



UCN, University College of Northern Denmark
IT-Programme
AP Degree in Computer Science
CSC-CSD-S212

Third Semester Project

UCN Web shop – Programming and Technology report

Martin Glogolovský, Petra Pastierová, Matej Rolko, Viktor Tindula
20 December 2022

University College of Northern Denmark

Academy Profession Degree in Computer Science

Class:

CSC-CSD-S212 - Group 1

Title:

Third Semester Project

UCN Web shop – Programming and Technology report

Participants:

Matej Rolko

Viktor Tindula

Martin Glogolovský

Petra Pastierová

Supervisors:

Jens Henrik Vollesen

Gianna Belle

Jakob Farian Krarup

Date of submission:

20 December 2022

Number of normal pages/characters (including white spaces):

17,2 pages/ 41 370 characters

Repository path:

https://github.com/PoustniGaara/Third_Semester_Project

Table of Contents

Introduction	3
Problem area.....	3
Problem statement	3
Methodology.....	5
1. Architecture	6
2. System components	7
2.1. Database.....	7
2.1.1 Relation model.....	7
2.2. Webservice	10
2.2.1 Data Access Layer	11
Security	13
2.2.2 RESTful API.....	14
Autmapper.....	15
Validation filters	16
2.2.3. REST CLIENT	17
2.3. Webpage	19
2.3.1. MVC	19
Authorization	20
2.3.3. Cart	24
2.4. Desktop client app.....	26
2.5. Other system functionalities	27
Error handling	27
Dependency injection	31

Testing.....	32
4. Conclusion	36
5. Group evaluation	37
Appendices.....	39
A. List of references.....	39
B. Group contract.....	41

Introduction

The purpose of this report is to summarize the Programming and Technology section of the Third Semester project in AP Computer Science degree at University College of Northern Denmark. The assignment was for the students to create a distributed software system, with front-end and back-end programming and to demonstrate knowledge of how to solve a concurrency issue. In this report we will explain the steps taken in our third semester project from a Technology and Programming perspective, with adherence to the learning objectives described in the Computer Science National Curriculum (2019).

Problem area

In our third semester project, we document the development process of creating a web-shop for our university, UCN. Currently, UCN makes school merch products in limited amounts that are either given out to students for free on special occasions or sold at various school events. Leftover merch is stored on Campus Hobrovej in Aalborg and at this time, there is no way for students and other interested parties to obtain these products outside of school events. As students, we think there is considerable potential in having a web-shop service where we could purchase merchandise to show off our school pride.

Problem statement

In our problem statement, we have defined a main question for this project:

“How to create a working distributed system using the .NET Framework, while establishing that the system works across the network on all supported platforms and ensuring atomicity and in-real-time synchronization?”

To answer this question, we need to consider additional questions aimed at specific aspects of the system:

“How to solve concurrency issues that occur with collaborative processes when several users interact with the system?”

“How to establish proper system performance, ensuring that all system components communicate with each other correctly?”

“How to create user-friendly UI in the webpage and the dedicated client application?”

Methodology

The system was programmed in the C# programming language, using Microsoft Visual studio IDE, and built on .NET 6 framework. We used ASP.NET Core Web API for the webservice implementation and ASP.NET Core Web App (MVC) for the webpage implementation. Various libraries, such as RestSharp, ADO.NET and AutoMapper were used in the project. The UI for the dedicated client app was done using Windows Forms. For testing, Xunit framework was used. For version control, Github.com. The system data is stored in a Microsoft SQL relation database. The database server for this project was provided by the university and is located at hildur.ucn.dk. To manage the database, Microsoft SQL Server Management Studio was used. The project was carried out using Scrum, and Jira Software for project tracking. All project work was documented in this report, using Microsoft Word. For report progress tracking, Trello was used.

