Exam Report

Rust - Website

Erhvervsakademi SydVest, Computer Science third semester exam project

---

Team members:
David Kalatzis
Greg Charyszczak
Kevin Luu

Github repository:
https://github.com/Skelly-Co/RUST-Web-Application

2019 December 19

Table of Content

1. Introduction

1.1 Background

The awareness and stability of a brand depends a lot upon the impression it reflects to the people surrounding it. With so many brands and companies which we view and interact with on a daily basis, it is hard for a startup company to get some needed attention from consumers.

In order to prevent a loss of a brand like this from happening, RUST tasked us with making a clothing shop, but most importantly to create a site that stands out from the rest, and that could prove as a place to direct people to learn more about RUST as a company, and what makes it unique compared to other brands of its kind. Since the company was still a startup the shop itself was not of utmost importance, it stood instead more as a future plan, for once the brand grows bigger, and instead the vision and feel of the site was the crucial part of this project, in order to build the brand first and mark it to people conscious, before it really starts as a serious business.

1.2 Problem Statement

With the amount of brands and companies currently in the market, it is very difficult for a newly developed company to rise, and have a healthy enough condition to gain popularity amongst all the others brands. Couple that with the fact it is very rare and difficult for a clothing site to match the style of the clothes the brand is selling, and you have a problem which could prove difficult to solve.

1.3 Product Vision

To achieve the goal of the company, we decided to make a web application, which would promote the company and market to consumers what RUST is all about. The web application would most importantly have a landing page which would grab the user's attention and intrigue them to look deeper, and then inform them through a unique and interesting vision page about the brand itself and the people behind it. Further than that it would include a shop component for use in the future, where users

will be able to easily view in detail, in the specific vibe and atmosphere RUST has created, all of the products sold to them by the company, and with ease be able to purchase them in the color, size and quantity in which they prefer.
With all that in mind, we came to the following product vision:

"*For people who need a unique and standalone brand to represent themselves in, the RUST website allows for a first look and introduction to a different and exclusive brand, through an informative and atmospheric webpage.*"

1.4 Technical Background

This semester equipped us with the necessary skills in order to implement our vision, by taking part in the Software Development Method (SDM) courses, Software Development Programming (SDP) courses and Computer Network and Distributed Systems (CDS) courses, we gained the knowledge and tools to successfully plan, design and implement a web shop application, with the required features asked by the client.

2. SDP - CDS

2.1 SDP -CDS Introduction

Before we started the initial development of our software, we had to decide upon the design and implementation of the program itself, along with the more in depth requirements that we would have to keep on our minds during the development process.

Things such as the infrastructure that would be used, the security measures that would have to take place, so that we could ensure the safety of the application along with its data. As well as the whole architecture that would have to take place and have to be designed, in order to accomplish a good level of quality and coherence throughout the whole project, and provide the best possible product the team could hope to achieve.

During this semester we have been lectured on both the software development aspect of programming, as well as the computer network and distribution component.

On the SDP front we were lectured about web application development. In order to display our knowledge of that component, we have used Angular 8 as our framework, combined with a Restful API, and using Clean Architecture as our main source of direction in the design of our application. We also applied the skills we acquired when it comes to the client side of things, such as HTML CSS and Typescript.

As for the CDS component our solution encompasses both security through techniques that were taught such as token authentication, the usage of CORS, and deployment to the cloud through services such as Microsoft's Azure.

2.2 System Architecture

In order to develop the project, a specific system architecture had to be chosen, so that we could have a clear and definitive way of how we should structure the application, and define the way the user will interact with it once it's launched in the production environment.

Clean architecture was the selected architecture, as it fitted perfectly to the conditions of our system, and it was also the architecture which we learned about, during the third semester at the academy. It handed us a very clear definition of how each layer should be structured in our system, which made it easy for each team member to work independently on each module, while still following a very similar structure with everybody else involved, thus keeping the possibilities of errors appearing very limited and rare, and the code and design well efficient and comprehensive.

Other considerations had to be taken as well, including the tier system, our Restful API, the data model, design patterns and much more, all of which we discuss in depth below.

2.2.1 Clean Architecture

Our application as stated before, uses the Clean Architecture(otherwise known as Onion Architecture), which separates the system into three main layers, the core, the infrastructure and the UI.

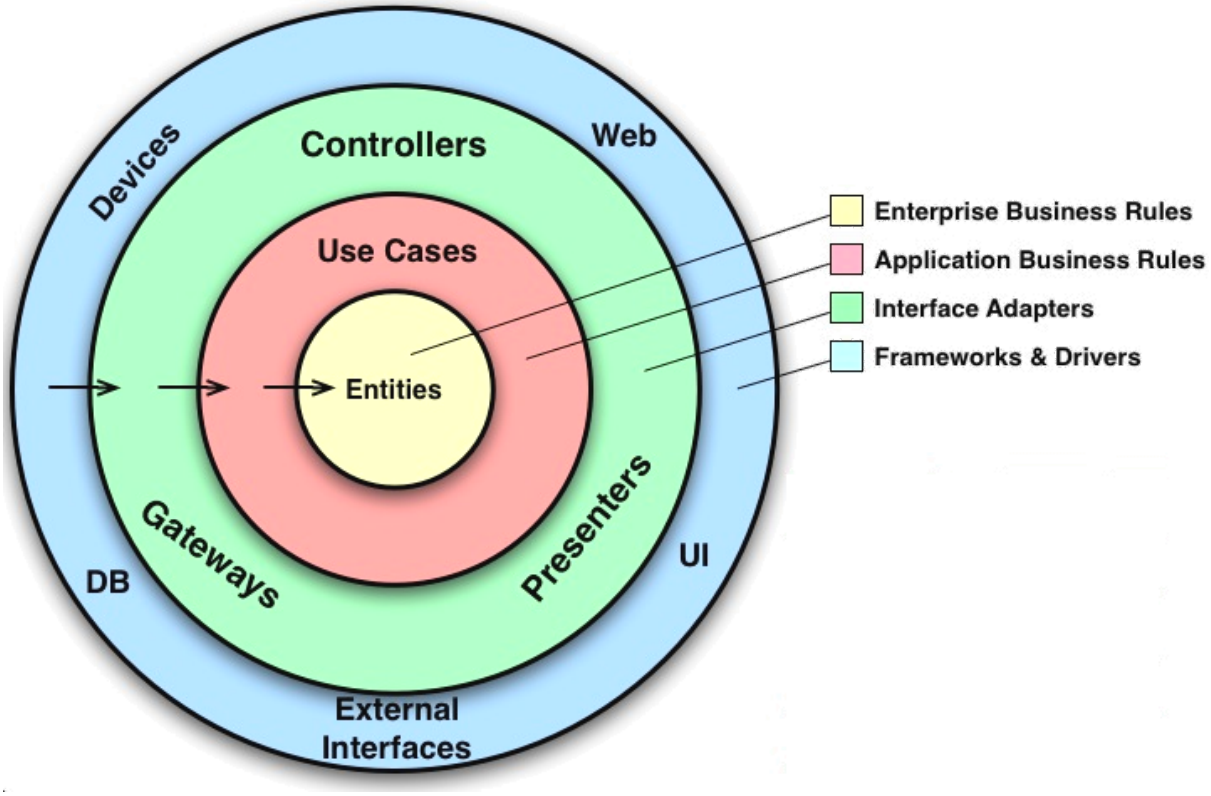
The core acts as the logic of the application where functions are being called from and processed, as well as checking for unexpected errors. It is the main and most integral part of the whole system.

As for the infrastructure, it is the module that connects and responds to our database, by supplying it with information about the way it should modify the data.

Finally the UI, or otherwise the RestAPI in our case acts as the middleman between the client side and our backend, providing it with important requests made from the user's perspective, and handing it to the core.

Everything in our architecture depends on our core layer**(Figure 1)**, which is the base of the whole application. The core has no dependencies to any other layer, meaning that we are able to swap in and out different layers for the UI and Infrastructure, while keeping the core of our application intact and completely unchanged.

Figure 1. Example of Clean Architecture



2.2.1.1 Reflections on Clean Architecture

We reflected on the architecture and the benefits that are provided by using such an architecture, and came up with the following advantages that we acquire from using the Clean Architecture.

- **Flexible**
  Firstly the clear benefit of using Clean Architecture is the fact that every layer depends upon one and only layer, which is the core, and they are isolated in individual modules, while the core itself does not depend upon anybody, meaning that each part outside of the core can be easily taken out and replaced, so our database or UI are easily replaceable and adjustable, depending on the need of our client.
- **Secure**
  There is also a very strict use case when it comes to Clean Architecture, there are not many choices in how you can use this architecture, which results in an easy to handle system, where everything is clear and simple to the developers, because of that it is hard to create some sort of error in the system, or not follow the laid out code design.
- **Cohesive**
  Because of the clear way this architecture works, it is very easy to understand the way it behaves without having to go into many technical details, its intended usage is very clear and easy to read and understand.
- **Testability**
  Also because our core entities are completely separate from everything else in our system, they are very easily testable, and they do not get hampered by external things such as databases and services, which proves useful even more because the application is written with test driven development in mind.
- **Future proof**
  It acts as a great monolith in case we would need to split our system into microservices in the future for improved control.

