the greatest parts of classic server-side apps (routing, controllers, and eloquent database access - MVC) with the greatest parts of client-side apps (JavaScript rendering and no full page refreshing). [21]

By basis, it is framework agnostic, currently works with most of the server-side (Laravel, Ruby, Django) and client-side frameworks (Vue, Svelte, React). Throughout the webshop project the technologies combined are Laravel and React.

2.2. Working principle

As previously mentioned the server-side build is almost completely similar to a regular Laravel project. By introducing Inertia in the system, one can still utilize all the functionalities and drivers provided by the framework: the routing is being processed server-side, the M(V)C model is fulfilled by the usage of models and controllers, authentication skeletons are available as well as the possibility to implement authorization, event listeners, notification system and many more.

The key area where Inertia enters, is the view layer. According to their fundamental principles, server-side frameworks use server-side rendering (Laravel blade template), with Inertia the views become JavaScript powered pages making room for all the front-end libraries. [22]

Naturally, just by changing simple blade templates to JavaScript pages, the pages do not act as single-page applications, on a navigation click the browser client would still make a full page reload. Inertia solves this by anchoring as a client-side routing library. [21]

Inertia has a main `<Link>` component, a light wrapper around a normal `<a href >` link. Upon clicking this link, Inertia intercepts the click, and handles it by making an XMLHttpRequest (XHR) request instead. XHR requests make it possible to retrieve data from a URL without having to do a full page reload. This enables a web page to refresh only the part of the page which is needed. [23]

As a result when making the XMLHttpRequest (XHR), the server realises that the request is coming from Inertia, and to solve the full reload issue, instead of returning the page's whole HTML response, it only returns a JSON response containing the page component name and similarly to React, sends the data as props. Behind the scenes, Inertia swaps out the old page component with the received one, and keeps a history of the states. [22]

2.3. Protocol

Behind the scenes, Inertia's protocol works as follows:

When a user visits a webpage for the very first time, the result is the same as if Inertia was not present - the browser performs a standard full-page request, where the client gets back a full HTML response, containing the website assets (CSS, Javascript). [24]

The addition with Inertia is that additionally the browser receives the root `<div>`

including the page component name and its data passed as props.

The base template: [25]

```
<html>
<head>

</head>
<body>

<div id="app" data-component="{{ $component }}"
data-props="{{ json_encode($props) }}"></div>

</body>
</html>
```

This <**div**> acts as the mounting point, the client-side framework in my personal case React is booted here. [21]

With the Inertia application booted, by accessing the `<Link>` anchor the upcoming requests will be XMLHttpRequest (XHR) objects, filled with a so called X-Inertia header. This is how the server differentiates regular requests and Inertia requests, hence the ability to bypass full page refreshes. [25]

Upon detecting the Inertia request, the server only returns the new page component name and props as JSON. Lastly the instant reload effect is achieved by Inertia keeping a browser history: after the XHR request, it updates the old page component and props with the currently received one. [21]

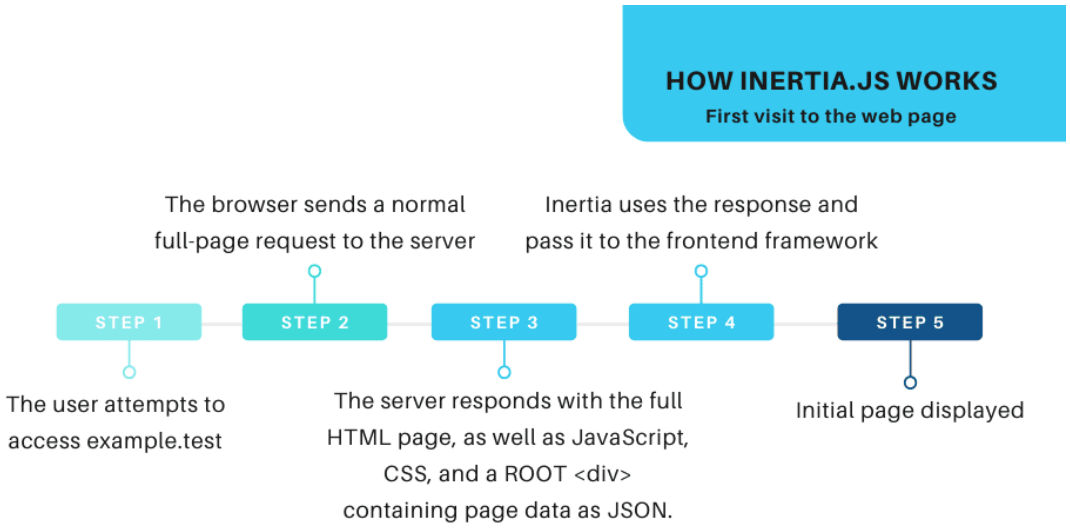


Figure 2.1: Inertia protocol on first page loading: full resources are sent back - acting as a traditional application. Source: [24]

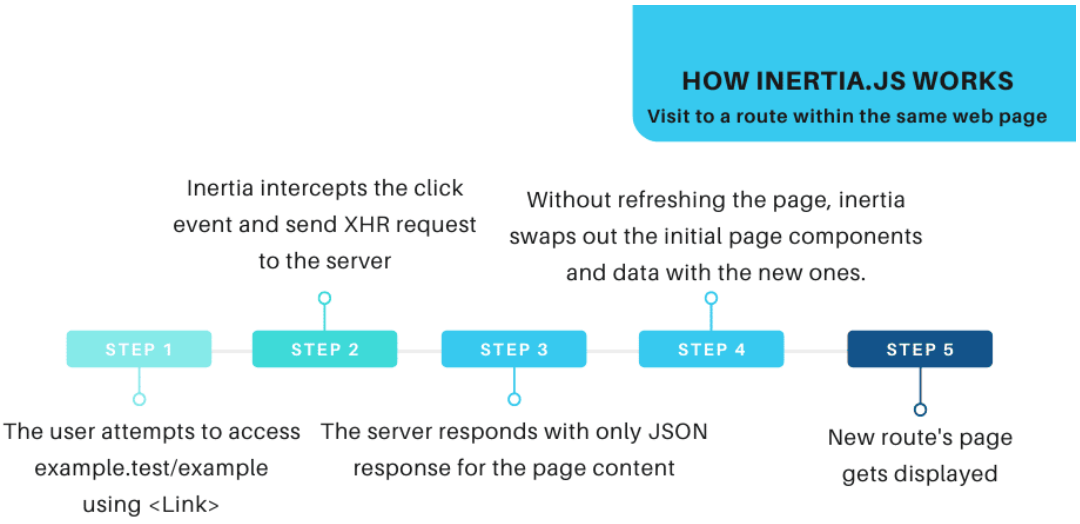


Figure 2.2: Inertia protocol on subsequent page visits: dynamic swapping. Source: [24]

Chapter 3

Database structure

Every single web project in general tends to have a database set up in the background. A database can be interpreted as a system that helps organize data for quick access by storing records in tables with different fields. By this, one can use natural language to search and retrieve information. [26]

In order for the applications to work with data they constantly interact with the database. Nowadays, the two main databases can be differentiated as SQL and No-SQL databases, referring to whether or not they're written solely in Structured Query Language (SQL). [27]

3.1. Database considerations

"Structured Query Language" known as SQL is a query language that has been widely utilized for data querying and management in relational database management systems starting from the 1970s. [28] In the moment SQL is still commonly used for querying relational databases. Looking at its structure, data is stored in tables and rows, while tables can be linked to each other through their fields - forming a relation. This architecture forms a relational structure between the tables, which offers fast data storage and recovery, and it becomes able to handle great amounts of data and logically complex searches. [28]

On the other hand, NoSQL databases use so called documents instead of tables. Due to the fact that they allow a dynamic schema for unstructured data, the importance of pre-planning and pre-organization of the database is significantly lower than in SQL systems. [28]

A common misconception is that NoSQL databases or non-relational databases don't

handle relationship data well. NoSQL databases can store relationship data, but the architecture differs from the common relational system. [29]

Non-relational systems act like file folders, they assemble related information of all types without necessarily categorizing the data. [27]

3.1.1 Scaling

Scaling is an important aspect in an application whenever the problem of data enlargement arises. Databases are able to scale vertically or horizontally, and this can act as a determining factor when opting for a given database. [27]

On one hand, SQL databases can scale vertically, as a result increasing the processing power of existing hardware, by adding more CPU, RAM or SSD capabilities. [30]

NoSQL on the other hand are databases which are horizontally scalable. To achieve larger storage space and load, the solution is to add more servers or nodes to the database. Cost-wise SQL databases are generally more expensive than No-SQL ones. [31]

3.1.2 Data queries

SQL is a great choice when the data needs to be structured and related. The query language allows flexibility, and results in efficient data access. Complex and larger queries can be performed with relational databases, with consideration for data integrity. [28]

NoSQL serves as a solution in cases of big-data: large amounts of information, which can constantly change throughout time. Since data can be categorized into different structures: documents, key-value pairs, wide columns, graphs and searches the query time can be significantly faster, but without the integrity guarantee. [32]

3.2. MySQL implementation

As of now Laravel provides first-party support for five databases: [33]

- MariaDB 10.3+ (Version Policy)
- MySQL 5.7+ (Version Policy)
- PostgreSQL 10.0+ (Version Policy)
- SQLite 3.8.8+
- SQL Server 2017+ (Version Policy),

but there is the possibility to implement other SQL/No-SQL databases with the help of drivers.

Taking everything into consideration, while constructing a e-commerce webshop I considered ensuring the **ACID compliance** (Atomicity, Consistency, Isolation, Durability) the most important factor. Accordingly the system had its data model based upon a relational, **MySQL** database.

For larger abstraction the database schema of the webshop was modelled as follows:

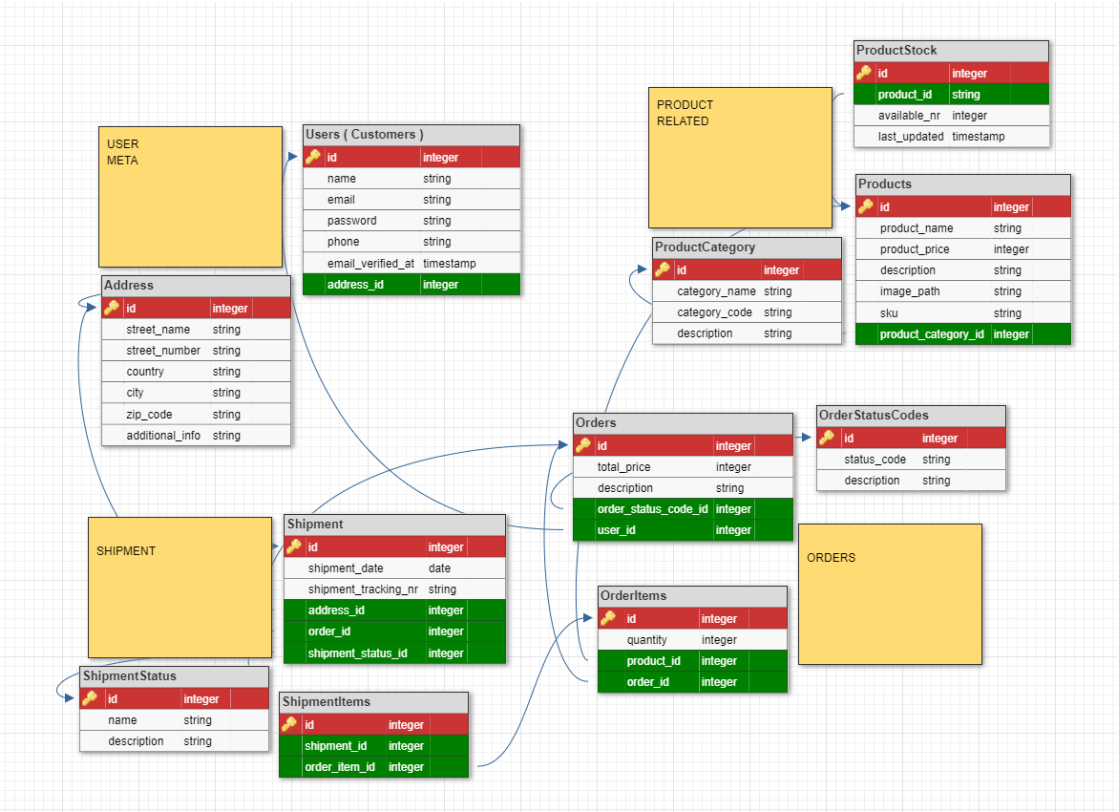


Figure 3.1: Database schema designed in dbdesigner.net

3.2.1 Database logic

By its logic, the database can be divided in four main segments: User, Product, Order and Shipment related tables. Each table contains a primary key (depicted with red), and occasionally a foreign key (depicted with green) which links one model to another. From an atomic point of view, each segment has its own core tables, forming a relationship through some of their fields.