2.2.2 Tier System

Our software was developed as a three-tier architecture system, meaning that we have a presentation layer(interface), which runs on the client's side, an application layer(business logic) in the form of a RestAPI, and a data layer which is stored on a server, in our case on an Azure server. Splitting up these three components into different locations represents a three-tier architecture, which brings numerous benefits to help us keep up the pace required to deliver a compelling software product for our clients.

A clear first example of the advantages that such an architecture possess, is the fact that you are capable of updating and changing the software in one part of the tier without impacting in any way the other tiers. Which also allows each team member to work in one area of the system, without the fear of having conflicts in between. Furthermore we are able to scale the application to how we see fit, for example our separated application layer, can be deployed and use multiple databases, without being locked into one particular software. Additionally this division of modules provides reliability and independence of the underlying services. And finally it gives us an ease of maintenance in the code base, managing the presentation layer and application layer separately is much easier that way, especially in a team environment.

2.2.3 REST API

We decided to implement a RESTful web service since the representational state transfer (REST) is one of the easiest API to use, being that it is based on the HTTP protocol. The Rest API takes the role of the User Interface (UI) in the Clean Architecture of our application. As a kind of console application, which is part of Visual Studio, the Rest API is fully replaceable by any other UI API and adds the functionality to obtain data and generate operations on that data in the formats of XML and JSON. The used HTTP operations in this application are GET, PUT, POST and DELETE which allow for the functionalities of reading, creating, editing and deletion of data.

A client, either a user or a software performs an operation in order to manipulate or show a resource. This usually happens in the frontend, where the client can for example use a browser or any other form of application to call for an HTTP request. The operation is sent to the backend where the service calls the repository, then the repository in the backend performs the operation, which directly talks to the database. By following this said approach, we ensured that the data is being processed

in a secured and encapsulated environment. The application is segregated into different layers, which means that the communication of a layer happens only between one to two layers. For the frontend (client) only the actual product is visible.

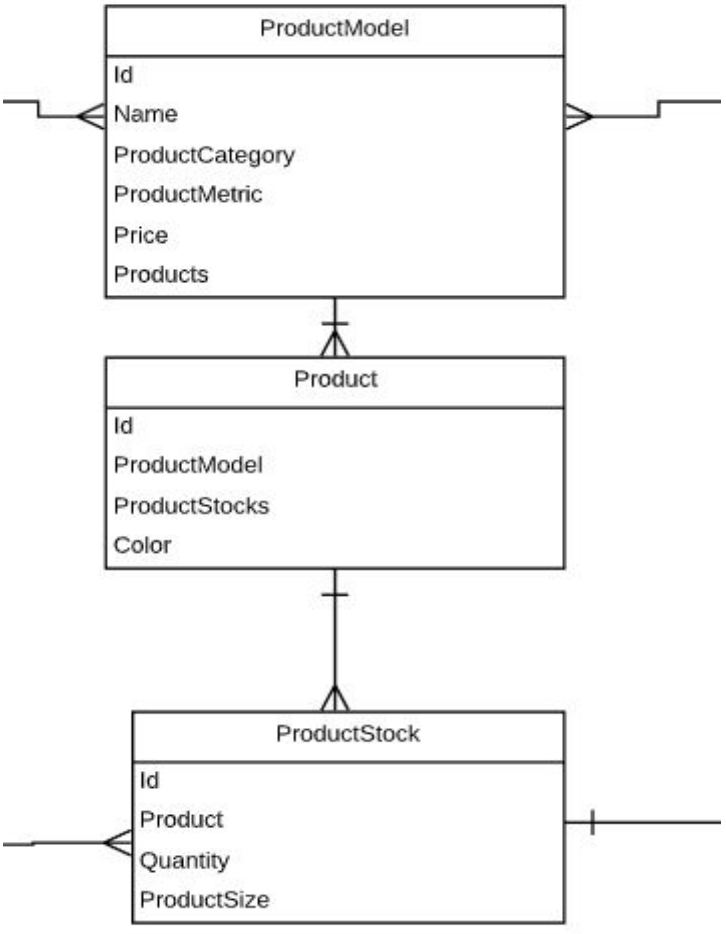
Meaning that for example whenever the client has a request sent to the backend, only our RestAPI will be responsible for receiving that data, processing it, and passing it through to the further layers, and back again to the client, in that way we are keeping everything separated and responsible only for the functionalities for which they are made for.

2.3 Conceptual Data Model

Since clean architecture is based upon the entities created for the project, we had to visualize the way they interact with each other, how they are connected, and the properties each of them hold, so that we could structure the system as best as possible, in a way that is easy to understand and expandable for future development.

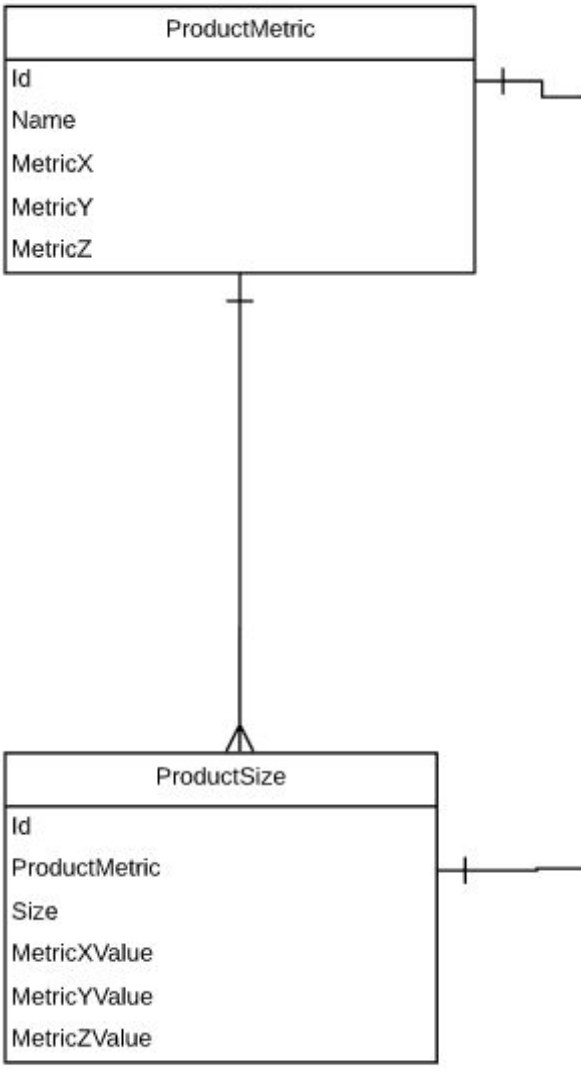
One of the major problems that had to be solved early on was the way we depict a product in the shop list, and how that differs from the actual stock which depends upon the specified size and color.

To solve this issue we separated the product into 3 major parts, first we have the product model, which is a schema for the whole product to follow, here is where we get the data of what kind of product to display along with the price and name. We then have a product entity which depicts the product with a specified color chosen by the user, and finally we have the product stock, which gives us information on the quantity of a specific product, along with the size it has.



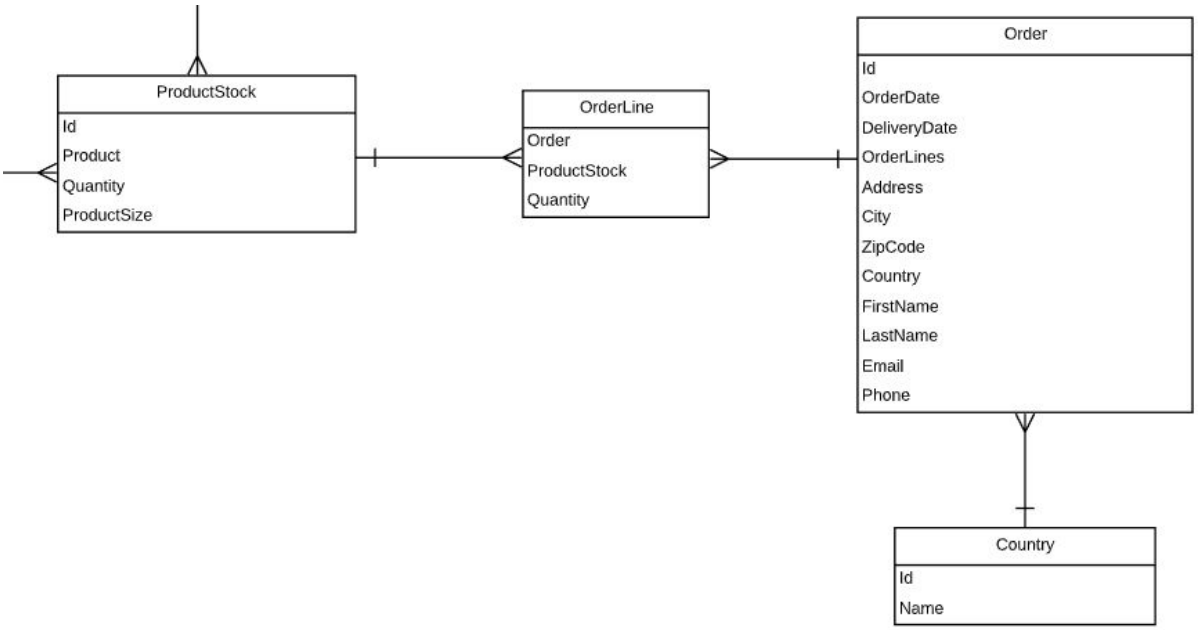
The next design issue that we faced, was the fact that each type of product has a different size metric, for example pants are measured differently compared to a shirt, or a belt which only has one measurement needed.

We managed to fix this by once again separating the product's metric and it's size, now every product size has a specific kind of metric which indicates to the size how it should behave, and in what way to display the metric data.

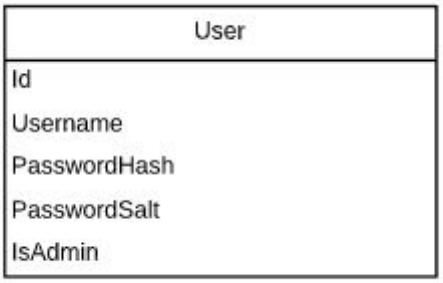


Finally we had to consider the way the orders play a factor in our application, what we came around to was creating an entity order which envelopes all of the user's order details, such as their name, email, address, phone number and so on, and then an order line which would contain the order along with the product stock the user has chosen. A minor thing to note is the fact that the country is a separate entity, the reason for this is the fact that we have to pass a lot of data to that entity, which would be

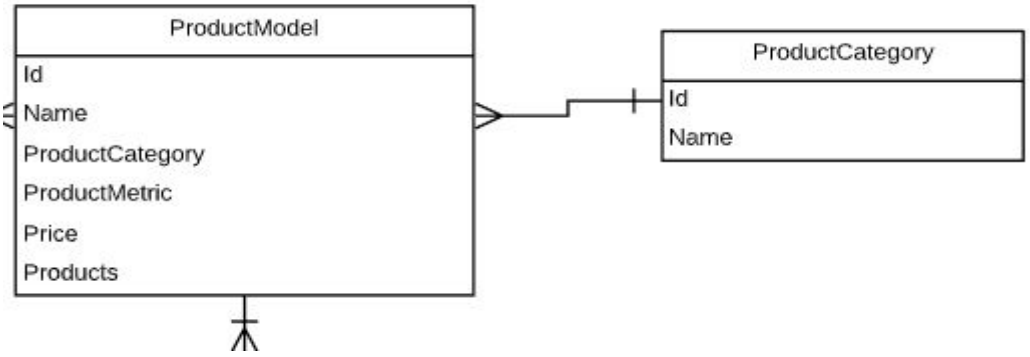
modified and updated further in the future, so separating it made that a lot easier to achieve.



There are a few other simpler connections and entities to comprehend, such as the user entity, which is mainly used as a way to authenticate our application, and is completely separate from any other entity.



And the product category which is connected directly to the product model, and is used to identify the category in which the model belongs to, which is later used for sorting functions on the client side.



The whole conceptual data model, depicting the way each entity in our system is connected and behaves can be seen in the **Appendix D**.

2.4 Design Patterns - Design Argumentation

2.4.1 Design Patterns

Dependency Injection

The use of dependency injection enables the development of loosely coupled code. Through this design pattern we are able to decrease tight coupling between our classes, by reducing the amount of information a class knows about its dependencies, which in turn makes the class flexible, and also easily testable for use in unit tests. This allows us to use these dependencies across our system, without the components using them having to know how they are created, making the code more readable and reusable, the dependency is no longer locked onto a class, thus the component can also be configured differently or used in a different context without the need to change the code.

A common example of this pattern in work is in our services, which depend upon the domain interfaces, so in order to make this work, we simply provide a variable of that repository's type, and inject it through the constructor of the said class.

```
private readonly ICountryRepository _countryRepository;


public CountryService(ICountryRepository countryRepository)
{
    _countryRepository = countryRepository;
}
```

Dependency Inversion

When it comes to interacting between high level modules, in our case the application services interacting with the repositories of the infrastructure, they should be easily reusable and unaffected by changes in other modules, in order to achieve this effect we had to introduce some abstraction that would decouple those components from each other, which in our case comes in the form of interfaces.

In this example we have a class called ProductService which belongs in the core layer of our architecture, that said class needs to call methods on a repository class, but since the ProductService class belongs in the core layer, it shouldn't directly depend upon other layers, such as the infrastructure.

For that reason we have created an interface, which acts as a level of abstraction so that the service can interact with a repository without

having to know about it, nor having the core layer depend on the infrastructure layer.

```
namespace RUSTWebApplication.Core.DomainService
{
    public interface IProductRepository
    {
        Product Create(Product newProduct);

        Product Read(int productId);

        IEnumerable<Product> ReadAll();

        Product Update(Product updatedProduct);

        Product Delete(int productId);
    }
}
```

That interface is then implemented by a repository, and is then used from the service class, combined with the help of dependency injection, to call the necessary functions.

Observable Pattern

This design pattern was used throughout the project's frontend aspect whenever we had to send an HTTP request to the backend, the benefit created from this pattern is the fact that normally, if we had to send a request to the backend the whole application would freeze awaiting a response, but with an observable we are able to subscribe to that method and allow the website to function normally, and once the backend has responded the observer will get notified that the observable has completed the subscribed method, and it will execute the needed code.

Here we are displaying a case were such an observable is used, first we call the method and pass the required parameters, after which we call a function from the productModel service and pass the parameters, and then subscribe to that function awaiting the response from the backend, while our application is still functioning normally. When the response is sent back, we just execute the code inside the subscribe method, and pass in the required data from the backend to the frontend part of our website.

```
getProductModels(currentPage: number, categoryType: string): void{
  this.currentPage = currentPage;
  this.categoryType = categoryType;
  this.productModelService.getProductModels(currentPage, this.itemsPerPage, categoryType)
  .subscribe(filteredList => {this.productModels = filteredList.data; this.totalPages = filteredList.totalPages;});
}
```

2.4.2 Design Argumentation

Shared Components

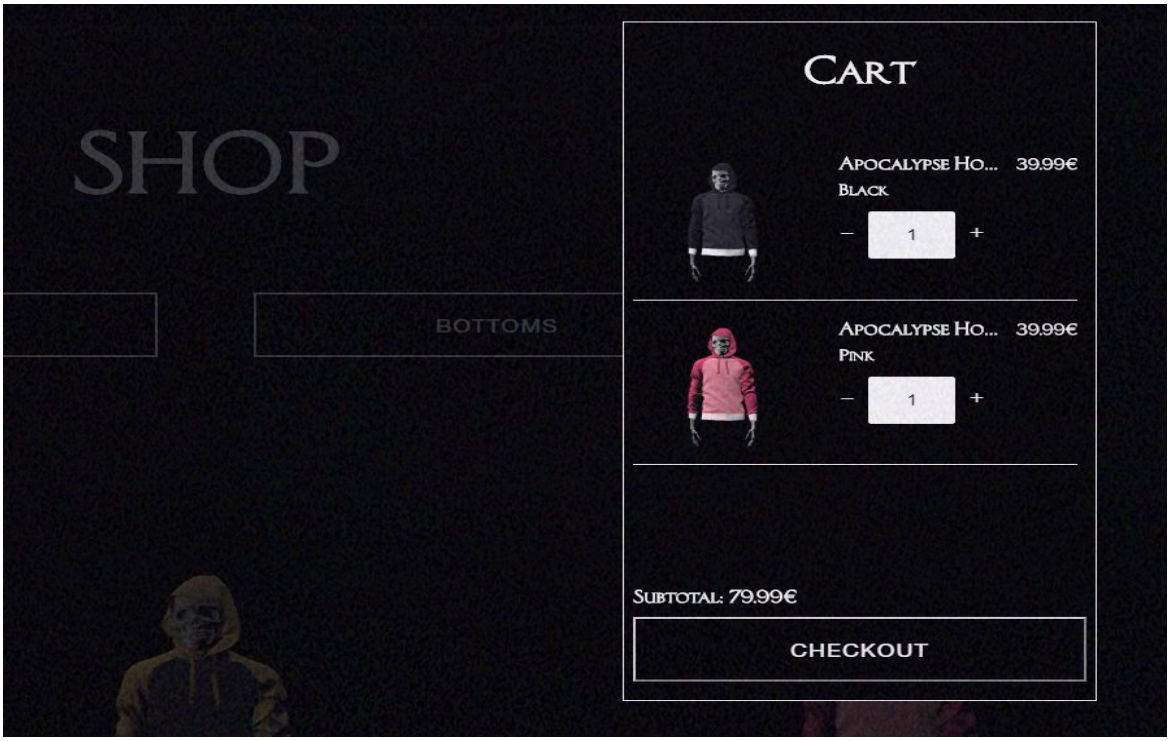
The team decided that some of the components in the frontend aspect of our application should be created separately, so as to decouple them from each other. Which would then give us the ability to use them extensively wherever we needed them, the biggest example of that decision is our navigation bar component, which had to be used in the biggest portion of our web application, excluding views such as the admin one.