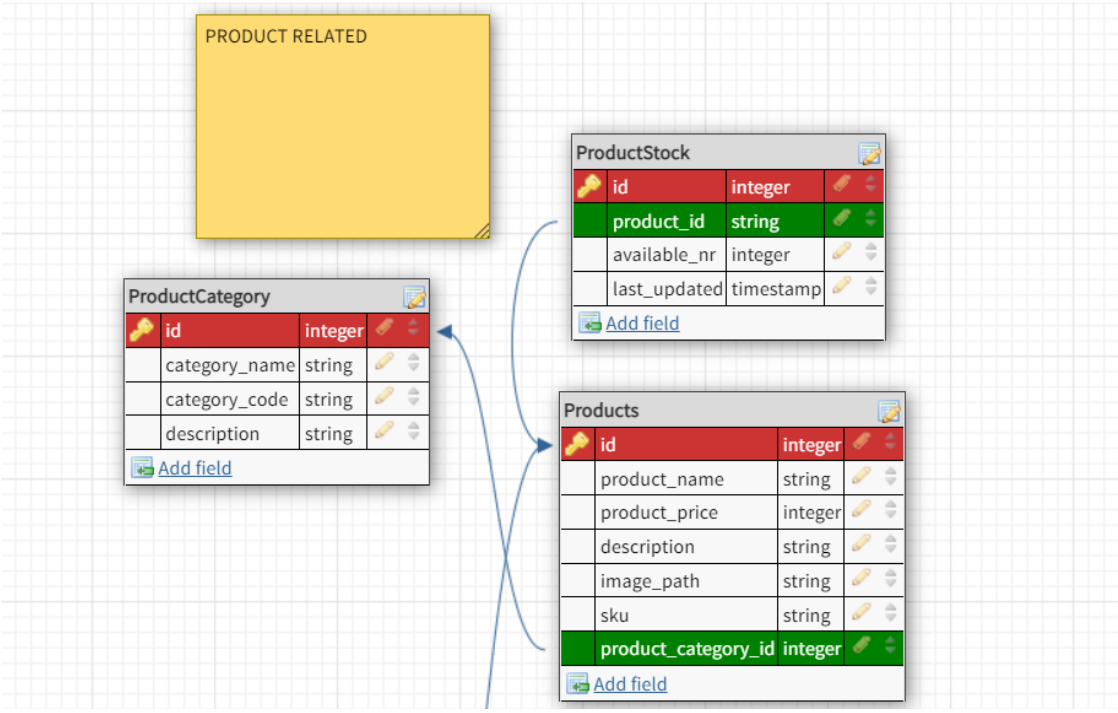


Figure 3.2: Product related tables

The Product table contains the name, price and many other properties of the products found in the database. In order to gather the product's stock availability, the two are connected with a foreign key. Each product belongs to a Category, while the inverse logic results in each Category having many products.

If we proceed to an outer layer, the segments (Product - Orders) form a relation as well in the database.

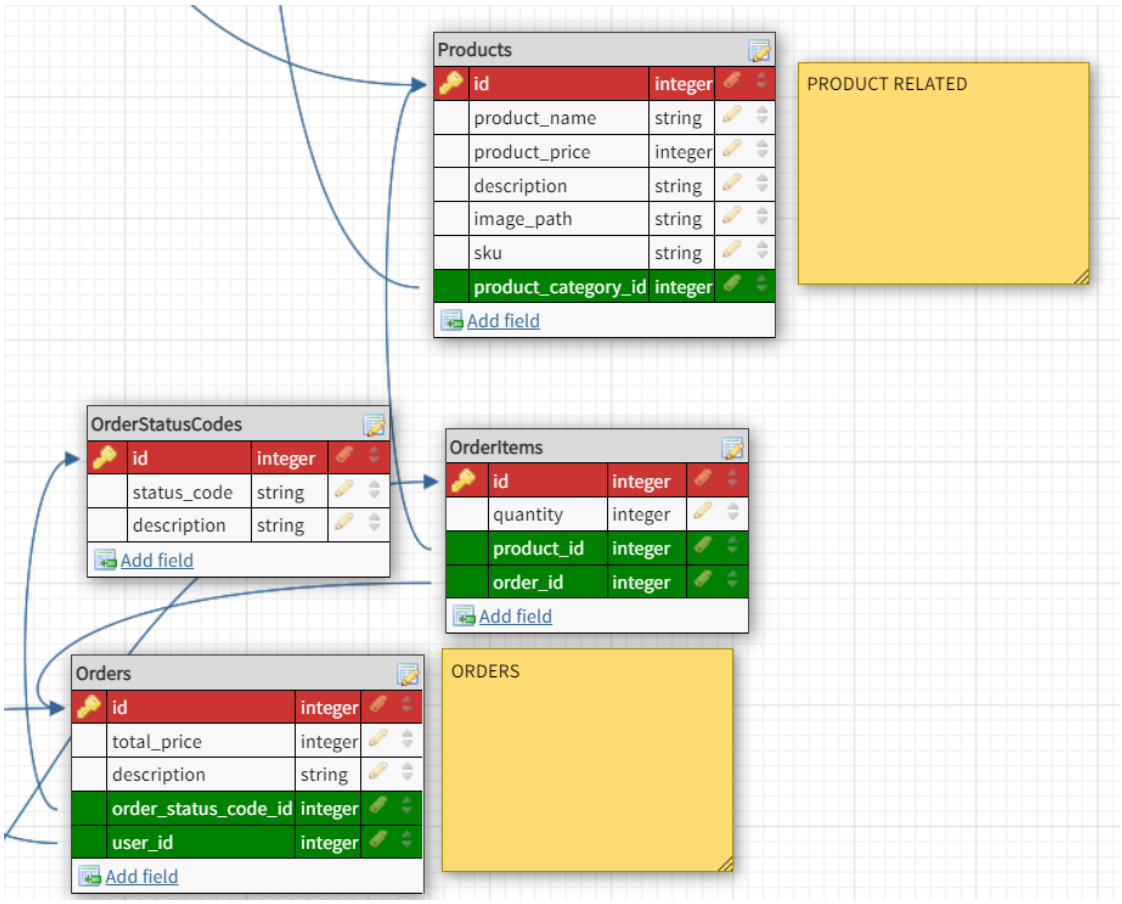


Figure 3.3: Relationship between Order and Product segments

Every Order can be destructured into its OrderItems, whereas it can be said that every OrderItem consists of a Product. These carry into effect the concept of browsing, purchasing and tracking of products and orders.

Chapter 4

Technology stack

The webshop's structure can be interpreted as three building blocks: Laravel as the server-side framework, React.js responsible for the user interface and Inertia.js solving the problem of the two blocks communicating.

4.1. Backend technology

4.1.1 MVC model

Model-View-Controller known as MVC is a pattern in software development generally known for implementing user interfaces, data flow, and controlling logic isolated. In its core it results in a clear separation between the software's business logic and display. [34]

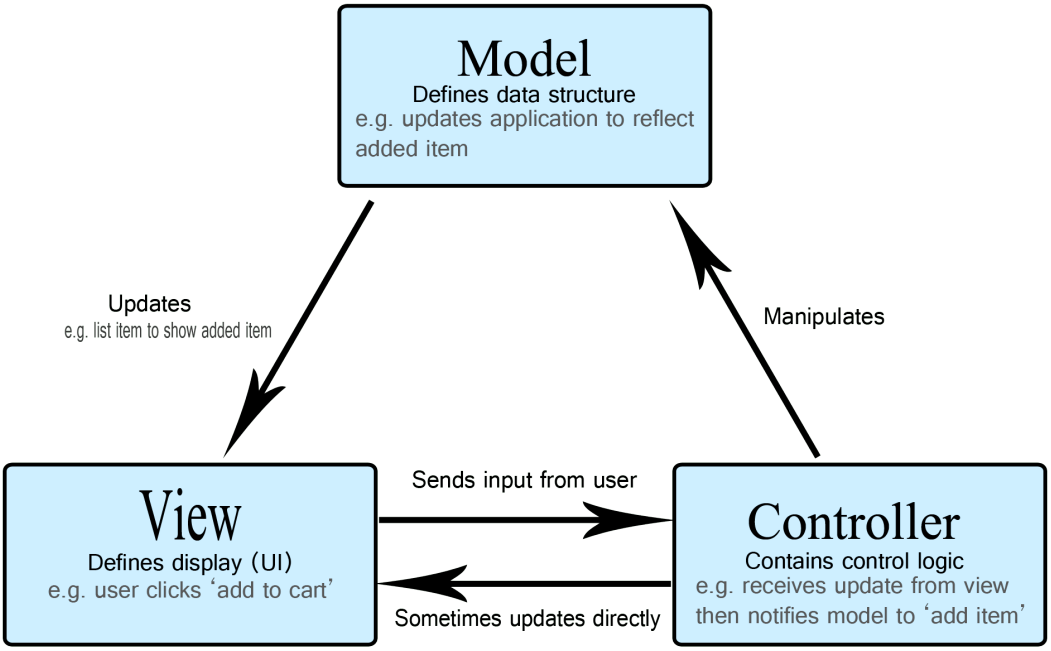


Figure 4.1: Model-View-Controller pattern and their roles. Source: [34]

The three main building blocks of the Model-View-Controller pattern can be described as:

- Model: Responsible for data and business logic
- View: Handles the user interface: layouts, views and display
- Controller: Processes commands between the model and view layer, transfers data.

When developing smaller projects, an unstructured filesystem might not backfire drastically, however - with the growth of a project - having a structure in the application and keeping the responsibilities isolated leads to improvements in maintainability and easier debugging. [35]

4.1.2 Laravel

When choosing a framework, I aimed to opt for a framework with developer-friendly documentation, and seemingly endless functionalities. In my experience, the predictability of a system allows development to scale and improve drastically.

Laravel is an open-source PHP framework, which provides high-quality backend func-

tionalities: routing, database eloquent ORM, controllers, caching, authentication and many more. Due to PHP's scaling-friendly nature, Laravel has out-of-the-box support for distributed cache systems, and the user doesn't have to worry about security issues such as cross-site scripting, or SQL injections which are all covered. [36]

Throughout developing the webshop project, I've got familiar with Laravel's ecosystem, offering great packages which help in the most robust way to improve one's application. [37]

4.1.3 Server-side folder structure

Laravel's folder structure emphasizes the importance of keeping responsible functionalities isolated. Each folder contains purpose specific PHP files, which can interact with each other throughout runtime.

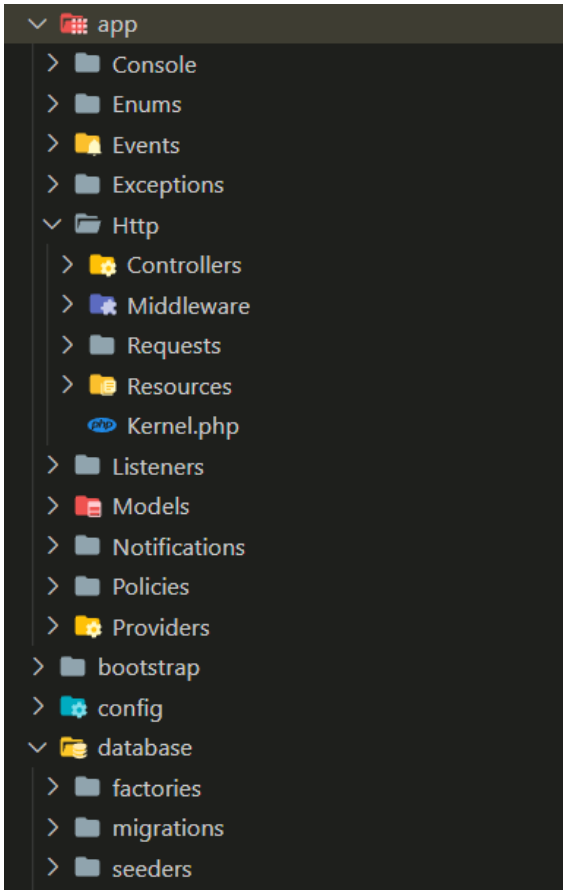


Figure 4.2: Laravel's folder structure taken from VS Code

The majority have the app folder as their root. The components that were most frequently used during the thesis project:

- Database: creating and manipulating the connected MySQL database
- HTTP - Controllers : Controlling and processing the webshop's requests coming from the browser, forwarding data to the view layer
- Model: as described in the next chapter, database tables can be modelled as PHP classes
- Notifications/Event Listeners: sending and processing notifications through database or email
- Policies: provides authorization: for instance an admin's and a normal user's capabilities can be different on the website

4.1.4 Database

The webshop was powered by a MySQL local database. Laravel makes interacting with databases extremely simple. It provides support for utilizing raw SQL, a fluent query builder, and the Eloquent Object-Relational-Mapper. [38]

Migrations

Migrations are the tool for building the database schema. It provides a version control like utility to manipulate the database tables. Within a migration a whole new table can be added with its new fields, or the respective fields can be changed on an already existing table.

My task was to implement the database schema presented in the database considerations chapter: creating the respective tables, fields and setting up the relationships correctly.

In order to create the Products table in the database, a migration class has to be created.

```
public function up()
   {
      Schema::create('products', function (Blueprint $table) {
         $table->id();
         $table->string('product_name');
         $table->decimal('product_price');
         $table->string('sku');
         $table->text('description');
         $table->string('image_path');
         $table->foreignId('product_category_id')
            ->constrained()
            ->onUpdate('cascade')
            ->onDelete('cascade');
         $table->timestamps();
      });
   }
```

The products schema consists of a blueprint, which gives a specification about the fields: their type (common database types - boolean, integer, string, varchar, date),their name and also provides the possibility to allow null values, or set a default value.

In relational databases tables can form a relationship, the foreign key constraints can be set up in the migrations. By telling Laravel that "product categoryid" is a foreign key pointing to another table, the system connects the two tables Products and ProductCategories.

With attention to getting the best out of the Laravel ecosystem, some strict naming conventions are ought to be followed. These naming conventions are used to force referential integrity at the database level. The table name should be the plural of the given model in English (product - products table). On the other hand, if we want the automatic detection to be applied regarding foreign keys as well, the keys have to be defined in such a way that an id sequence follows the model name we want to reference. The foreign key for the ProductCategory, would have the syntax of: product category id.

Following these rules brings a faster and smoother development experience - Laravel automatically detects the components that need to be connected and keeps the data integrity. Laravel provides a helper command line tool named Artisan, with which we can generate the files and fields with the correct naming convention.

After running the migrations, the database connected to the Laravel project undergoes an update - every table and field defined in the project is migrated over.

Seeders

Before any user interaction happens on the website, one might want to preload the database with useful data. Laravel provides the ability to seed the database with data using seeder classes. [39]

A seeder class' run method gives room to use the query builder to manually insert data or to utlitize the Laravel provided factory methods.

For a better user experience, I prefilled the webshop with categories and products, which the customers can browse by their own preference. To seed the products table, I have made a skeleton JSON file, containing the given product's information (name, price, category id)

```
"products": [
      {
         "product_name": "Macbook Air (M1, 2020)",
         "product_price": "999",
         "sku": "1",
         "description": " Apple's M1 SoC (System-on-Chip).",
         "image_path": "images/Products/Laptops/Apple/MacAirM1/",
         "product_category_id": "2"
      },
```

After providing the seeder class with this data, as well as the seeder's run method gets called from the root DatabaseSeeder file, Laravel inserts the provided resource in the correct MySQL table.

4.1.5 Eloquent ORM

While writing raw SQL or using the query builder might be useful in complex situations, to complete the MVC pattern, Laravel includes Eloquent, an object-relational mapper (ORM). Eloquent ORM is the projection of a database table to a corresponding Model class.

Models

A model is a simple PHP class, which is used to interact in all sorts of form with the appropiate table: retrieval, creation, update or deleting are all made possible through simple syntaxes. [40]

```
class Product extends Model
```

Setting up the webshop, after all the database tables and their associated models have been created, eloquent makes these models ready to interact with. The framework provides prewritten methods (retrieve, create, and many more) on the Model class for this reason:

The retrieval of all the models in the database can be interpreted as

```
Product::all()                    SELECT * FROM products table;
```

By using Laravel's built-in helper method, inserting a new record would be equal to:

```
Product::create([data]) <=>

INSERT INTO products (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

4.1.6 Eloquent relationships

In the webshop, a Product and a Category model are connected - one category can have multiple products and a product belongs to a category. Generally, to connect any of the database tables, Laravel supplies relationship helpers, which make managing and working with these relationships really simple. Laravel provides support for most of the relationships found in relational databases. Each of these relations can be set up in the respective model's class.

One to one

A one-to-one relationship is a very straightforward type of database relationship. It is used to unequivocally map two models together: one user has exactly one cart, and the cart belongs to exactly one user. The relationship can be defined by forming the relationship's methods on the models. [41]

```
public function cart()
   {
      return $this->hasOne(Cart::class); }
```

The hasOne method accepts as argument the name of the Model related. By following Laravel naming conventions for tables and primary keys, Laravel can automatically detect the desired model.

In order to access the user given a cart, we have to define the inverse of the relationship:

```
public function user()
   {
      return $this->belongsTo(User::class);
   }
```

After connecting the models, we can make use of eloquent's dynamic properties. Dynamic properties allow the developer to access relationship methods as if they were properties defined on the model: [41]

```
$cart = User::all()->first()->cart()
$user = Cart::all()->first()->user()
```

In the application another example for a one-to-one mapping is the relationship between a Product and an OrderItem. Every OrderItem belongs to exactly one Product and vice versa.

One To Many

A one-to-many relationship can be defined as a relationship where a parent model can have multiple children models, and the inverse logic being that multiple models belong to the same parent model.[42]

The perfect example is the connection between a Product and a Category. By defining the relationship method on the Category model class

```
public function products()
   {
      return $this->hasMany(Product::class);
   }
```

it can be said that one category can be the parent of multiple children Product models. The inverse definition sums up as

```
 public function category()
   {
      return $this->belongsTo(
      ProductCategory::class, 'product_category_id');
```

```
    }
```

Meaning that each and every Product in the webshop belongs to exactly one category. This allows the reach of dynamic properties once again, anywhere in the system the two collections can be retrieved:

```
'products' => $productCategory->products()
'category' => $product->category()
```

Considerably, another instances of a OneToMany relationship are between the models: Cart - CartItems, Order-OrderItems, Product-CartItems, Address-User, Address-Shipment.

Has One Through

The "has-one-through" relationship in essence defines a previously mentioned one-to-one relationship with a given model. However, the difference is that the two models are not connected directly, has-one-through relationship signals that the declaring model is being matched with one instance of another model by proceeding through a third model, a "pivot one". [43]

Although the webshop hasn't reached the actual use case for defining this relationship, in a future implementation it could be beneficial.

```
 shipment_delivery_person
   id - integer
   name - string
   phone - string
   badge_number - string

shipment
   id - integer
   reference_number - string
  shipment_delivery_person_id

order
   id - integer
 shipment_id
 order_number - integer
```

If we extend the application, and introduce the real-time shipping tracking, we could manipulate our database with a Shipment Delivery Person model. By having a one-to-one relationship between a Shipment and an Order, we would be able to track and display an Order's Delivery Person related information after the making of a has-one-through relationship.

Has Many Through

The "has-many-through" , similar to has-one-through relationship provides a convenient way to access a has-many distant relations via an intermediate table. [44]

Looking at an example, the webshop has a Cart system defined for each and every User. A user and the items in the cart are not directly in a relationship, but by introducing the Cart model in the middle, we are able to retrieve the user's cart items through its cart. Abstraction wise it can be understood as:

```
  users
   id - integer
   name - string
   email - string
   phone - string

cart
   id - integer
    user_id

cartitems
   id - integer
   cart_id
   product_id
```

The foreign keys form an upward chain here. Defining the relationship methods on the models sums up as:

```
 public function cartItems()
   {
      return $this->hasManyThrough(CartItem::class, Cart::class);
   }
```

By inputting the two parameters in the function parantheses, we are instructing Laravel to look for a CartItem model through the intermediate Cart model. To succeed in this, the tables must contain the foreign keys in the order of the retrieval. (user -> cart -> cartitems)

Now simply put, in the shopping cart area we are able to access all the items with just one line.

```
'cartItems' => $user->cartItems
```

Has Many Deep

While the has-many-through relationship tends to be adequate in most cases, for an even faster and more convenient retrieval, I made use of the Laravel ecosystem, installing a package called eloquent-has-many-deep. [45]

By basis, has-many-deep provides the same functionality as the has-many-through relationship, but the levels and intermediate tables one can reach in depth are unlimited, hence resulting in a more accessible relationship.

Previously we defined how to reach the User's CartItems through the pivot Cart model. While the retrieval of CartItems' is already a great addition, if we wanted to display the actual Products' information belonging to the CartItem, we would have to dig one level more. Without the definition of a new relationship whatsoever, this could be interpreted as:

```
foreach($user->cartItems as $cartItem){
$product[] = cartItem->product;
}
```

We can make use of the previously defined has-many-through CartItems relationship, and after instructing Laravel to look one layer deeper for the products, we can generate a simple retrieval method.

```
use \Staudenmeir\EloquentHasManyDeep\HasRelationships;

public function cartProducts()
{
    return $this->hasManyDeepFromRelations($this->cartItems(),
    (new CartItem())->product());
}
```

The two liner in best case syntax changed to a simple column-like call.

```
'cartProducts' => $user->cartProducts
```

4.1.7 Resources

As a consequence of using Inertia.js, the need for an API disappeared. However, as described in the Inertia chapter the JavaScript powered view layer of the application receives its resources in a JSON format. Laravel provides a transformation layer called resources that sits between the Eloquent Models and the actual JSON responses returned to the front-end.

By using Eloquent resources, we can fully and expressively customize our data. There are other helper facades as well to return a JSON, but intuitively, resources create an isolated and robust structure that is only responsible for this transformation. We are in control of entirely selecting the fields, relationships and data we send over to the front-end. [46]

```
public function toArray($request)
   {
      return [
         'id' => $this->id,
         'firstname' => explode(" ", $this->name)[0],
         'lastname' => explode(" ", $this->name)[1],
         'phone' => $this->phone,
         'email' => $this->email
      ];
   }
```

While sharing the data with our user interfaces take the form of:

```
'user' => Auth::user() ? new UserResource(Auth::user()) : null,
```

4.1.8 Routing

While in a general React.js application the client-side routing can become cumbersome, with Inertia every route of the application can be defined server-side. This means that Laravel's common routing pattern is adequate here, with slight changes regarding Inertia. [47]

Available Router Methods

Presented in the first chapter, the client and server communicates with HTTP verbs. Consequently, Laravel's router allows the developer to register routes that respond to the HTTP verbs:

Laravel routes in general require an HTTP verb, accept a URI and a closure, resulting in a very simple and expressive method of defining routes and behavior, neglecting the complicated routing and neverending configuration files:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Basic routes

The webshop might contain a very basic route, whom response we could define without any hassle of using controllers or views. [48]

Route::get('/hello', function () return 'Hello World';);

View Routes

There are certain cases, when there is no need for a controller. If the route is only supposed to return a view, we can use the Inertia provided ::inertia() or ::render() to signal that Inertia has to intercept the call to return the view, while avoiding a full page reload.

An example from the webshop would be the homepage, where our only task is to render the dashboard view, and we require only one resource from the routing class.

It can be argued, whether this amount of data would require a controller or not. All three solutions bring the same results (although behind the scenes Laravel::view() makes a full page reload, which is an unwelcome behavior in our case).

We are commanding Laravel that, upon requesting the "/" URI to provide the client with the view residing in the Home/Homepage path, combined with the resource response retrieved. Implemented in the web.php routes file this could look like:

```
Route::view('/', 'Home/Homepage',
'newestProducts' => Product::inRandomOrder()->limit(4)->get());

Route::inertia('/', 'Home/Homepage',
[ 'newestProducts' => Product::inRandomOrder()->limit(4)->get()]);

Route::get('/', function () {
  return Inertia::render('Home/Homepage', [
     'newestProducts' => Product::inRandomOrder()->limit(4)->get(),
  ]);
})->name('home');
```

The inertia method accepts a URI as its first argument and the name of the view we want to return as its second. If needed, additional data can be forwarded to the view layer in an array syntax.

Named Routes

In Laravel named routes serve the purpose of calling redirects and page visits in a simpler fashion. They allow the generation of a naming convention for specific routes. It can be setup by chaining the name() method with the chosen name to the end of the route definition. [49]

```
Route::get('/', function () {
 return Inertia::render('Home/Homepage'})->name('home');
```

While Laravel and most of the server-side frameworks allow one to generate URLs from named routes out of the box, unfortunately these route helpers are not available in an SPA client-side. [47]

The solution here is to use a package called Ziggy, that outputs a JavaScript global route() helper function that works like Laravel's URL facade, making it easy to use the defined named routes in the JavaScript frontend. [50]

With this we can utilize routes in React as follows:

```
<ResponsiveNavLink
 href={route("home")}
 active={route().current("home")}>
```

Grouped routes

Route groups are a solution to sharing route attributes - such as middleware, controllers or prefix - across a large number of defined routes without needing to define the desired attributes on each route individually. [51]

Middleware: The common middleware sharing is the most useful group in the webshop application. As detailed later in the paper, there are two layouts in the webshop for whoever's browsing the page: a guest and an authenticated layout. Both have differences in design, and hidden parts.

By assigning the 'auth' middleware to all the routes listed, no unauthenticated user can access any of those routes.

```
Route::middleware(['auth', 'verified'])->group(function () {

  Route::resource('cartitems', CartItemController::class);
  Route::resource('carts', CartController::class);
  .
  .
  .
  });
```

Without using the group middleware, the redundant syntax could be as shown, which in cases of hundred routes becomes cumbersome.

```
Route::resource('cartitems',
CartItemController::class)->middleware(['auth']);
Route::resource('carts', CartController::class)->middleware(['auth']);
```

Route model binding

In many routes, for example the content page of a given product, there is the need to query the database to retrieve the product that corresponds to the chosen ID.

Laravel route model binding provides a solution to automatically inject the model instances into the route definitions and it is also possible in the controller methods. This way, instead of transferring over the product's ID, we can inject the entire product model that was chosen.

```
use App\Models\Product;

Route::get('/products/{product}', function (Product $product) {
   return $product->product_name;
});
```

As seen above, the product variable is type-hinted as the AppEloquent model and the variable name matches the product URI segment.

Handling this route or a given controller method, Laravel will automatically inject the model instance that corresponds to the matching ID from the request URI. Laravel provides automatic error handling as well, in case no model instance is found, a 404 HTTP response is generated and thrown back. [52]

4.1.9 Controllers

While Laravel makes it entirely possible to define every part of the request handling in the routes callback, for a better structure and transparency, controllers are the perfect place to handle these behaviours. A controller class is ought to represent the responsibility its solving: all incoming requests related to an Order can be grouped into the OrderController. [53]

Basic controllers

Laravel controllers extend the base controller predefined in the framework. By rule controller methods can take any function name and signature.