And so we created the navigation bar component, and reused it like so in any component that had a need for it, which saved us a lot of code recycling.

```
<!-- Navigation Bar-->
<app-navigation-bar></app-navigation-bar>
<!-- Product -->
<div *ngIf="productModel && currentProduct">
<main class="page-product" >
  <div class="product-wrapper">
    <div class="product">
```

Cart Data

Another frontend decision that had to be argued for, was figuring out the correct way to save the data added to the cart by the customer inside the local storage, so that the information would still be there if the user left the website, so as to improve the user experience.



What we ended up deciding upon was creating a new model called ProductCart, which would belong only in the frontend module, and includes all the information shown for the product inside the cart.

```
import { ProductStock } from './product/productStock.model';

export class ProductCart {
  name: string;
  price: number;
  color: string;
  size: string;
  imagePath: string;
  quantity: number;
  productStock: ProductStock;
}
```

That way we could easily handle the cart's data in the local storage using a newly made service called cart service, which had the responsibility of managing the locally stored cart data, such as adding a new cart product to the storage.

```
addProductCart(productCart: ProductCart){
  if (this.getProductCarts() != null){
    this.productCarts = this.getProductCarts();
  }
  this.productCarts.push(productCart);
  localStorage.setItem('productCarts', JSON.stringify(this.productCarts));
}
```

And whenever we had to load that data to our cart again, we could easily call the getAll function of the cart service to receive the locally stored cart data, and display it to the user, like so.

```
addProductToCart(productCart: ProductCart){
  this.cartService.addProductCart(productCart);
  this.productCarts = this.cartService.getProductCarts();
}
```

An interesting thing to note, is the fact that every time a user is adding a product to the cart, which is done from the product details component, we would have to pass that product to the cart through a function, which belongs in the navigation bar component. To solve that we made the navigation bar component a child of the product details. The reason for that being the simplicity and ease of calling methods from another component. Even though there may have been a better way to accomplish this as far as design goes, we were unable to, mainly because of our lack of experience in Angular, since it was fairly recently introduced to us.

```
@ViewChild(NavigationBarComponent, {static: false}) cart:NavigationBarComponent;
products: Product[] = [];
currentProduct: Product;
currentProductStockIndex: number;
productModel: ProductModel;
```

Product Filtering

Part of the shop page of our web application is the ability to view all of the listed products currently in stock, along with important information about the particular clothing. But so much information could be overwhelming to a customer, especially when it's all listed in one location with no discernable order, for that reason we have implemented a filtering functionality to our shop view.

There are two aspects to the filtering in our application, the first one comes in the form of paging, in our case every page displays a total of six products, and can be navigated using the pagination at the bottom. As for the second part, we give the ability to the users to filter the products depending on the category that they desire to view, right now these categories are top and bottom, but are very easily expanded upon.

In the backend side of the system we had to decide the way in which we should handle both filtering methods. The initial step was to create a FilteredList class, which encapsulates any kind of specific filtering model we have made because we decide to make it a generic type.

```
public class FilteredList<T>
{
    public int CurrentPage { get; set; }
    public int ItemsPerPage { get; set; }
    public int TotalPages { get; set; }
    public IEnumerable<T> Data { get; set; }
}
```

As of now we only had one filter model, since that was the only one necessary at the time, but because of this design it means we can easily add filtering for different entities without much of a headache.

We then made that said filter model called ProductModelFilter, for use in our shop list view.

```
public enum CategoryType { Default, Top, Bottom };

public class ProductModelFilter
{
    public int CurrentPage { get; set; }
    public int ItemsPerPage { get; set; }
    public CategoryType CategoryType { get; set; }
}
```

Once this initial setup was done, we had to figure out the way in which we would filter the product models in the repository, and decided upon this design. At first we check if the current page and items per page given equal to zero, in which case we return back the full list of product models, without any sort of filtering.

```
FilteredList<ProductModel> filteredList = new FilteredList<ProductModel>();
if (filter.CurrentPage == 0 && filter.ItemsPerPage == 0)
{
    filteredList.Data = _ctx.ProductModels.AsNoTracking();
    return filteredList;
}
```

We then decided to make a new enum value for CategoryType called Default, since enums cannot be null we had to create a default value for our enum to check whether the client wants to filter by category or not. The reason as to why we used enums is because they are error proof and very flexible. Otherwise, if the client asks for a specific category to be viewed, we filter with the specified enum and return the given result.

```
if (filter.CategoryType == CategoryType.Default)
{
    filteredList.Data = _ctx.ProductModels.AsNoTracking()
        .Skip((filter.CurrentPage - 1) * filter.ItemsPerPage).Take(filter.ItemsPerPage);

    if (_ctx.ProductModels.Count() % filter.ItemsPerPage != 0)
    {
        filteredList.TotalPages = (_ctx.ProductModels.Count() / filter.ItemsPerPage) + 1;
    }
    else
    {
        filteredList.TotalPages = _ctx.ProductModels.Count() / filter.ItemsPerPage;
    }
}
else
{
    filteredList.Data = _ctx.ProductModels.AsNoTracking().Where(pm => pm.ProductCategory.Name.Equals(filter.CategoryType.ToString()))
        .Skip((filter.CurrentPage - 1) * filter.ItemsPerPage).Take(filter.ItemsPerPage);

    int totalFilteredProductModels = _ctx.ProductModels.Where(pm => pm.ProductCategory.Name.Equals(filter.CategoryType.ToString())).Count();
    if (totalFilteredProductModels % filter.ItemsPerPage != 0)
    {
        filteredList.TotalPages = (totalFilteredProductModels / filter.ItemsPerPage) + 1;
    }
    else
    {
        filteredList.TotalPages = totalFilteredProductModels / filter.ItemsPerPage;
    }
}
```

As seen in the picture above, we also get the total number of pages from the backend, instead of doing that function in the frontend, the reasoning behind this is that whenever you get the product models by navigating using the pagination, you are invoking this filter method, so there is actually no extra call made to the backend for this to work and no overhead in performance.

2.5 Security

In order to secure our system and make sure that our data and software hasn't been exposed to attacks from malicious users, or simply by people who are unaware of their actions, we had to ensure a lot of security and safety checks, as well as restrictions to disallow unwanted use of the system's important data.

2.5.1 Authentication

Firstly in order to stop users who do not have the authority, from being able to reach views/components which have administrative purposes, thus allowing untrusted users to mishandle and modify vulnerable and important information that belongs in our web application, we implemented an authentication system both in our backend and in our frontend.

To achieve this we implement an authorisation guard which redirects users to a login page in case they are trying to reach into authorized only views.

```
const routes: Routes = [
  { path: '', redirectTo: '/', pathMatch: 'full' },
  { path: '', component: HomeComponent},
  { path: 'admin/login', component: AdminLoginComponent},
  { path: 'admin/products', component: AdminProductsComponent, canActivate: [AuthGuard]},
  { path: 'admin/metrics', component: AdminMetricsComponent, canActivate: [AuthGuard]},
  { path: 'admin/categories', component: AdminCategoriesComponent, canActivate: [AuthGuard]},
  { path: 'admin/orders', component: AdminOrdersComponent, canActivate: [AuthGuard]},
  { path: 'admin/countries', component: AdminCountriesComponent, canActivate: [AuthGuard]},
  { path: 'vision', component: VisionComponent},
  { path: 'shop', component: ProductListComponent},
  { path: 'shop/:category', component: ProductListComponent},
  { path: 'shop/product/:id', component: ProductDetailsComponent},
  { path: 'checkout', component: CheckoutComponent},
  { path: 'credits', component: CreditsComponent},
];
```

Once the user has inserted their credentials, we send a POST request to our RestAPI, through an authentication service, which returns a response if a user with such credentials is found, we retrieve a token from

the response and store it into the local storage for further use.

```
login(username: string, password: string): Observable<boolean> {
  return this.http.post<any>(environment.apiUrl + '/api/users',{ username, password })
    .pipe(map(response => {
      const token = response.token;
      if (token) {
        localStorage.setItem('currentUser', JSON.stringify({ username: username,  token: token }));
        return true;
      }
      return false;
  }));
}
```

Now that the token is stored in the local storage, and the user has administrative privileges, anytime they call to the database through administrator only functions, we will include the token to verify that the user is authenticated. This is done by passing a header to the request containing the token which is received from the local storage.