```
 public function show(Order $order)
{
    return Inertia::render('Orders/Show', [
      'order' => $order->only('id','name')
   ]);
}
```

In our routes file, we can assign this controller method to a route as:

```
Route::get('/order/{order}', [OrderController::class, 'show']);
```

Resource

In some cases defining our controllers and methods are the only option, however with the appereance of a common patern Laravel defined a new controller "type".

Vast majority of the requests rely around an Eloquent model, and generally the same sets of actions are performed against these models: get all, show one instance, create a new model, update the given model, or delete the instance. Each of our eloquent model in the system can be abstracted as a "resource".

Since this pattern of manipulating data became so common, Laravel resource routing assign the typical Create-Read-Update-Delete (CRUD) routes to a controller automatically. Laravel provides an artisan command to generate our controllers in a resourceful way out of the box. After generating them, the controllers will contain a method for each of the available resource operations.

Our only task is to define a resource route in our web.php routes file:

```
Route::resource('orders', OrderController::class);
```

The actions created automatically and paired with the controller's method are:

GET	**/orders**	index	orders.index
GET	**/orders/create**	create	orders.create
POST	**/orders**	store	orders.store
GET	**/orders/{order}**	show	orders.show
GET	**/orders/{order}/edit**	edit	orders.edit
PUT/PATCH	**/orders/{order}**	update	orders.update
DELETE	**/orders/{order}**	destroy	orders.destroy

The first column of the table represent the HTTP verbs received by the server. The second column matches the exact request URI-s, with the possible model bindings (order). The third columns maps these URI's to the given controller method (/orders -> index method in the controller). The last column is providing a named route, for easier access.

Responses

The controller method's have the responsibility in the MVC model to return data to the frontend. In a classic Laravel environment where the views are blade templates a

controller show method's response could be designed like:

```
public function show(Order $order)
{
    return view('orders.show', [
        'order' => $order
    ]);
}
```

In an Inertia based environment, the controllers change a bit, in a way that we need to provide both the name of the JavaScript page component, as well as all the props that the view layer requires.

```
public function show(Order $order)
{
    return Inertia::render(
        'Orders/Show/Show',
        [
            'order' =>
            [
                'id' => $order->id,
                'reference_number' => $order->reference_number,
                'total_price' => $order->total_price,

            ]
        ]
        );
}
```

A really important aspect in Inertia rendered pages is that to ensure that the pages load quickly, only the minimum amount of data needed should be transferred to the view layer.

While in Laravel's classic syntax, it might be acceptable in some situations to pass the whole order model instance, in JavaScript based pages it can affect the speed significantly. On the other hand, every resource returned from the controllers to the view layer becomes visible client-side, hence the avoidance of sensitive data is important. [54]

4.1.10 Validation

One possible outcome in Laravel's validation, is that the server receives XHR requests from a JavaScript powered frontend. The way Laravel solves this problem is that instead of generating a redirect response, it generates a JSON response that contains all the validation errors, and sends it back with a 422 HTTP status code.

This work perfectly in systems that catch 422 responses and then manually update the error bag of the form. [55]

Inertia on the hand never receives 422 responses, and solves the whole problem of validation by operating in a classic full page form submission. The difference Inertia makes is that on the server-side the validation errors aren't returned as a JSON response with 422 status code, but it is redirected to the original form page, meanwhile flashing the session with the validation errors.

```
$user->address->fill($request->validate([
        'country' => 'required',
        'street_name' => 'required',
        'street_number' => 'required|numeric',
        'suite' => 'required',
        'city' => 'required',
        'state' => 'required',
        'postal_code' => 'required',
    ]))->save();
```

There's an automatic process going in the background: whenever there are validation errors in the flashed session these get shared with Inertia, hence their availability as props on client-side. As detailed later in the React frontend section, these validation errors can be retrieved within the form's helper error object.

```
const { data, setData, post, processing, errors, reset } = useForm({
    country: "",
    street_name: "",
    street_number: "",
 });
```

4.1.11 Shared data

There are certain situations when passing the same data to each page manually becomes inconvenient. To handle this drawback, Inertia introduces the concept of shared data. After predefining the data we want to share across numerous pages, this becomes available client-side. [56]

A perfect example is the fact that the webshop's top navbar requires the authenticated user info, if there is one. The server-side adapters provide a way in the HandleInertiaRequests class to pass props beforehand. The shared data defined here and the regular page props will be merged on the client-side.

```
class HandleInertiaRequests extends Middleware
{
'auth' => ['user' => Auth::user()
? new UserResource(Auth::user()) : null],
}
```

The concept of page loading speed applies here as well, since these shared props will be transferred with every request, only the most essential data should be defined here. Other use cases from the application are the sharing of the product categories, sharing the notification list as well as the contents of the shopping cart.

4.1.12 Notification system

Laravel provides support for outputting notifications across a variety of delivery channels. From this stack, two perfectly fit the business case of a webshop: prompting the user with a notification in the navbar, as well as sending a order summary email after a successful payment and purchase.

I opted for the implementation of notification system - via the database channel - which can be retrieved and displayed in the application, together with an email based feedback system.

Database notifications

Notifications are represented as PHP class, the choice of the delivery channel can be chosen inside here. [57]

The database notification channel stores the notification information in a database table, which can be generated by an artisan command. The table will contain fields that

have information about the model of the notifier, the id of the notifier as well as JSON structured data about the notification message. Lastly the notification model contains a boolean field whether the notification was read or not. The delivery channel can be set as shown below:

```
public function via($notifiable)
   {
      return ['database'];
   }
```

The toArray method inside the notification class serves as the place to define the data that the notification receives. After finalizing the order, a user gets notified about the details of the order combined with a route to access the order.

```
public function toArray($notifiable)
   {
      return [
         'route' => 'orders.show',
         'details' => 'Order has been placed successfully.
         Your items will arrive soon.',
         'id' => $this->order->id,
         'reference_number' => $this->order->reference_number,
      ];
   }
```

Since they are stored in a database, the given user's notifications can be retrieved with some predefined helpers.

```
//To fetch all notifications for one user:
$user->notifications;

//To fetch all the unread notifications:
$user->unreadNotifications;

To mark a notification as read
$notification->markAsRead();

To delete a notification
```

```
$notification->delete();
```

Mail notifications

Mail notifications provide a way to send notifications through email. To define that a notification should be forwarded via the email channel the config in the PHP class should be prompted. [58]

```
public function via($notifiable)
   {
      return ['mail'];
   }
```

Laravel's MailMessage helper class contains a few simple methods to ease the making of email sendings. While I only made use of the subject changer, customizing the sender, recipient and mailer are all possible. The webshop sends out an email containing the order details and a link to track the order. Laravel autodetects the email it has to send to on the User model. [59]

```
public function toMail($notifiable)
   {
      $url = route('orders.show', [$this->order]);
      return (new MailMessage)
         ->subject('Order complete')
         ->markdown('mail.order.order_paid',
         ['order' => $this->order, 'url' => $url]);
   }
```

As seen above, Laravel also supports mail markdowns - which in essence are pre-built HTML templates for mail notifications. These tend to act as a starter point, where the developer has full control on customizing it. The markdown view receives the data passed in the notification class.

Sending notifications

After defining the notifications, they can be forwarded with the help of a notifiable trait or a notification helper. There are no restrictions on the model of the notified entity - most of the times this is the User model - however it can be chosen freely by adding the Notifiable trait Laravel provides on the model's class.

```
//Notifiable trait                    //Notifiable facade
$user->notify                         Notification::send($user,
(new OrderCreated($order));           new OrderCreated($order));
```

4.1.13 Events

Laravel's events serve as a simple observer pattern implementation, making room to subscribe and listen for various events that occur within an application. [60]

Events can play the role of decoupling various parts of the webshop. A single event can have multiple listeners attached and reacting to it. One of the implementations I've chosen is to listen to OrderProcessed events and notify the user with an email containing the details. [61]

Event and listener detection

Events and listeners can be defined in the EventServiceProvider file in Laravel's application. A listen array accepts the event class on the left side, and the listener associated to it on the right. [62]

```
protected $listen = [
   Registered::class => [
      SendEmailVerificationNotification::class,
   ],
   PaymentProcessed::class => [
      SendPaymentReceivedNotification::class,
   ],
];
```

Defining events

As seen above events are described by PHP classes. They have properties that detail informations about the event and its environment. In the webshop this revolves around the Order model's order instance that has been processed. [63]

```
public function __construct(Order $order)
{
   $this->order = $order;
}
```

Defining listeners

A listener can be defined as a class as well, however here the task is to define the handle method. The handle method accepts the event instance the listener is registered to, and within its handle method any action can be performed. If the event of finalizing an order and receiving a payment occured, the listener should send an OrderPaid notification to the user who belongs to the order. [64]

```
    public function handle(PaymentProcessed $event)
{
    //
    $event->order->user->notify(new OrderPaid($event->order));
}
```

Dispatching events

Dispatching an event can happen manually, by using the event's static dispatch method. The method accepts as argument the paratemer defined in the event's constructor. [65]

```
OrderShipped::dispatch($order);
```

On the other hand, it needs to be mentioned that some events happening in the application get dispatched automatically by Laravel. For instance, models dispatch several events, making it possible to hook into certain moments in a model's lifecycle: retrieved, creating, created, updating, updated, saving, saved, deleting, deleted, trashed, forceDeleted. [66]

4.1.14 Authentication

By basis, authentication is a really important feature in every web application. Laravel allows entirely manual implementation for it, however since this can be complex and a potential security threat, Laravel offers features that help tremendously in dissolving the problems around authentication. [67]

Laravel's ecosystem provides starter kits that can act as a starting point in a web app's authentication. The webshop utilizes Laravel Breeze [68] which is a minimalistic, basic implementation of all the Laravel's offered authentication features: login, registration, password reset, email verification, and password confirmation.

In its essence, Laravel's authentication consists of two important helpers: guards and providers. Guards keep track of how users are authenticated in every request. Breeze

offers a session based solution, maintains and manipulates state using Laravel based session storage and cookies. [69]

After installing the package, Laravel Breeze creates authentication controllers, routes, and the skeleton of the views as well. Throughout the thesis project, I redefined and changed the shell that the authentication provided. The login and register part changed in its controller and in its views. The views received a fully new look, which serves the purpose of registering and signing in on a webshop, as seen in Appendix B. In hand in hand with this are the controllers, where with the extension of the application, the registration data required additional validation and data processing. (Validating address, country, creation of a new cart)

Middleware

As it will be described in the frontend chapter, the webshop has a guest and a authenticated layout. Laravel provides a convienient middleware to handle and secure the routes.

The Auth middleware is automatically registered in the application's HTTP kernel, which makes it available for use. The routes we want to be reachable only for authenticated users can be described with the group method:

```
Route::middleware(['auth', 'verified'])->group(function () {
//Signed in routes here
}
//Guest routes here
```

Consequently, the guest routes have to be defined outside the Auth middleware.

4.2. Frontend toolset

With the presence of Inertia.js in the application, space has been made for modern frontend libraries. The webshop application's user interface layer was made entirely with React.js, arm in arm with the design framework Tailwind CSS.

4.2.1 Tailwind CSS

Tailwind CSS is a utility-first CSS framework, which can be written directly in the markup of the elements. [70]

Tailwind CSS behind the scenes operates by scanning all the HTML files, React written JavaScript componenents, and more generally any other templates - it searches

for the corresponding classnames from which the styles are generated and put into a static CSS file. Throughout the webshop's designing it proved to be really flexible, with zero-runtime.

In its core, Tailwind , contrary to other CSS frameworks does not define a series of styling classes for the elements of the DOM such as a button or a link. Tailwind builds on the creation of utility classes, where the developer can freely mix and match any possible combination of these. [71]

Customization

Tailwind is the perfect framework for building custom-made user interfaces, hence the creators designed it since the start with making customization an ease. [72]

By default, Tailwind will search for an optional tailwind.config.js file at the root of the application. This file is the place to define any desired customization. More or less, Tailwind provides the possibility to create an own theme, where the screen breakpoints, the color palette, spacing, font families and many other aspect can be customized.

Throughout the webshop project, I only applied for the addition of a main font, Poppins. This can be requested and imported from Google's font API: [73]

```
@import url('https://fonts.googleapis.com/css2?
family=Poppins:wght@300&display=swap');
```

Without any configuration Tailwind resembles sans families as the default. By changing the sans variable to our imported font, after rebuilding our project every text takes its new form.

```
theme: {
      extend: {
         fontFamily: {
            sans: ["Poppins", ...defaultTheme.fontFamily.sans],
         },
      },
   },
```

For more complex projects, the applicaton's main theme colors can also be defined here. Since I had an invididual task on the design, I opted for Tailwind's default indigo colors, prompted from 50 to 900 in color intensity.

Utility classes

As described in Tailwind's introduction chapter, the utility-first concept refers to the concept of building the design around a specific style element (text-color, text-background), and not a specific DOM component (button) Tailwind defines these classes as utility classes, which enable the control of the majority of CSS elements: text color, background, border, layouts (flex, grid), font style, weight. A great addition was the :hover effect, where a developer can take fully control on the given element's hovering effect.

By looking at real world applications, maintaining a utility-first CSS project has way more advantages than maintaining a CSS codebase. This conclusion arises from the fact that maintaining HTML is much more simpler than the support for CSS. [74]

Some of the most common utilities applied in the webshop:

```
//Change the text color of a block, results in color: #6A5ACD;
<a className="text-indigo-500" >

//Change the background of a block, results in background-color:#6A5ACD;
<a className="bg-indigo-500" >

 //Change the font style
<a className="font-bold" >

  //Fully customize the border
<a className="border border-indigo-300
border-r border-l border-t border-b" >

 //Tailwind provides simple hover handler: normally the background
 takes the color of -50, on hover intensify to 500
 <a className="bg-indigo-50 hover:bg-indigo-500" >
```

Responsiveness

One major feature Tailwind provides is full support for responsiveness. Every utility class in Tailwind can be applied conditionally at different breakpoints, which allows responsive design to take place in the inline markup of the elements, without ever leaving the file. [75]

Tailwind provides five breakpoints by basis, delimited by the common device sizes and resolutions:

Breakpoint prefix	Minimum width	CSS
`sm`	640px	`@media (min-width: 640px) { ... }`
`md`	768px	`@media (min-width: 768px) { ... }`
`lg`	1024px	`@media (min-width: 1024px) { ... }`
`xl`	1280px	`@media (min-width: 1280px) { ... }`
`2xl`	1536px	`@media (min-width: 1536px) { ... }`

Figure 4.3: Tailwind CSS breakpoints with respect to minimum screen widths. Source: [75]

By the consideration that a customer might want to utilize the webshop from a mobile device, a tablet or screen sizes that are not common, with this Tailwind feature the most visited parts of the webshop (products, cart, orders) profited from responsiveness.

For a higher understanding and actual examples of best practices, I traversed many user interface kits created in Tailwind, with respect to responsiveness. [76] [77] [78] [79]

For instance, a specific product's page, specifically its media content benefited from the responsive design, allowing a user friendly view on every device.

```
<div className="mx-auto mt-6 max-w-2xl sm:px-6
lg:grid lg:max-w-7xl lg:grid-cols-3 lg:gap-x-8 lg:px-8">
```

4.2.2 React.js

The front-end part of the application was developed with React.js which states to be a declarative, efficient, and flexible JavaScript library for building reusable UI components. [80]

Facebook designed React with the idea in mind to make the creation of interactive user interfaces a simplicity and reliably fast. Each state of the application can be designed, meanwhile on data change React takes care of efficiently updating and always rerendering just the bare minimum of the components. [81]

React introduced a so called JSX syntax as well. [82]

```
const element = <h1>This is a JSX syntax</h1>;
```

In its core, JSX is a syntax extending pure JavaScript - it is used within the components to tell React how to draw the UI. The main design concept in the library is that React only separates concerns with segments called components, instead of putting markup and logic in whole different files from each other. [82]

4.2.3 Virtual DOM

Behind the scenes React uses a virtual Document Object Model (a Javascript object) which plays the pivotal role in the great performance since a virtual DOM is way faster than a regular DOM. A virtual DOM object can be comprehensed as a lightweight copy of the real DOM. [80]

React applied the following strategy: for each real DOM object, there is a corresponding virtual DOM object. Their main difference is that the virtual DOM lacks the actual ability to change the UI elements. Nonetheless, manipulating the virtual DOM is extremely fast compared to actually changing the real one. [83]

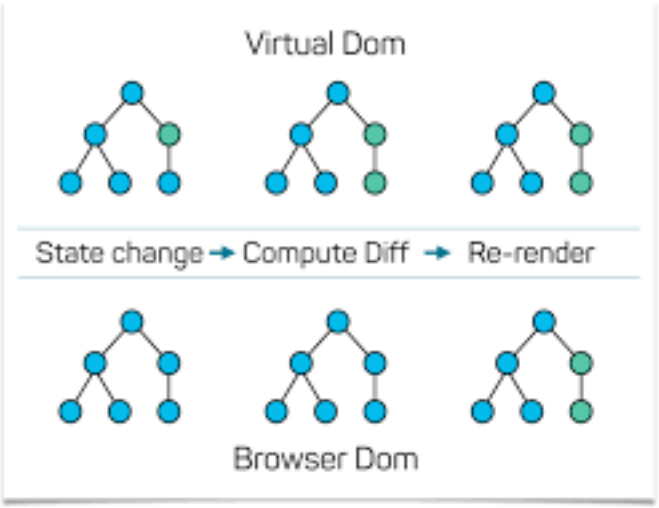


Figure 4.4: The virtual and real DOM gets compared, and React only updates the minimum changed objects [84]

The achievement of reliable speed is due to the fact that when React renders a JSX component, React takes a snapshot of the virtual state before rerendering, after which the virtual DOM objects get updated. Previously stated, in case of the virtual DOM this update gets done extremely fast. After updating the lightweight copy, React processes the diffing, the comparison of the snapshot and the new updated DOM. By having in its knowledge set the actual changed objects, React only has to update the diffed objects. [85] [84]

4.2.4 Component based structure

The whole foundation of React is to promote the build of encapsulated components that manage their own state, then combine and mix these components to make more complex interfaces. [86]

The webshop application's folder and component structure followed the same design pattern:

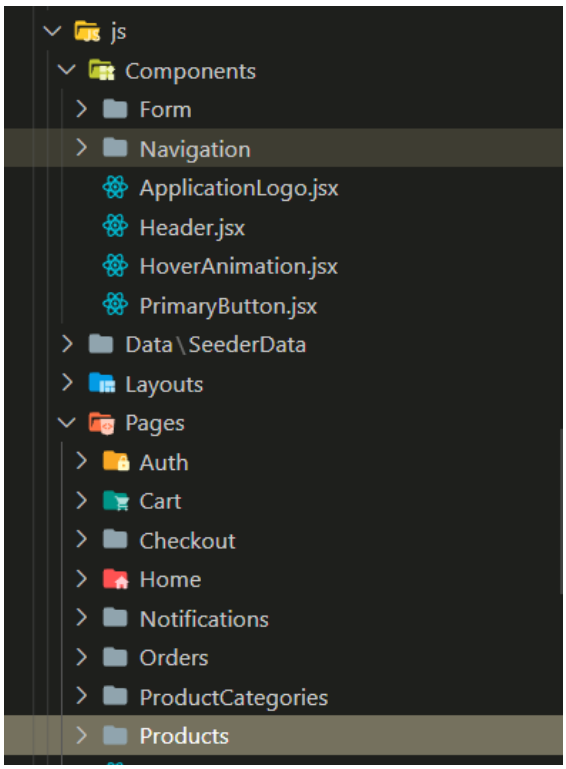


Figure 4.5: React component based folder structure, where every folder has its own responsibility [75]

Upon studying different design patterns, I've divided the webshop in the following parts:

- Components: React components which do not belong to a specific area, they can be and are reused several times in the UI: e.g. the Logo of the application, form input elements, a button
- Data: the data folder contains the products and categories list predefined in a JSON.
- Layouts: There are two main layouts as detailed later on, one for the guest customers and one for the signed in ones. All the page components extend one of these layouts.

- Pages: The pages folders contain in their subfolders all the features and pages the webshop offers. Moreover, these subfolders carry all the components needed for the realization of the actual responsibility. In case of the Products page, this sums up as:

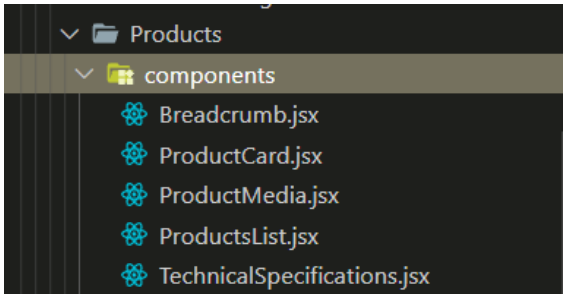


Figure 4.6: Products page related components: a list component, a card component and the other components needed for the whole page

4.2.5 Layouts

Layouts are a great way to build the skeleton of some pages. The webshop achieves the same, by defining the common layout for the view layer: a main navigation bar on the top, a second navigation bar below it, a footer at the bottom of everything, and in-between the content unique for each page.

The layout also allowed the introducing of three openable drawers: one for the shopping cart, one for the categories menu and the last one showing the current notifications. Essentially, React's useContext hook provides access to the drawers opening and closing functions to all children pages of the layout. [87]

```
<NotificationDrawerContext.Provider
 value={{ notisOpen, setNotisOpen }}
 >
```

Even when using Inertia one could simply make use of the layouts as they are generally reused in React, wrapping the given page component between the layout component. However this does force the layout instance to be destroyed and rebuilt on each visit. This results in not having a persistent layout state when going back and forth between pages. This can be easily solved by using Inertia's layout features. [88]

Moreover a default layout can be set for every rendered JavaScript page in the application. In app.js where the SPA's page loading gets resolved, the desired layout can be imported and prompted as the default to extend. [89]

By constraining it with an additional condition, every page which is not placed under the Auth folder (meaning it's not part of the authentication flow) receives the AuthenticatedTemplate as its default layout.

```
resolve: (name) => {
      const page = resolvePageComponent(
         `./Pages/${name}.jsx`,
         import.meta.glob("./Pages/**/*.jsx")
      );

      page.then((module) => {
         if (!name.startsWith("Auth/")) {
            module.default.layout =
               module.default.layout || AuthenticatedTemplate;
         }
      });

      return page;
   },
```

4.2.6 Forms

Inertia provides a form hook called useForm which manifests the input change and error handling in an almost automatic way. [90]

For instance in our login page, we can set to detect and handle the data of our three input fields in an object form. UseForm provides setters (setData) and getters (data) for our inputs, moreover we can display our server-side validation errors by decomposing the errors object.

```
const { data, setData, post, processing, errors, reset } = useForm({
      email: "",
      password: "",
      remember: "",
   });
```

Additionally, Inertia provides all the methods mapped to the HTTP verbs - which requests are forwarded to the Laravel server-side. [55]

```
submit(method, url, options)
get(url, options)
post(url, options)
put(url, options)
patch(url, options)
destroy(url, options)
```

Chapter 5

Webshop fundamentals

The combination of Laravel, React and Inertia.js all provisioned to create the use cases a webshop might offer in a full-stack environment. While a webshop's database logic can take a variety of forms differing from webshop to webshop, for the most simple case - as in my application - the goal was to be able to provide the customer with the opportunity to purchase a selected product within a customer friendly flow.

Altogether, the following can be said about the webshop's core functionalities:

- Upon reaching the website, a user is prompted with the homepage of the webshop: it contains advertising and marketing elements revolving around the products and capabilities the site offers
- A user can browse multiple products in various categories. By basis, the user is advised (and prompted) to sign up and login for an easier all-around access (address, user information is set this way), but a guest can browse the products freely as well.
- Each category has a page, where every product is listed within the category. This list can be searched, sorted and filtered through backend solutions.(Ascending or descending order of product price, date, rating).
- Each product has its own presentational page, where more detailed descriptions and technical specifications become available
- A shopping cart is integrated, where items' quantity can be updated or entirely removed
- The current logic of the webshop states that only signed in users can fulfill the criteria to finalize an order
- Upon selecting the products an order flow welcomes the customer, where they can further add or remove products, insert their billing and shipping address and proceed to the actual payment.

- After finalizing the payment, the user receives a database based notification in the navbar as well as an email notification - both of them have the role to detail the recently processed order

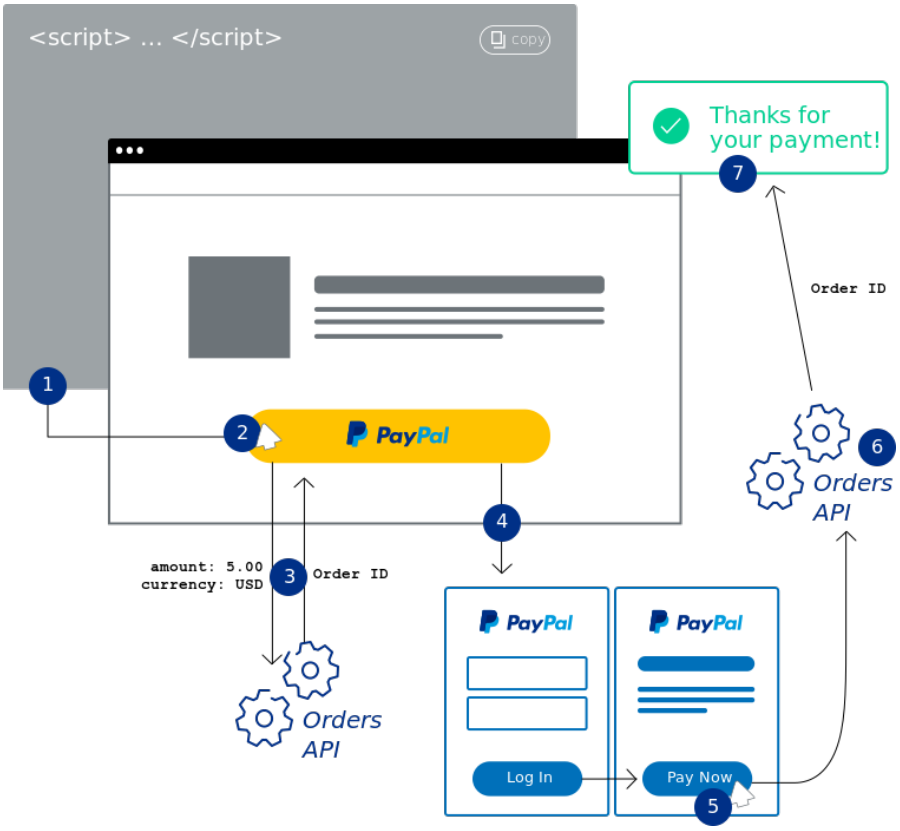
5.0.1 Payment integration

To furnish the whole flow of ordering with an option to pay in an actual way, I integrated PayPal's payment system in my project. This could be easily done with the documentation resource PayPal provided. [91]

Technical flow

The PayPal integration sets up online payment options using the PayPal JavaScript SDK, which in essence outputs payment options to the customer - either using a PayPal account or a credit/debit card. [92]

Before starting any integration, there's a need for a Sandbox Developer account. After completing the registration, the user has to generate a Sandbox client ID together with a client secret key. These will be needed to use the API provided by PayPal. [93]



gincenter

Figure 5.1: PayPal's technical flow of operation. Contains the principal steps. Source: [94]

In order to fully achieve the payment processing, I had to implement the following steps: [94]

- I needed to add PayPal's payment buttons to the UI. I reused the functionalities given by the package made specially for React, called React-Paypal. [95] React-paypal-js provides a solution to developers to abstract away complexities around loading the JS SDK. It enforces best practices by default so customers can be part of the best possible user experience. Adding the buttons to the frontend and handling their clicks looked like:

```
<PayPalButtons
 createOrder={(data, actions) =>
 handleCreateOrder(data, actions)
 }
 onApprove={(data, actions) =>
 handleApprovedOrder(data, actions)
 } />
```

- In the next step supposedly the customer clicks the payment button. This button's click handler forwards the request to our Laravel backend
- The server side uses the package provided for PayPal commands on backend. [96] After receiveing the click, we save the total price amount of the order, the currency (defaults to USD) and call the PayPal Orders API to set up a transaction.
- PayPal Checkout experience is launched
- The customer has to approve the transaction
- The server is called again to finalize the transaction. PayPal captures the order and by returning to the user, we show a confirmation message or an error message depending on how the payment went through.

Chapter 6

Summary

The whole process of developing a full-stack webshop application from scratch turned out to be extremely instructive. From being introduced to web programming at the university with literally no prior knowledge, throughout the whole journey of gaining new technological understanding and improving myself on a daily basis - all played a huge part in the desire to advance and develop further.

The thesis project started with the step of examination. Inertia.js appeared just recently in the past couple of years, hence the first period of my time was spent on understanding the whole concept of its working principle. After thorough analysis and practical testing, I could shift my focus on planning the whole project.

The application was built with three building blocks: back-end Laravel, front-end React.js and the monolith Inertia.js between. From my prior experience, time spent with in-depth structuring and applying of best-practices brings countless advantages later on in the project. I focused on creating a maintainable - and with respect on the future - a scalable project structure in both the server-side and client-side blocks.

The project had to be powered by a database, where the differences between a relational and non-relational database had to be taken in consideration. Upon reflecting on the possible outcomes, a MySQL relational database proved to be adequate for the system. The back-end development started with the configuration and implementation of the database. Designing a database schema priorly certainly helped a lot in the actual shifting of data and tables to a Laravel manipulated database.

The PHP framework Laravel established once again as powerful tool for running the server-side part of a application. I was able to implement the routing, controllers, models and database migrations in a very straightforward fashion. Laravel paired with Inertia made the whole data-flow comprehensive, the need of building an API disappeared

totally. By the help of the framework I could additionally boost the webshop with a event-based notification system, where customers receive a tracking email about their processed order.

While having a small experience in creating interactive user interfaces, React.js happened to be a great challenge as well as a great library to learn on. The speed and native feel of single page applications play a huge part in the webshop's user experience. I tried to follow best-practices regarding the frontend, this allowed me to fully create a component based application where every component has its own responsibility. Inertia and React mixed brought out the best of templating, reusable layouts proved to be of greater help in the development of use-cases.

Last but not least, the whole project revolved around the idea of an e-commerce webshop. With this in mind, the project benefited from the implementation of features as category and product browsing, user profile creation, cart based shopping together with the product ordering and payment system. The space for additional features is endless, the webshop could be improved for instance with a support system, additional payment integrations, moreover the extension of shipment options could bring a plus.

Taking every step of the development in consideration, the thesis work brought to light many essential aspects I need to further improve on. My goal for the future is to definitely stay in web development and explore technologies that assist in the creation of web applications.

Bibliography

[1] *Evolution of HTTP*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Evolution_of_HTTP.

[2] *Transport Message Exchange Pattern: Single-Request-Response*, en. [Online]. Available: https://www.w3.org/2000/xp/Group/1/10/11/2001-10-11-SRR-Transport_MEP.

[3] *HTTP Overview*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview.

[4] M. O. Arshad, *Understanding HTTP – The internet's communication protocol*, en. [Online]. Available: https://alazierplace.com/2019/05/understanding-http-the-internets-communication-protocol/.

[5] *HTTP Methods*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

[6] *Client side frameworks*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/Introduction.

[7] *Software Framework*, en. [Online]. Available: https://en.wikipedia.org/wiki/Software_framework.

[8] R. RANJAN, *What is a framework?* en. [Online]. Available: https://www.netsolutions.com/insights/what-is-a-framework-in-programming/.