As for the backend there are two different entities responsible for the user, one is to read the login input from the user, and the other is to store the users with their hashed password. This is done to prevent the possibility of someone injecting the query and easily accessing the data.

To accomplish this we generate a random salt based on the password which is then used to encrypt the password, thus securing it from rainbow table attacks.

```
public bool VerifyPasswordHash(string password, byte[] storedHash, byte[] storedSalt)
{
    using (var hmac = new System.Security.Cryptography.HMACSHA512(storedSalt))
    {
        var computedHash = hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
        for (int i = 0; i < computedHash.Length; i++)
        {
            if (computedHash[i] != storedHash[i])
            {
                return false;
            }
        }
    }
    return true;
}
```

Whenever a login is executed, it will take the hashed password saved in the database along with the salt that was used as part of the hashed password, then the password given by the client is compared to the hashed password byte by byte, this time using the saved salt.

In the case that these two passwords match, a trigger is sent and a JWT(Json Web Token) token is generated which contains the claims, meaning the information asserted about the user.

This concludes the general principles and design for authentication created around our application.

2.5.2 CORS

To further increase the security of our web program, we had to handle the CORS(cross-origin resource sharing), so that we can allow only the origin, and specific websites which we trust to possibly load us some necessary resources in the future, while also avoiding and disallowing anybody from being able to send unwanted HTTP requests, which could potentially mess with important data, by either deleting it or modifying it, or using it for malicious use.

In order to enable CORS while also securing our system from probable external misuse, we had to add a specific policy for the CORS to follow, in which we specify which origin is able to send HTTP requests.

```
services.AddCors(options =>
{
    options.AddPolicy("AllowSpecificOrigin",
        builder => builder
            .WithOrigins("http://localhost:4200").AllowAnyHeader().AllowAnyMethod()
        );
});
```

We then just had to make sure to configure our startup class to use CORS with the specified policy we have just created.

```
app.UseCors("AllowSpecificOrigin");
```

This was everything needed to be done, in order to handle CORS in a secure and safe way for our project.

2.6 Infrastructure

Storing data safely and securely in big masses has been of big importance as the years pass, and it is expected from a lot of companies. Because of the increased usage of data that occurs daily there are a lot of expenses that have to be covered.

In our case our client, RUST is a small startup company which does not have the comfort to such big expenses, which is one of the major reasons as to why we build our application using cloud computing, specifically Microsoft's Azure cloud service.

Cloud computing offers a vast amount of advantages, which could benefit a lot of companies and businesses. Mainly, the amount of time and resources that have to be invested in building up an application using cloud services is cut by a significant amount. The fact that you do not have to take into consideration the investments for hardware and software is enticing, and is one of the cheapest ways to have on-demand data storage. Especially in our case being students developing software for a startup company.

Furthermore future expansions of projects and systems can be hugely expensive and time consuming, quite a lot of times companies have to make big investments towards hardware and resources needed, sometimes beyond the resources and computing power necessary. This is where using cloud computing has a humongous advantage, cloud computing offers great scalability. The customer is able to adjust the service to their needs depending on the bandwidth storage, as well as the computing power required.

Additionally another capability of cloud services is being able to launch at a moment's notice throughout any region in the world, this is very advantageous for a small fashion company such as RUST, which has to influence people who belong in many different types of cultures. In order to acquire their attention and sell their brand. Some more advantages include the ability to backup and recover your data, running across different operating systems, the security of your data and much more.

There are of course some disadvantages to using a cloud service, mainly the fact that you do not own the cloud, you do not fully have control over your data, and if something happens to the service, you do not possess any power to prevent any unwanted outcome. That being said, the good outweighs the bad in our situation, and the usage of cloud computing is clear for a project such as this, providing a myriad of benefits towards the goal of our project.

2.7 SDP-CDS Conclusion

Overall using all of these techniques and design patterns specifically for the development of the software for our client, has resulted in a software product which is reliable and secure towards any user, but also maintainable by providing a sense of adaptability and flexibility. Since our system's architecture gives us the ability to further improve and add features in the application with ease, thusly handing over a quality product to our client which is able to meet their ambitions on the technical level.

To achieve this we have utilized a lot of the things we learned during this third semester, such as token based authentication, cloud infrastructure and system distribution with the use of Restful APIs, displaying in the process our knowledge of those said practises.

3. SDM

3.1 SDM Introduction

Before starting off our project we had to ensure the approach and development methodology that we were most comfortable with, and the one that would help us improve our software quality, as well as our communication with the client, and the ability to respond quickly to sudden changes in our requirements list.

With that in mind we decided to stick mostly towards the XP methodology and practises that we learned in the duration of this semester, such as pair programming, unit testing, code simplicity/clarity and more. Reason being that we had to quickly adapt to changes, and also we required a methodology that was designed purely for software development. We did however exclude and modify a few of the practises which did not fit our project's environment, and ignored the Scrum methodology completely, the rationality behind that decision, being that our project had a very small life cycle until production, and only had one production launch. Instead of many iteration loops showcased to the client over the course of weeks. On top of that, the fact that we did not want to spend a lot of time on documenting our process, and preferred to work to improve the quality of the software itself, led to that exact decision.

It is also important to keep in mind that the end result will be continued to be put in practice, once the project end date has been reached. Some team members who are involved in the project might not continue working on the project or a completely new team will take over this project, to refine the final result. Because of this reason it is important to establish a quality management strategy for other teams to take over and let the project become a successful long term project.

3.2 Project Foundation

In order to have a clear heading of our project and goal, we wrote a project foundation document, so that we could define the strategy of our development, and create a clear understanding and communication between the development team, and the client.

3.2.1 Resources and Conditions

One of the crucial steps to establish our strategy towards the project was to identify the resources that we had at hand, so that we would know how to use them to the best of our advantage, and also the conditions and environment in which we had to develop our application, in order to avoid potential mistakes, as well as follow the correct guidelines.

3.2.1.1 External conditions

The external conditions define the base requirements of our goal, but also the external forces which we cannot alter, the details of which can be seen at **Appendix C**.

3.2.1.2 Person Resources

Another aspect of our project's foundation were the people involved in and around it, so we made a board(**Table 1**) representing the roles of each individual person and their responsibilities towards the succession of the project

Table 1. Roles and Responsibilities

Role	Responsibilities
Product Owner Michał Parużyński	• Creates a product vision and conveys it to the developers. • Represents the initial and ongoing requirements for the product to the team. • Ensures close collaboration with the development team.
Supervisor Lars Juul Bilde Bent Pedersen Henrik Kühl	• Organizes the workflow and ensure that the developers understand their goal. • Monitor the developers productivity and provide useful feedback. • Sets up goals and deadlines.
Developer David Kalatzis Greg Charyszczak Kevin Luu	• Estimate the effort required to accomplish a task. • Identify any impediments to the progress of the project. • Achieves the goals that are defined in a set window of time. • Attends to daily meetings and reports their current progress.
3D Artist Marcin Dwornikowski	• Prepares work to be accomplished by gathering information and materials. • Plans visual concepts and showcases them to the team. • Produces art assets to meet the product's vision.

3.2.2 Strategic Analysis

In order to formulate a strategy and achieve the company's goal we conducted research on the company, and it's operating environment which resulted in the following strategic analysis.

3.2.2.1 Stakeholders

First of all it was important to clearly define a stakeholder analysis, it is the people involved first and foremost who shape and define the project, and it is also people who may cause problems and issues towards the progression of the task at hand.
With that information in mind, we identified the following stakeholders surrounding the project:

1. Internal: Developers, Designers
2. External: Product owner, Supervisors, Customers

A Power-vs-Interest matrix(**Figure 1**) was also drawn to help us visualize the importance of each stakeholder.

Figure 1. Power-vs-Interest Stakeholder Matrix

As a result the stakeholders that classified as of out most importance are the customers, since the whole website is based around their satisfaction and reception. Knowing that our main priority was to ensure a pleasing and simple user experience that would stand out to help establish the brand's name.

Having that in mind, we knew that fully publishing the website for real customers may have not being possible, which made the product owner another really important stakeholder to focus on.

3.2.2.2 Strengths and Weaknesses Analysis

Afterwards the team constructed and brainstormed to build a list of ideas to identify where we stand, both in our strengths and our weaknesses.

Finally we made a simple strengths and weaknesses board(**Table 2**) in order to showcase this.

Table 2. Strengths and Weaknesses

Strengths	Weaknesses
• We are able to communicate between us very quickly and efficiently, especially given the fact that the team is so small. • We can easily schedule up project meetings, as all team members live near the Business Academy. • We can rapidly test out ideas because of our architecture. • We have an easy connection to the product owner, which allows for quick feedback.	• We had a slow start because the team was busy and unable to fully focus on the project. • To ensure that frequent team meetings could happen, each team member had to reschedule their own calendar. • We overestimated how many things we could do in the given time, thus resulting in the team being pressed during the last days.