[9] *Framework vs Library*, en. [Online]. Available: https://www.interviewbit.com/blog/framework-vs-library/.

[10] J. Mathew, *Inversion of control*, en. [Online]. Available: https://medium.com/@jalena.mathew.21/inversion-of-control-dependency-injection-f5a59005306.

[11] P. Böllhoff, *Framework vs library*, en. [Online]. Available: https://kruschecompany.com/framework-vs-library/.

[12] Neoteric, *Neoteric, Single Page application vs Multiple page application*, en. [Online]. Available: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58.

[13] *Single page application*, en. [Online]. Available: https://en.wikipedia.org/wiki/Single-page_application.

[14] *SPA (Single-page application)*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/SPA.

[15] *What Is a Single Page Application (SPA)?* en. [Online]. Available: https://www.outsystems.com/glossary/what-is-single-page-application/.

[16] *What is a single page application?* en. [Online]. Available: https://www.techferry.com/articles/single-page-applications.html.

[17] *App shell*, en. [Online]. Available: https://developer.chrome.com/blog/app-shell/.

[18] K. Lawson, *What is a single page application*, en. [Online]. Available: https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application.

[19] *Choose Between Traditional Web Apps and Single Page Apps (SPAs)*, en. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps.

[20] J. Reinink, *Who is it for?* en. [Online]. Available: https://inertiajs.com/who-is-it-for.

[21] J. Reinink, *Introducing Inertia.js*, en. [Online]. Available: https://reinink.ca/articles/introducing-inertia-js.

[22] J. Reinink, *How it works?* en. [Online]. Available: https://inertiajs.com/how-it-works.

[23] *XMLHttpRequest*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[24] *The Ultimate Guide to Inertia.js*, en. [Online]. Available: https://kinsta.com/knowledgebase/inertia-js/.

[25] J. Reinink, *The protocol*, en. [Online]. Available: https://inertiajs.com/the-protocol.

[26] D. Wabuge, *What is a Web Database and What Types are There?* en. [Online]. Available: https://techjury.net/blog/what-is-a-web-database/.

[27] *NoSQL vs. SQL: Important Differences Which One Is Best for Your Project*, en. [Online]. Available: https://www.upwork.com/resources/nosql-vs-sql.

[28] B. Anderson, " SQL vs. NoSQL Databases: What's the Difference?" en, [Online]. Available: https://www.ibm.com/cloud/blog/sql-vs-nosql.

[29] *MongoDB - What is NoSQL?* en. [Online]. Available: https://www.mongodb.com/nosql-explained.

[30] *What is NoSQL?* en. [Online]. Available: https://www.softwaretestinghelp.com/sql-vs-nosql.

[31] T. Hargreaves, *Database considerations*, en. [Online]. Available: https://medium.com/@sidetracking/database-considerations-8444df88ea2e.

[32] Z. Y. Ruihan Wang, "SQL vs NoSQL: A Performance Comparison," en, [Online]. Available: https://www.cs.rochester.edu/courses/261/fall2017/termpaper/submissions/06/Paper.pdf.

[33] *Database introduction*, en. [Online]. Available: https://laravel.com/docs/9.x/database#introduction.

[34] *MVC*, en. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/MVC.

[35] *How Laravel implements MVC and how to use it effectively*, en. [Online]. Available: https://blog.pusher.com/laravel-mvc-use/.

[36] *Why Laravel?* en. [Online]. Available: https://laravel.com/docs/9.x#why-laravel.

[37] *Starter kits*, en. [Online]. Available: https://laravel.com/docs/9.x/starter-kits.

[38] *Database*, en. [Online]. Available: https://laravel.com/docs/9.x/database.

[39] *Writing seeders*, en. [Online]. Available: https://laravel.com/docs/9.x/seeding#writing-seeders.

[40] *Eloquent*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent.

[41] *One-to-one*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent-relationships#one-to-one.

[42] *One-to-many*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent-relationships#one-to-many.

[43] *Has-one-through*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent-relationships#has-one-through.

[44] *Has-many-through*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent-relationships#has-many-through.

[45] Staudenmeir, *Eloquent-has-many-deep*, en. [Online]. Available: https://github.com/staudenmeir/eloquent-has-many-deep.

[46] *Eloquent resources*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent-resources.

[47] *Routing*, en. [Online]. Available: https://inertiajs.com/routing.

[48] *Basic routing*, en. [Online]. Available: https://laravel.com/docs/9.x/routing#basic-routing.

[49] *Named routes*, en. [Online]. Available: https://laravel.com/docs/9.x/routing#named-routes.

[50] *Ziggy*, en. [Online]. Available: https://github.com/tighten/ziggy.

[51] *Grouped routes*, en. [Online]. Available: https://laravel.com/docs/9.x/routing#route-groups.

[52] *Route model binding*, en. [Online]. Available: https://laravel.com/docs/9.x/routing#route-model-binding.

[53] *Controllers - Introduction*, en. [Online]. Available: https://laravel.com/docs/9.x/controllers#introduction.

[54] *Responses*, en. [Online]. Available: https://inertiajs.com/responses.

[55] *Server side validation*, en. [Online]. Available: https://inertiajs.com/forms#server-side-validation.

[56] *Shared data*, en. [Online]. Available: https://inertiajs.com/shared-data.

[57] *Database notifications*, en. [Online]. Available: https://laravel.com/docs/9.x/notifications#database-notifications.