3.2.2.3 Critical Conditions

We have also pondered about the critical conditions**(Table 3)** of our project, which are entirely based upon the client's needs, and came up with the following.

Table 3. Critical Conditions and Precautions

Critical Conditions	Precautions
• Limited time. • Loss of progress. • Users need to have secure data.	• Prioritizing and planning ahead • Using github to guarantee a version controlled development environment, other important documents should be backed up. • Implemented business logics should be constantly reviewed.

3.2.2.4 Risk Analysis

We also conducted a risk analysis(**Table 4**) in an effort to predict future issues and figure out the best way to manage them, and how we could mitigate them, as well as sort out which problems would cost us the most.

Table 4. Risk Analysis

Risk	Risk Management
Human:	
Illness	We accept the risk and manage it - If a team member is feeling sick they have to notify the group, as long as the illness is not severe they can still work from their home, otherwise the rest of the team will have to replace their work.
Injury	We accept the risk and manage it - In case of the injury is not severe, the team member can still work from their home, otherwise if it is serious the team will have to do the work for the injured team member until they recover.

Team communication	We accept the risk and manage it - In the event of team miscommunication or disagreement, the whole team will have a talk and together decide on the best course of action.
Technical	
Loss of project data	We accept the risk and try to avoid it - To avoid this issue we use version control systems like Git and GitHub.
Azure issues	We accept the risk and try to avoid it - There could be problems where we lose our student license, or our Azure database, to deal with this risk we setup our cloud service together, and keep informed on the status of our Azure account.
Reputational	
Loss of client	We accept the risk and try to avoid it - In order to avoid this we try to keep a clear communication between us and the client, as well as provide the best quality product that we can produce.
Project	
Slow development	We accept the risk and manage it - We prevent such an issue from occurring by planning ahead, and knowing beforehand the goals that we wish to achieve and the ability of each team member.
Code quality	We accept the risk and manage it - To avoid this we defined our own code standards by following the general standards, but also what the team agrees on. Furthermore we utilized the code patterns and architectures that we learned throughout our curriculum.
QA quality	We accept the risk and manage it - We manage this risk by consulting with our product owner for constructive feedback, in the case that they are not available we get help from teachers and other students.
Politics	
Change to government laws	We accept the risk but have no control over it - Even though it isn't likely for something like this to occur, we keep ourselves informed in case it could impact our project.

Green: low risk, Yellow: intermediate risk, Red: high risk.

3.2.2.5 The Strategy for the Project

In summary, we have identified the strengths and weaknesses during the project. It is our goal to eliminate our weaknesses in order to seek out possible opportunities, which can benefit our project, it is also important to seek out the opportunities, which complement our strengths. To follow this strategy, it is vital to keep in mind that there are many risks inside and outside of the company, which could cause the project to fail. The most common risks stated above have been identified, and we have classified each of the risks either as low, medium or high, based on the damage the risk could cause to the project and the likelihood for the risk to occur. In addition, we have also identified the knockout criterias for the project, which could instantly cause the project itself to fail. Based on the analyses, proper solutions and special precautions on how to deal or avoid each risk individually have been designed.

3.2.3 Project Model

Having a clearly defined project model is needed in order for us to work on the project in a structured and efficient manner. We defined and agreed on the following five phases which will cover the entire lifecycle of the project.

- **Project identification and design:** The team works with the product owner to identify the feature and design of our project. Requirements are being defined, and the acquired information act as a guideline and general direction throughout the project.
- **Project setup:** Further definition of the project features and design are done and documented, after which the accountability for each feature in the project being defined.
- **Project planning:** Based on the developed documents, which have been created in the recent phase, a detailed and comprehensive implementation plan is being made.
- **Project implementation:** The implementation plan is put into practice in this phase. By following the agile method, features are constantly being revised until the product owner is satisfied.
- **End of project transition:** The requirements of the product owner have been satisfied. All the functionalities in order to manage the resulted product are available, and the product owner is schooled on how to manage the product himself.

3.2.4 Procedures and Internal Organization

As a team we had to come to some agreements over the accepted procedures during the project, for instance the meeting time, working hours, actions taken in case of absence etc., and so we agreed on the following.

- **Management Meetings:** Michal, the product owner and Greg, one of the developers agreed on having weekly meetings on either Saturday or Sunday, either via a Facebook-call or via any other tool which has call-functionality, to evaluate the progress which has been made during the week. Based on the evaluation, current features need to be either revised or additional features be implemented. The acquired information during the management meeting is being discussed in the following week during the project meetings.
- **Project Meetings:** The working hours that were agreed upon was from 9 am - 3:30 pm, the team would meet on a daily basis with the exception of weekends, and have a meeting for about 10 minutes discussing on what each person has done.
- **User Meetings:** The meetings for the users were not scheduled, but rather something that we would do whenever we had a finished behaviour in our program which sorely needed user feedback to adjust accordingly, that feedback usually came from other students and/or teachers.

The schedule can be found in **Appendix B.**

- **Work Sharing Principles:** It was decided upon, that the work done by each teammate would be shared to a version control system such as git, for other members to see and access, but also evaluate if it meets the standards designed by the team.
- **Configuration Management:** We also had a general configuration on the work setup, every team member was allowed to take a break whenever needed, with two scheduled breaks, the first being from 11:00am - 11:15am and the second from 1:30pm -1:45pm. Other than that the general behaviour that was agreed upon was that everyone should stay positive and open minded in case of arguments, and in moments when someone from the team couldn't figure out a problem they shouldn't be worried to ask for help.

The working agreement can also be seen in further detail in **Appendix A.**

3.2.5 Overall Project Plan

Estimation techniques:

In order to schedule and manage our project we used some estimation techniques.

The very first thing that had to be done in order to correctly estimate the project is the understanding of the project. We first came to a conclusion as to what the project is about, its advantages and disadvantages, this helped us form a general idea on how to plan a schedule around it.

We then did a what-if analysis, we broke down our scope into smaller tasks, and arranged them in the order in which they had to be done, as well as identify the effort and resources that had to be used to complete them, but also considered the risks that could be involved.

We also used a top-down technique, having other similar projects as a reference, we started off with the final goal of the project and broke it down into smaller parts which would be further analyzed and assigned to team members.

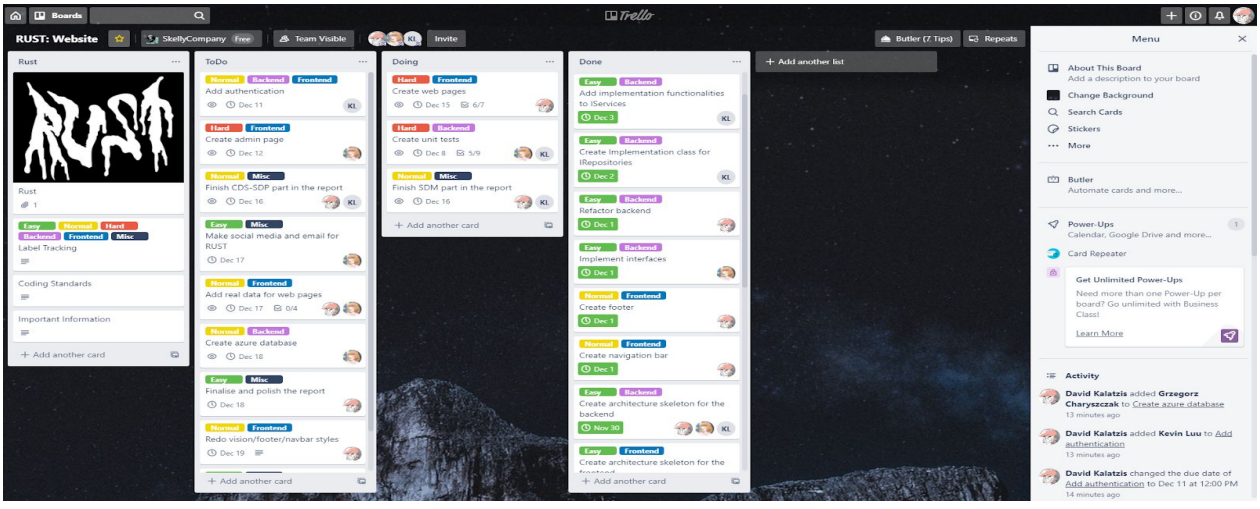
Project Plan:

The team kept on a very simple and clean project plan, because of the allotted amount of time that we had we couldn't put too much of our attention and energy towards documentation, and our communication with the client was very clear which helped a lot in that regard.

With that in mind in order for us to manage our approach and the process for developing this program, we have used Trello. In order to compensate for the simplicity of our project plan we kept our board very informative and would constantly update it throughout the day, so that everyone could be up to track.

Below we have a snippet(**Figure 2**) of our project plan using trello boards.

Figure 2. Project Plan Board



We did however also write a table detailing the overall project plan, as part of the report, and it can be seen at **Appendix E**.

3.3 Reflections on the original XP Practices

We went through all of the practises that are implemented by the XP methodology, so that we could have a better understanding and overview of XP, but also realize which XP practises could be used for our project, and which should be removed or modified.

- **Planning Game:**
  We decided to use this practise, so that we could have a better understanding between developer and client, and be fully aware of the client's requirements and their priorities.
- **Small Releases:**
  Because of the nature of this project we could not utilize this practise, since we only have one release day instead of multiple.
- **System Metaphor:**
  We benefit by using this practise, due to the fact that it allowed each item to be easily and intuitively understandable.
- **Simple Design:**
  This was an important practise to follow, but it wasn't always possible, since the project had to be expandable even after our release day.
- **Continuous Testing:**
  The reason we included this practise is so that we could validate if our new features were correctly implemented, and it helped us solidify our definition of done.
- **Refactoring:**
  This was another XP practise which we decided to follow, the reason being that it helped clear our code and make it easier to modify or change in the future.
- **Pair Programming:**
  We also used this practise, so that we could help each other in critical moments, it also was important for making tests, since having two people helped us realize problems that the other person would've not thought about on their own.
- **Collective Code Ownership:**
  Another practise that we used, solely for the fact that we are a team of only three people, so it was very easy to allow everyone to work on any part of the system, couple that with other xp practises like coding standards and continuous integration, there wasn't any big risks in following this practise.

- **Continuous Integration:**
  Using this practise has helped all team members keep on the same page, by keeping our application up to date and working.
- **40-Hour Week:**
  Considering the rapid development of XP it was clear that we had to adopt this practise, because we had to keep our team motivated and well rested.
- **On-site Customer:**
  This practise was not used, because it was not possible for us to have access to a customer so often, so we would rely on other students, and our communication to the client to get some feedback on our development.
- **Coding Standards:**
  At the very start we decided on coding standards for the team to follow, this allowed us to share the code between us easier, and lowered the learning curve.

3.4 Process Documentation

By using the XP methodology as our main strategy for developing the software we were able to keep a high quality for our product, and produce it in a quick way, with the help of the practises designed around the XP methodology. We did however have to alter them and document their way of usage in our application.

Below we have listed all of the XP practises we have utilized and the way we used them in the process of this project.

Planning Game:
We cooperated with the client to produce the maximum business value. First of all the client wrote down a list of features to be implemented, which we then gave a rough estimate on how long it will take to implement, finally the client made a list of which to produce and in what order.

Using this practise helped us understand the requirements and the priority that was desired by our client, and also establish our communication with the business.

Simple Design:
We always tried to stick with the simplest solution to a problem that we could come up with.

This resulted in us being easily able to adapt to any cases where the requirements for the projected had to change during development.

Continuous Testing:
Before adding any new feature, we would first write a test for it, once the test had passed we knew that our job was done.

This helped us to sustain consistency in the quality of our code, and the security and performance of our program as a whole.

Refactoring:
In cases where duplication appeared, we would always refactor our code without any fear of our application breaking, because of our test classes.

By implementing this practise we were able to keep our code much cleaner and easier to read in case of murder modifcation.

Pair Programming:
Whenever there was a difficult and complex problem, two people would sit at once machine and try to solve the issue.

Pairing up together helped a lot in crucial moments and resulted in faster development of the application.

Continuous Integration:
All of our code is integrated to our version control on a daily basis, and with the use of continuous integration services like Travis CI we were able to test before any major change.

This resulted in a rapid expansion of our project which also has proven value to the client, since they were fully working versions

40-Hour Work Week:
In order to sustain the quality of our code in such a rapid working environment we had to ensure that everyone was well rested, which is why we decide to follow this principle.

Giving everyone in the team a breather resulted in everyone of us having more motivation and energy to work on the project.

Coding Standards:
We established early on the code standards of our application for everybody to follow.

This means that the code is cohesive, and it is indiscernible which member wrote which piece of code.

3.5 Configuration Management

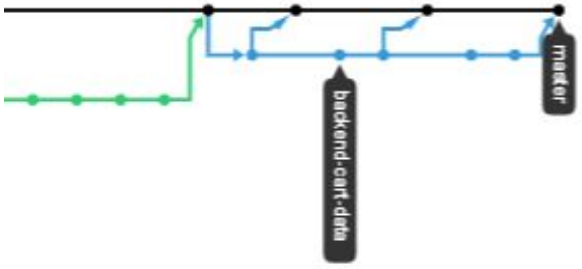
3.5.1 Branching Model

We decided at the very start to keep on with a simple branching model, meaning we only had the master branch as our default branch, and feature branches for any newly added features.

The reasoning behind this was because of the environment in which our application was being developed under, meaning we only had one release, and no further updates, so we saw no reason to add a development branch to further the complexity with no real advantage gained from it, and with the very small size of our team it was easy to keep up with what was happening without any further branches.

Below we are showcasing an example of how our branches work in a network graph.**(Figure 3)**

Figure 3. Github Network Graph

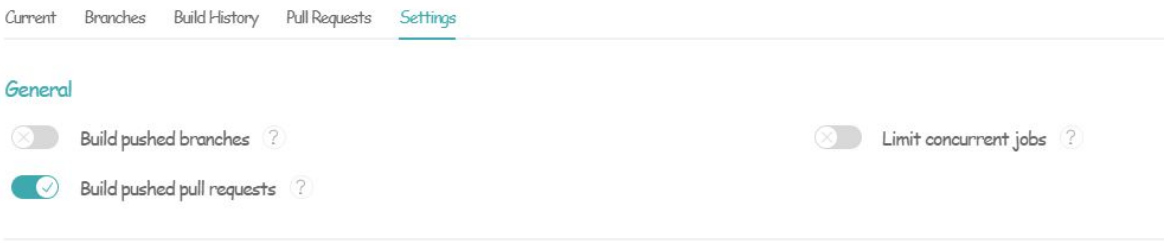


3.5.2 Continuous Integration

Travis CI was the chosen continuous integration system over TeamCity, being that Travis is much simpler and easier to set up for such a quick project, and the cloud based nature of it made it effortless for each member to access the CI.

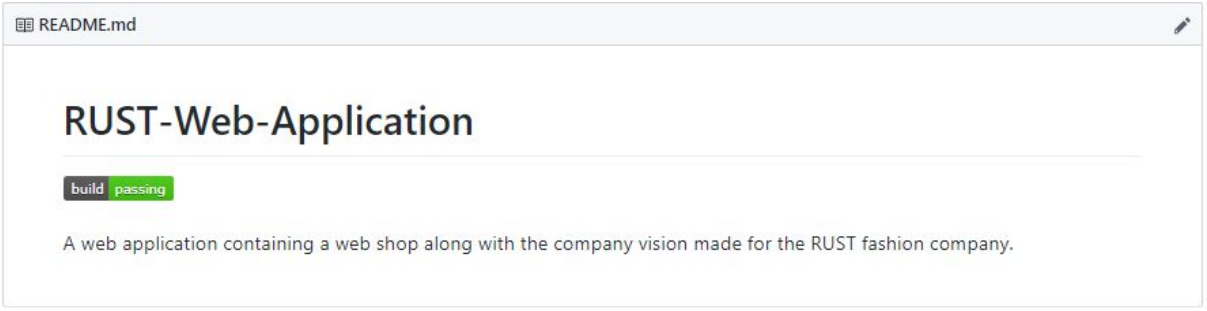
On top of that we made unit tests for every single business entity, and allowed Travis to build only during pushed pull requests(**Figure 4)**, since every new change and feature should be only done through other branches, thus encouraging this behaviour.

Figure 4. Travis Setup



We also utilized travis for other smaller things such as showcasing in our version control(Github) the current status of our master branch**(Figure 5)** to anybody viewing the repository

Figure 5. Github Travis CI Status



3.6 Test Driven Development

The team adopted the philosophy of test driven development throughout the whole development of this project, we constructed a unit test for every single business entity that we had created and made thorough tests, following and using the standards that we had set.

We would first start by creating one method of the unit test, and running the tests in hopes that they would fail, afterwards we would write the implementations for those tests in order to make them pass, and finally we would go back to the tests to polish and refactor them to fit our quality standards. This whole process was used for every single unit test and it results in us having full confidence that our project is secure and that our tests are running correctly. By doing test driven development we also ensured that only the necessary code was implemented.

In combination with the CI, whenever we had to pull request, Travis would build our program and check if the tests were still running successfully, which helped a lot to keep on track on when and where a problem occurred and gave bigger flexibility for every team member.