[58] *Formatting mail messages*, en. [Online]. Available: https://laravel.com/docs/9.x/notifications#formatting-mail-messages.

[59] *Customizing the sender*, en. [Online]. Available: https://laravel.com/docs/9.x/notifications#customizing-the-sender.

[60] *Events*, en. [Online]. Available: https://laravel.com/docs/9.x/events.

[61] *Events - Introduction*, en. [Online]. Available: https://laravel.com/docs/9.x/events#introduction.

[62] *Registering events and listeners*, en. [Online]. Available: https://laravel.com/docs/9.x/events#registering-events-and-listeners.

[63] *Defining events*, en. [Online]. Available: https://laravel.com/docs/9.x/events#defining-events.

[64] *Defining listeners*, en. [Online]. Available: https://laravel.com/docs/9.x/events#defining-listeners.

[65] *Dispatching events*, en. [Online]. Available: https://laravel.com/docs/9.x/events#dispatching-events.

[66] *Eloquent events*, en. [Online]. Available: https://laravel.com/docs/9.x/eloquent#events.

[67] *Authentication Quickstart*, en. [Online]. Available: https://laravel.com/docs/9.x/authentication#authentication-quickstart.

[68] *Laravel Breeze*, en. [Online]. Available: https://laravel.com/docs/9.x/starter-kits#laravel-breeze.

[69] *Authentication introduction*, en. [Online]. Available: https://laravel.com/docs/9.x/authentication#introduction.

[70] *Tailwind CSS*, en. [Online]. Available: https://tailwindcss.com/.

[71] *Wikipedia - Tailwind CSS*, en. [Online]. Available: https://en.wikipedia.org/wiki/Tailwind_CSS.

[72] *Configuration*, en. [Online]. Available: https://tailwindcss.com/docs/configuration.

[73] *Google fonts*, en. [Online]. Available: https://fonts.googleapis.com/..

[74] *Utility-first framework*, en. [Online]. Available: https://tailwindcss.com/docs/utility-first.

[75] *Responsive design*, en. [Online]. Available: https://tailwindcss.com/docs/responsive-design.

[76] *Tailwind UI KIT*, en. [Online]. Available: https://tailwinduikit.com/.

[77] *Tailwind UI compoennts*, en. [Online]. Available: https://tailwindui.com/components/.

[78] *HyperUI*, en. [Online]. Available: https://www.hyperui.dev/.

[79] *Kitwind kit*, en. [Online]. Available: https://kitwind.io/.

[80] *javatpoint - React.js*, en. [Online]. Available: https://www.javatpoint.com/reactjs-tutorial.

[81] *react.js*, en. [Online]. Available: https://reactjs.org/.

[82] *Introducing JSX*, en. [Online]. Available: https://reactjs.org/docs/introducing-jsx.html.

[83] *React Virtual DOM*, en. [Online]. Available: https://www.codecademy.com/article/react-virtual-dom.

[84] *What is a Virtual DOM?* en. [Online]. Available: https://mfrachet.github.io/create-frontend-framework/vdom/intro.html.

[85] *Virtual DOM*, en. [Online]. Available: https://reactjs.org/docs/faq-internals.html.

[86] *Components and props*, en. [Online]. Available: https://reactjs.org/docs/components-and-props.html.

[87] *useContext*, en. [Online]. Available: https://beta.reactjs.org/apis/react/useContext.

[88] *Pages - Introduction*, en. [Online]. Available: https://inertiajs.com/pages#top.

[89] *Default layouts*, en. [Online]. Available: https://inertiajs.com/pages#default-layouts.

[90] *Form helper*, en. [Online]. Available: https://inertiajs.com/forms#form-helper.

[91] *Checkout*, en. [Online]. Available: https://developer.paypal.com/docs/checkout.

[92] *Javascript SDK*, en. [Online]. Available: https://developer.paypal.com/sdk/js/.

[93] *Sandbox*, en. [Online]. Available: https://developer.paypal.com/tools/sandbox/accounts/.

[94] *Integrate Checkout*, en. [Online]. Available: https://developer.paypal.com/docs/checkout/standard/integrate/.

[95] *React PayPal*, en. [Online]. Available: https://github.com/paypal/react-paypal-js.

[96] *Laravel Paypal*, en. [Online]. Available: https://github.com/srmklive/laravel-paypal.

Appendix A

Source code

The source code is available on GitHub, on the following link:
https://github.com/gburapeter/WebshopThesisGBP

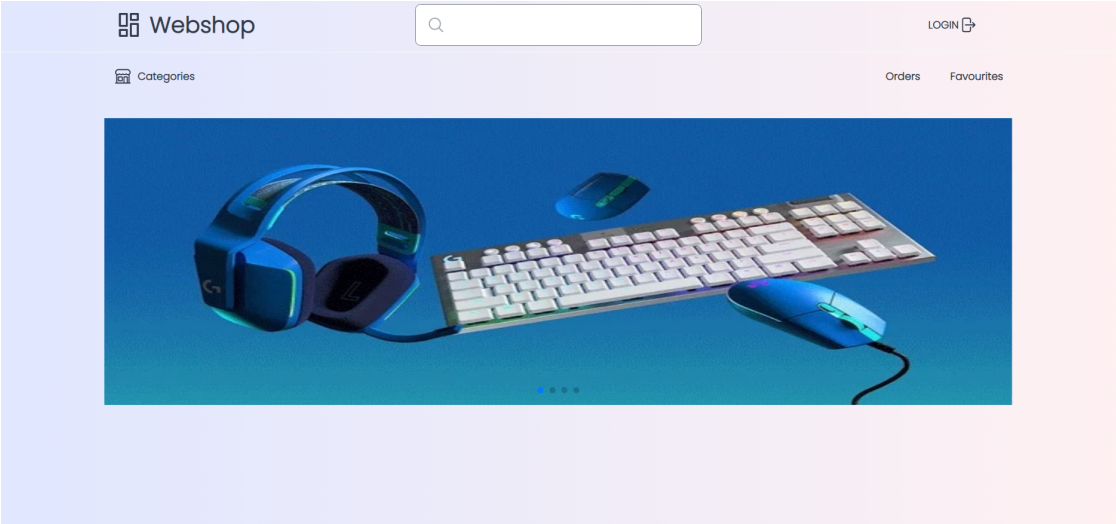
The github repository contains all the files and folders of the project. (Laravel + React). There is a readme.md file on the main page of the repository which describes the required steps to set up a local environment to test the webshop.

Appendix B

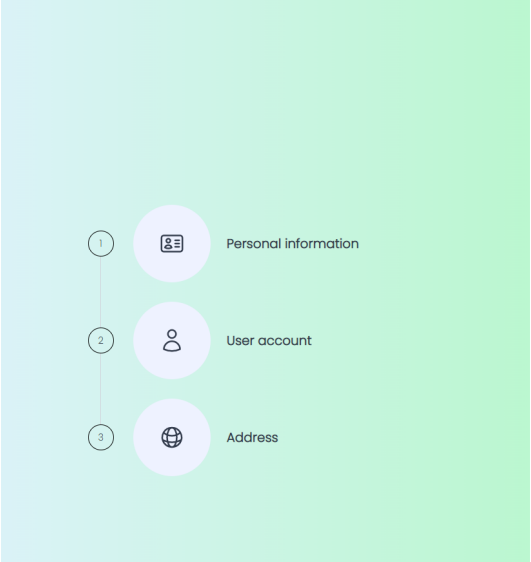
Webshop application

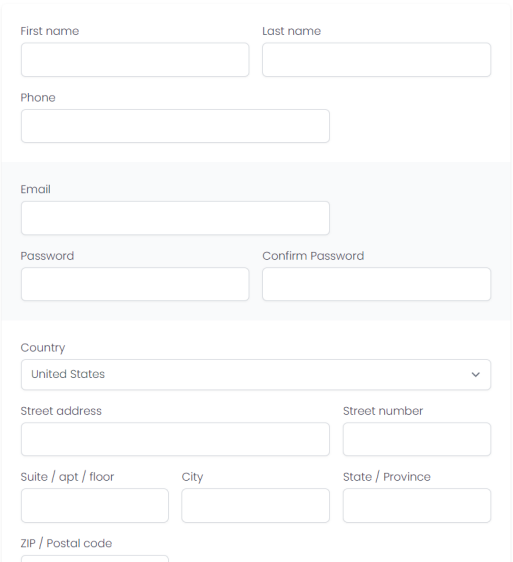
This appendix contains screenshots taken while browsing the webshop application. The webshop is constantly under further development, hence small differences in design between the screenshots appended here and the actual source code might be possible.

Welcoming page (Guest)

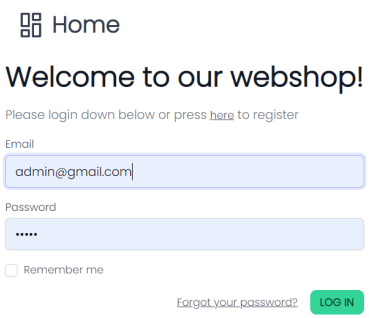


Register flow

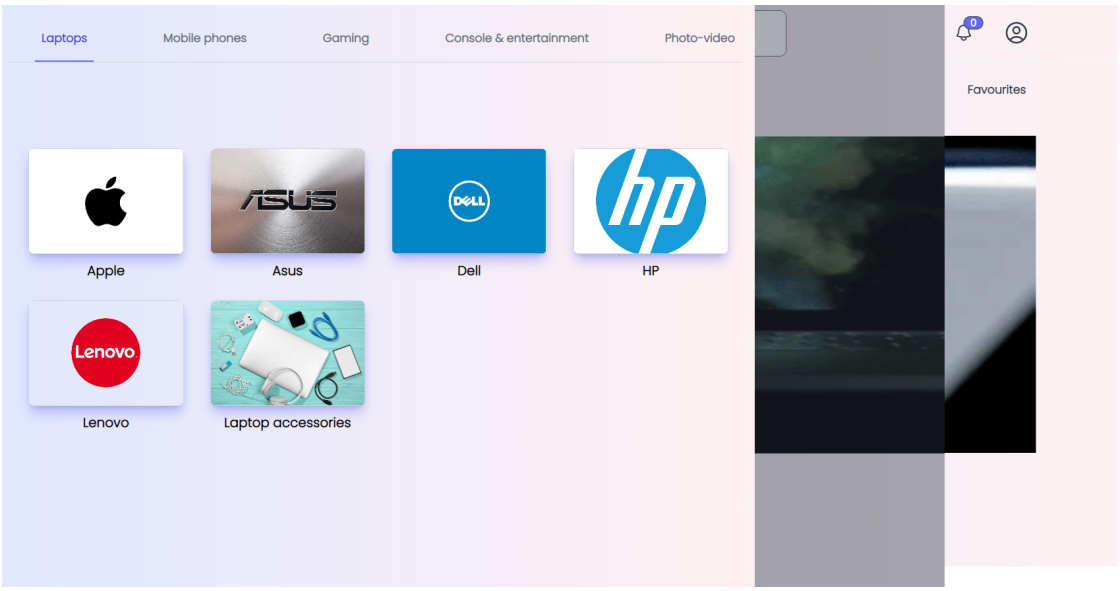




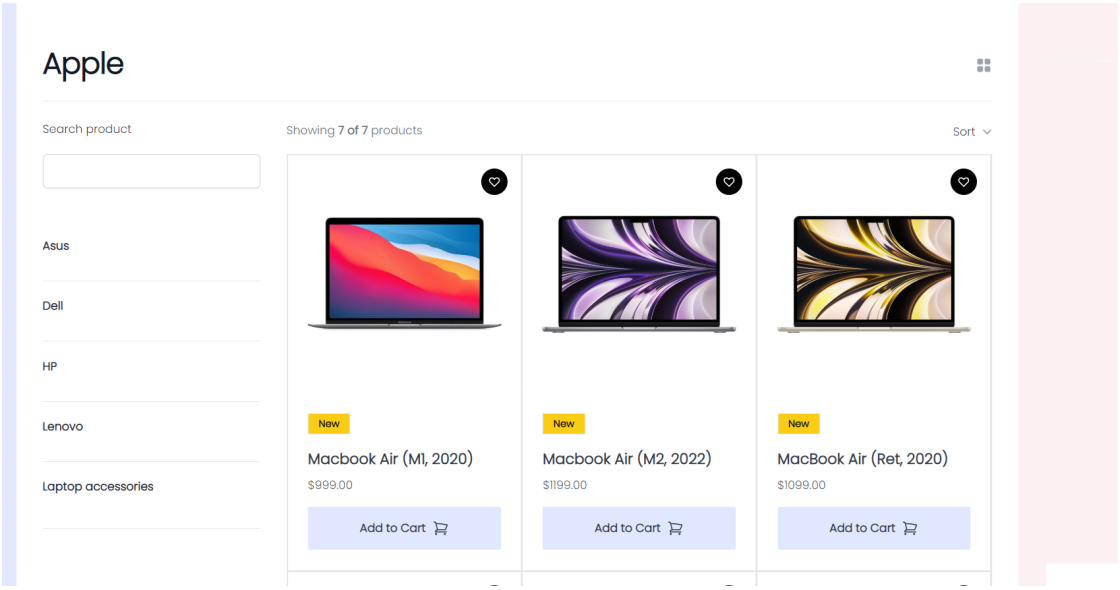
Login flow



Categories drawer



Products list



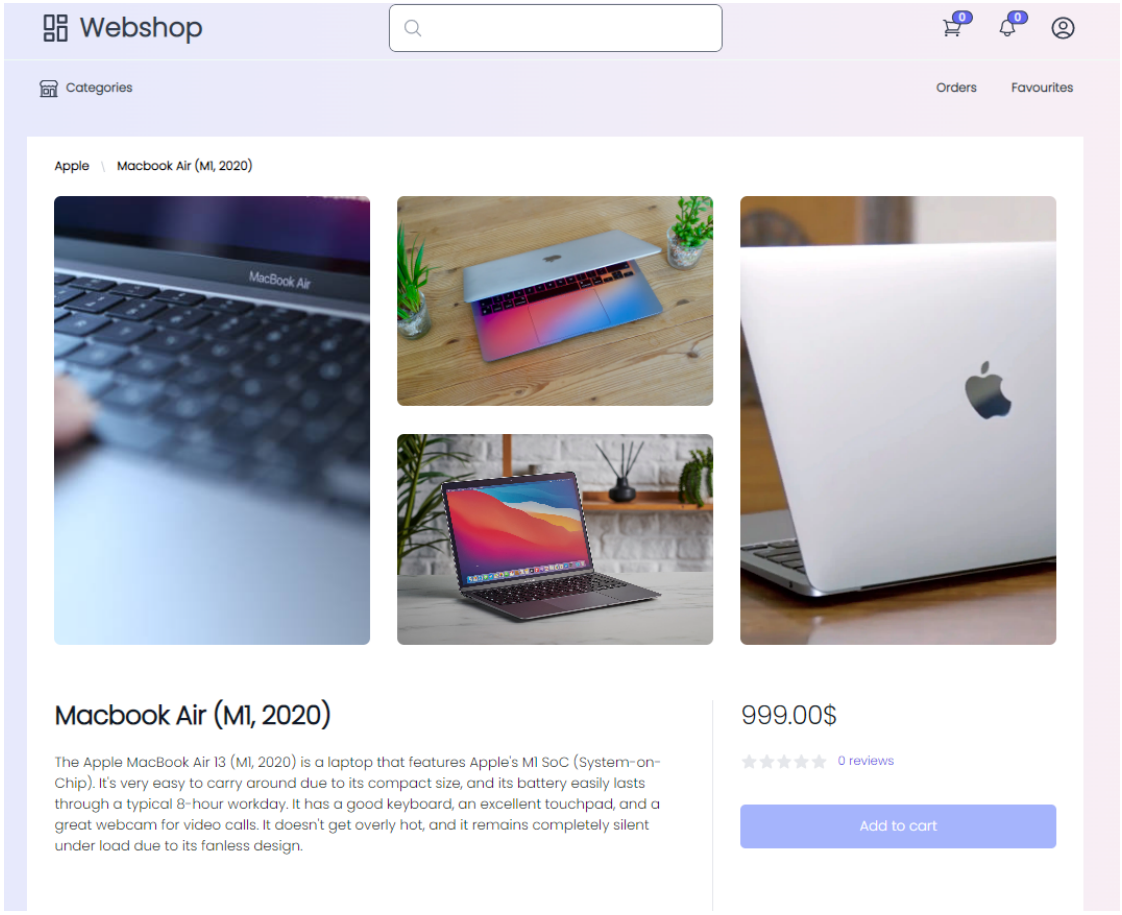
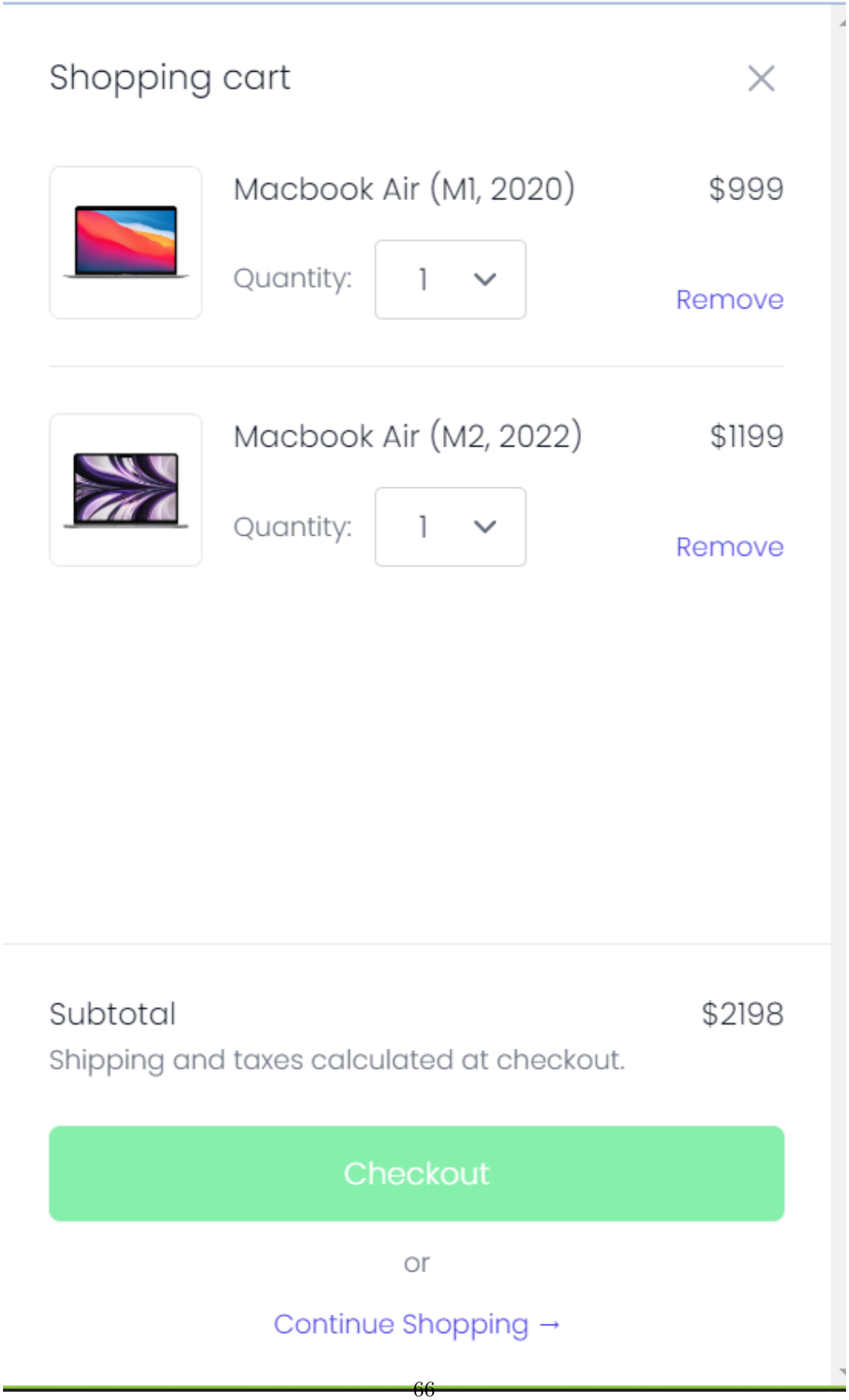
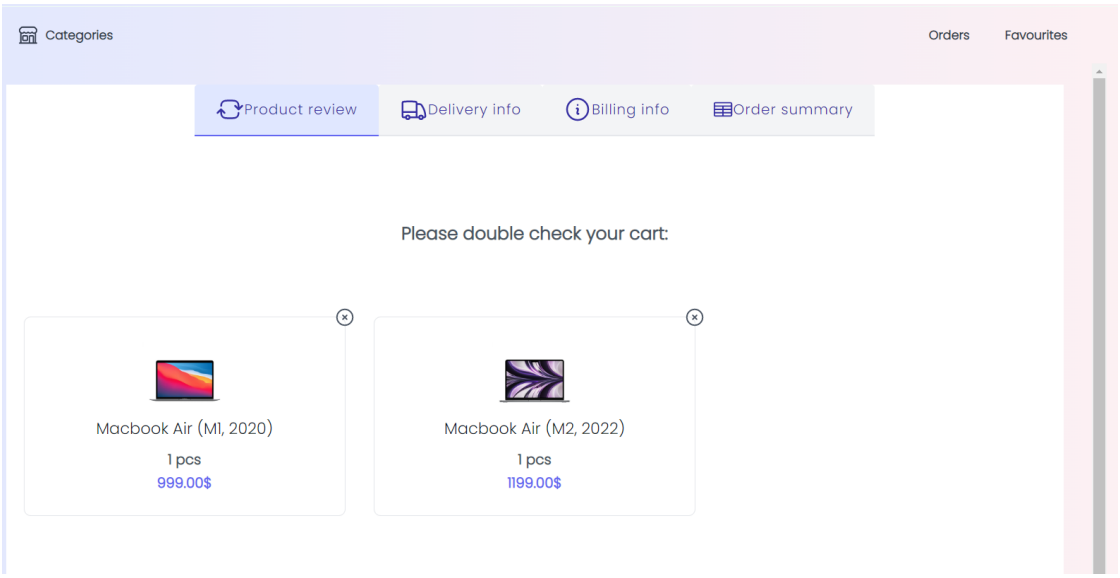


Figure B.1: Specific product page

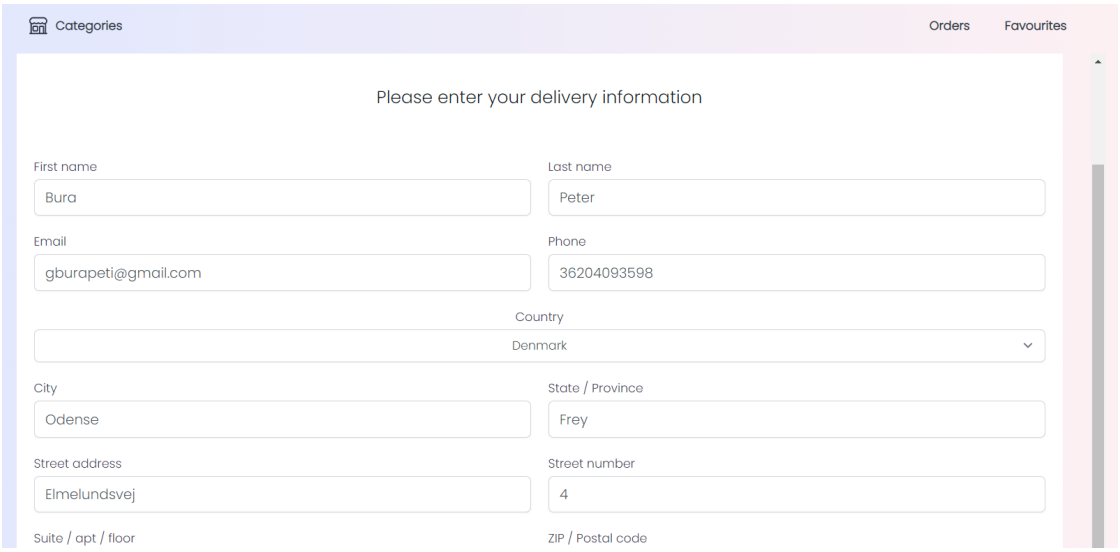
Shopping cart



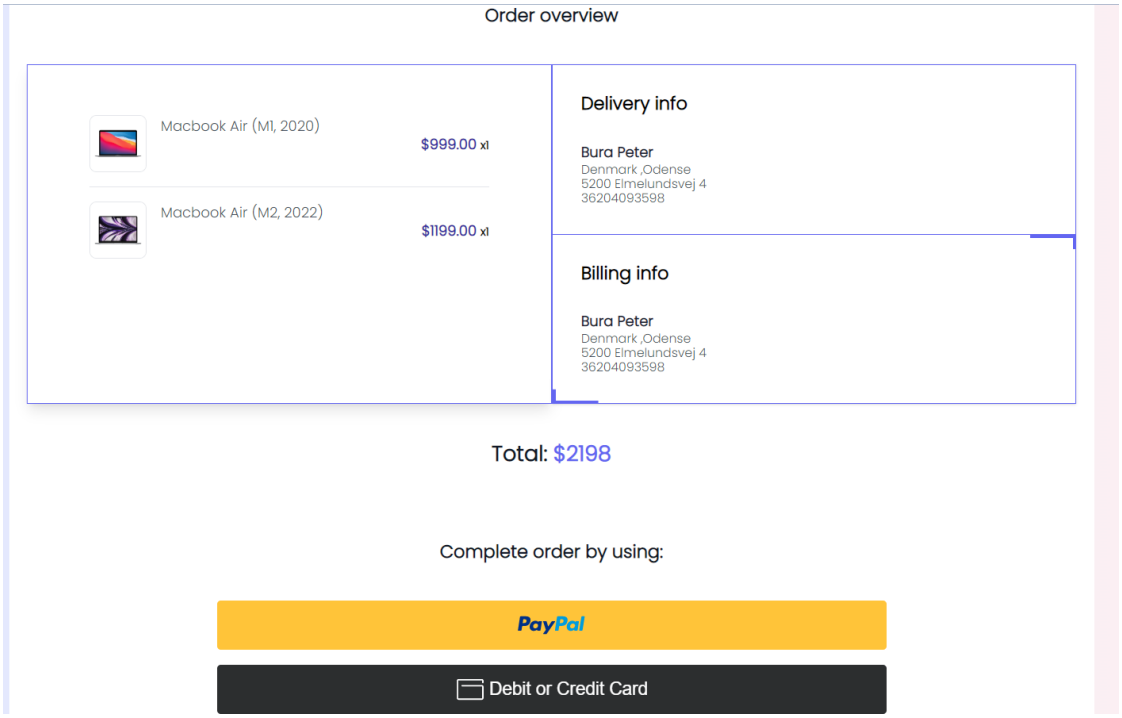
Checkout flow - first step



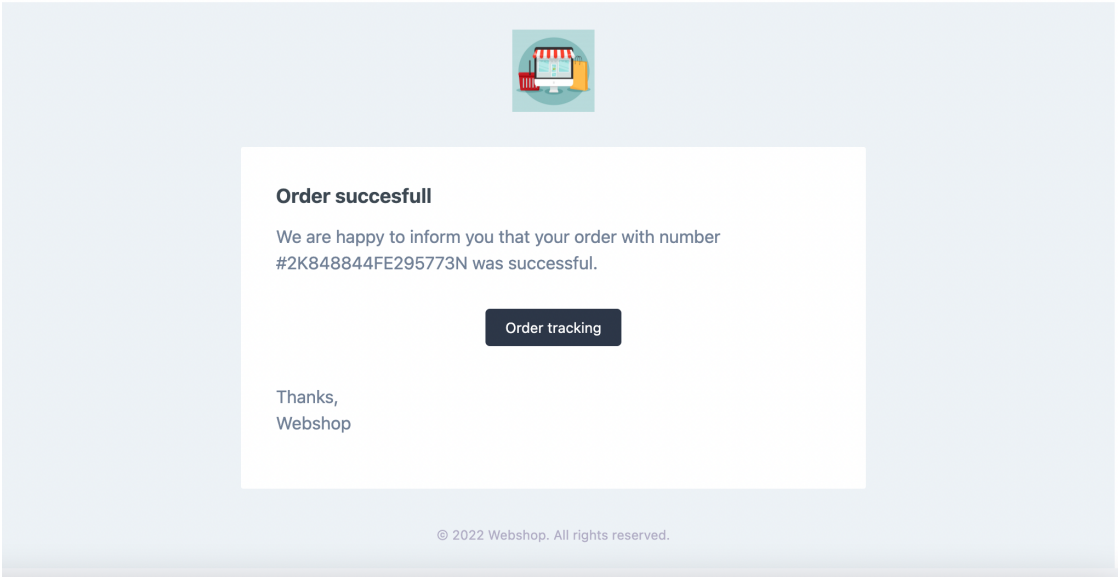
Checkout flow - billing step



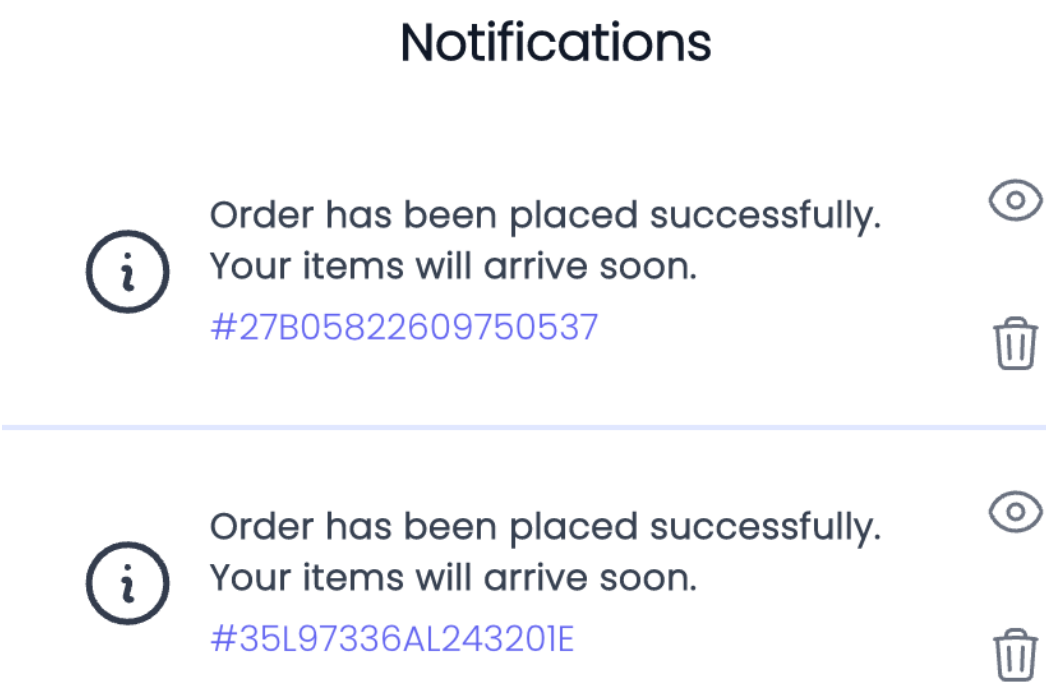
Payment step



Order email



Order notification



Order tracking page