In this example we are unit testing our product entity, specifically the create method, and expect to get back the created product with an id. We first arrange the test by initializing two product objects, one acts as the actual product being created, and the second one acting as what we are expecting to receive, afterwards we setup the services and repositories which are required to perform the test, and setup using the Moq library the said repositories, then in the act phase of our unit test we call the create method with the newly created product, and finally check if the actual result we have equals with what we expected to get.

```
[Fact]
public void Create_ProductValid_ReturnsCreatedProductWithId()
{
    //Arrange
    Product validProduct = new Product
    {
        ProductModel = new ProductModel { Id = 2},
        ProductStocks = null,
        Color = "Black"
    };
    Product expected = new Product
    {
        Id = 1,
        ProductModel = new ProductModel { Id = 2 },
        ProductStocks = null,
        Color = "Black"
    };

    Mock<IProductRepository> productRepository = new Mock<IProductRepository>();
    productRepository.Setup(repo => repo.Create(validProduct)).
        Returns(expected);
    Mock<IProductModelRepository> productModelRepository = new Mock<IProductModelRepository>();
    productModelRepository.Setup(repo => repo.Read(validProduct.ProductModel.Id)).
        Returns(validProduct.ProductModel);

    IProductService productService = new ProductService(productRepository.Object,
        productModelRepository.Object);

    //Act
    Product actual = productService.Create(validProduct);

    //Assert
    Assert.Equal(expected, actual);
}
```

3.7 SDM Conclusion

In conclusion learning and using the XP methodology has helped through a lot, and it suited our project really well since we had such a short deadline with a very small group. What helped us the most was the fact that there were really specific practises aimed for software development that laid the path on how we should process our way through the project.

We also learned a lot from some mistakes we made, and realized the strengths and weaknesses of XP, such as the fact that the continuous short releases of XP didn't really work in our case since we only had one and final release at the end of December, and the fact that we always had a need for a customer to give us feedback made things a bit harder since they weren't available as often.

By defining and setting a development framework we hope that future developers who might want to continue working on our project will be able to adopt our project strategy and modify it in order to improve our end product. We implemented the necessary features, where each part of the architecture can be easily replaced, meaning that the project is highly flexible.

4. Bibliography

Extreme Programming: Values, Principles, and Practises.
Link:
https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/

User Authentication with Angular and NetCore.
Link:
https://fullstackmark.com/post/10/user-authentication-with-angular-and-asp-net-core

The principles of UI design.
Link:
http://bokardo.com/principles-of-user-interface-design/

Unit testing best practises using .NET Core.
Link:
https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices

Continuous Integration with Travis.
Link:
https://www.vogella.com/tutorials/TravisCi/article.html

CORS in .NET Core: .NET Core Security.
Link:
https://dzone.com/articles/cors-in-net-core-net-core-security-part-vi

5. Appendix

Appendix A
Working Agreement

Working hours and time scheduling

- Work from 9am till 3:30pm.
- Discuss at the start of the day what we are working on.
- Breaks are allowed whenever necessary.
- Two breaks scheduled one from 11:00am - 11:15am, and second from 1:30pm - 1:45pm.

General Behaviour

- Be positive and and respect each other.
- Do not worry about not knowing something.
- Act open minded whenever there are arguments.
- Be collaborative.
- Be proactive and control a situation before it happens.

Definition of Done

- It has to be fully functional.
- It has to pass all the tests.
- Is it well structured and refactored.
- It follows the code standards set by the team.
- It is compatible with the current version.
- It can be merged with no conflicts.
- It has been approved by at least one more group member.

Appendix B
Overall Schedule

Week 45 (25-29/11/2019)
- Trello board
- Github organization
- Github repository
- Travis CI
- Planning
- Entity Relationship Diagram
- Balsamiq Mockup
- Scheduling
- Product Vision
- Problem Definition
- Product Backlog

Week 46 (2-6/11/2019)
- Testing
- Coding
- Visuals

Week 47 (9-13/11/2019)
- Coding
- Visuals

Week 48 (16-19/11/2019)
- Coding
- Hand in Report

Appendix C
External Conditions

Physical conditions: Developed at school, home

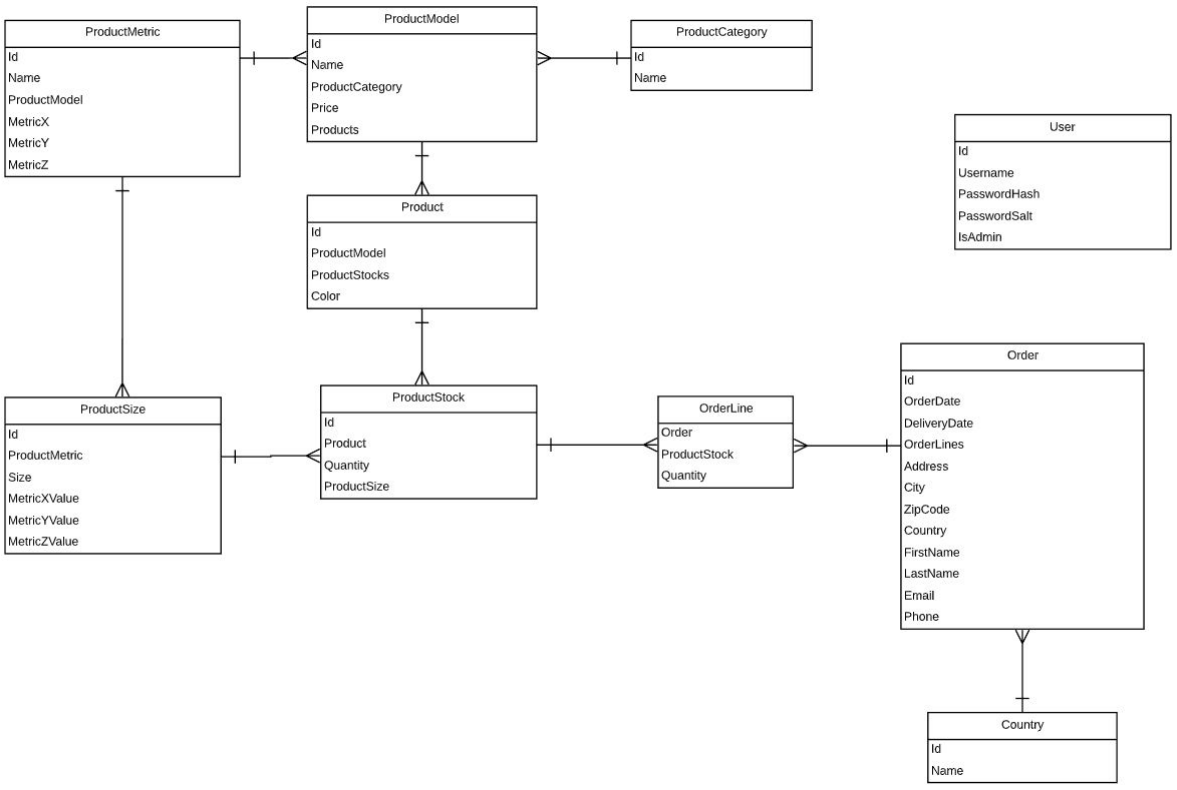
Equipment: Own computer/laptop

Tools:
- Visual Studio or similar IDE using .NET Core 2.2
- Angular 8.x
- Webstorm or similar IDE
- Travis CI
- Microsoft Azure
- Git
- Trello
- Google docs

Standards:
XP Practices (Test Driven Development, Continuous Integration etc.), Clean Architecture

Deadline: 2019 December 19th

Appendix D
Entity Relationship Diagram



Appendix E
Overall Project Plan

Date	**Activity**	**Milestone**
Week 48 (Nov. 25, 2019 - Dec. 1, 2019)		
25/11/2019	Kick off meeting	The team should have a general overview about the project, and what is required from them by the client.
26-27/11/2019	Project Foundation Initial planning Report writing	The project foundation should be completed and laid out to the team in detail. The team should start planning the design and structure of the project. And the report should start it's first steps.
28-29/11/2019	Initial setup Report writing	The initial part of the project should be setup now, such as the GitHub repository and Travis CI, as well as ways of communicating. More writing on the report should be done to keep up with what was made.
Week 49 (Dec. 2, 2019 - Dec. 8, 2019)		
2-4/12/2019	Backend development Initial visualization of the frontend	Things such as the entities and their services should be all setup. The initial planning of how the frontend aspect will look should be finished.
5-7/12/2019	Backend development Report writing	All the unit tests should now be finished and running, and authentication should be added. The report should be updated to keep up with everything that happened during the week.
Week 50 (Dec. 9, 2019 - Dec. 15, 2019)		
9-11/12/2019	Frontend development Cloud setup	The components required for the frontend should now be finished and working. The azure database should be ready for deployment.
12-13/12/2019	Frontend development Report writing	The admin page should now be finished and the frontend should be connected with the Rest API. More report writing to update on the changes.

Week 51 (Dec. 16, 2019 - Dec. 22, 2019)		
16-17/12/2019	Refactoring Bug Fixing Report writing	The project should now be finished and be fully functional in the most polished version possible.
18/12/2019	Report writing	Final touches and polishing for the report
19/12/2019	Handing in the solution	The final solution should be handed in through WiseFlow