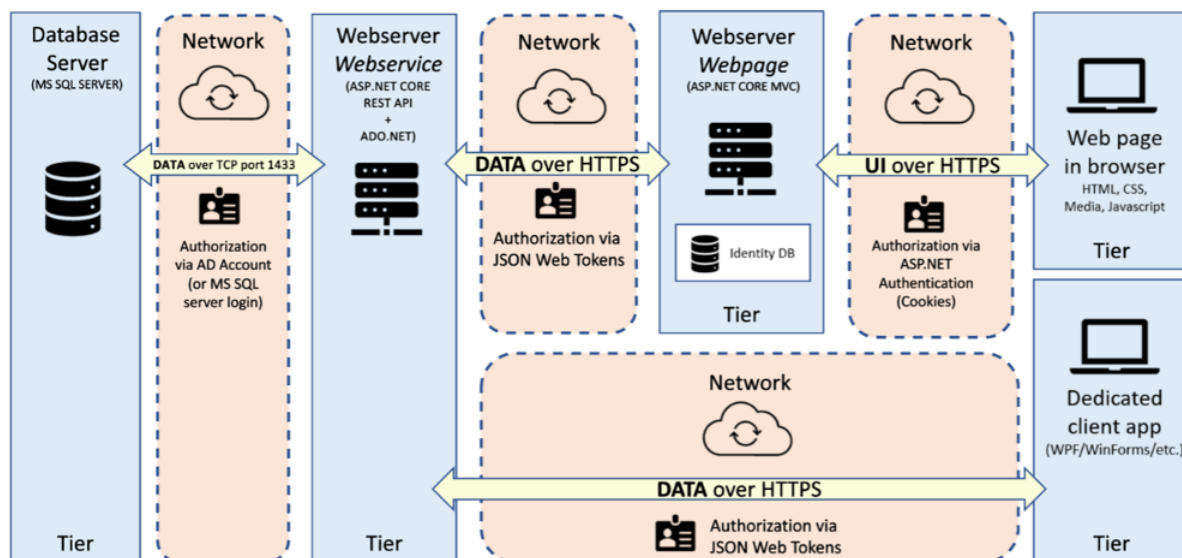
1. Architecture

Architecture is a set of principles which define the software's design and the development process. It is a conceptual model representing the system's functionalities, structure and how it is organized (Coulouris, 2012). In our third semester project, we are using a three-tier client-server architecture, which consists of a presentation tier, logic tier and a data tier (Techopedia, 2021).

Presentation tier constitutes the front-end layer, which displays information on a web browser, or web application in the form of a graphical user interface. It communicates with other tiers by sending action results to the browser and other tiers in the network through API calls. In our project, the presentation tier consists of a web shop in browser and a dedicated client app.

Logic tier, also referred to as middle tier or application tier represents the system's middleware and contains all the business logic. It controls the application's core functionality and performs detailed processing. This happens in the ASP.NET Web API.

Data tier represents the back-end and houses database servers in which the information is stored and retrieved.



There are a number of advantages when working with a three-tier client-server architecture. It is cost efficient in terms of maintenance, meaning specific parts of the application can be

independently updated or replaced without affecting the product as a whole, flexible in terms of upgrades and adding new hardware, such as additional database servers to keep up with data storage. It also makes for a more efficient development process, as it allows for a clear division of work between the project team.

However, it is also important to note that there are downsides to using a client-server architecture, for example if the main server goes down or gets a worm, virus or a Trojan, the users will likely catch it. Furthermore, if too many users try to connect at once, it may cause system overload (Terra J., 2022).

2. System components

In this chapter, we will go into detail about all system components, how they were built and explain the choices behind the different technologies we used.

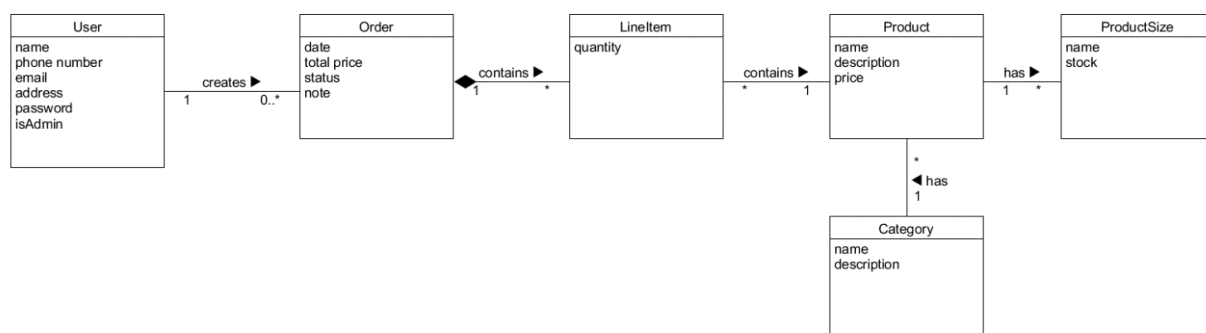
2.1. Database

The relational database is a crucial part of the system, allowing data to be saved and manipulated securely. For this project we are using a relation database located on university server hildur.cun.dk. The database provided for this project is a Microsoft SQL version 15.0.2095.3.

The next chapter describes the process of making a relational database for this project.

2.1.1 Relation model

The relational model represents data storage in a relational database. During the process of creating the relation model, we followed the database design process from the domain model diagram (Elmars, 2011).



The process of mapping the high-level conceptual data model into the record-based logical data model was done by mapping strategies between object and non-object representations. We used our domain model diagram, shown in the figure above, as a representation of the data that needed to be persisted. The conceptual classes from the domain model were mapped into tables in the relational model.

To assure uniqueness of primary keys in the relations, an ID was assigned to every object as the primary key, except for the User relation, where we are using the actual email addresses of users, because we prioritize an already unique attribute instead of creating a new one, that would need an additional column in the relation.

In our domain model, there were no multivalued attributes to be eliminated.

Next, we focused on composite attributes. In the User relation, we split name into (first) name and surname. We are aware that the address is also supposed to be split into additional components, such as street, street number, city, zip code and country, but we decided to keep it as one attribute for simplicity reasons.

One-to-many associations occur multiple times in our domain model. The first occurrence of this association is between User and Order. The User is the one-side, and the Order is the many-side, because users can have multiple orders, thus the primary key of a user, in this case an email address, is added to the Order attributes as a foreign key constraint. This also occurred in associations between Order and LineItem, LineItem and Product, Product and Category, and Product and ProductStock.

Many-to-many and one-to-one associations do not occur in our domain model.

Normalization

In order to ensure quality of our database, it had to be transformed from non-normalized schemas into schemas that follow the normal form, a process known as normalization. Normalization of data can be considered a process of analyzing the given relation schemas based on their functional dependencies and primary keys to achieve the desirable properties of

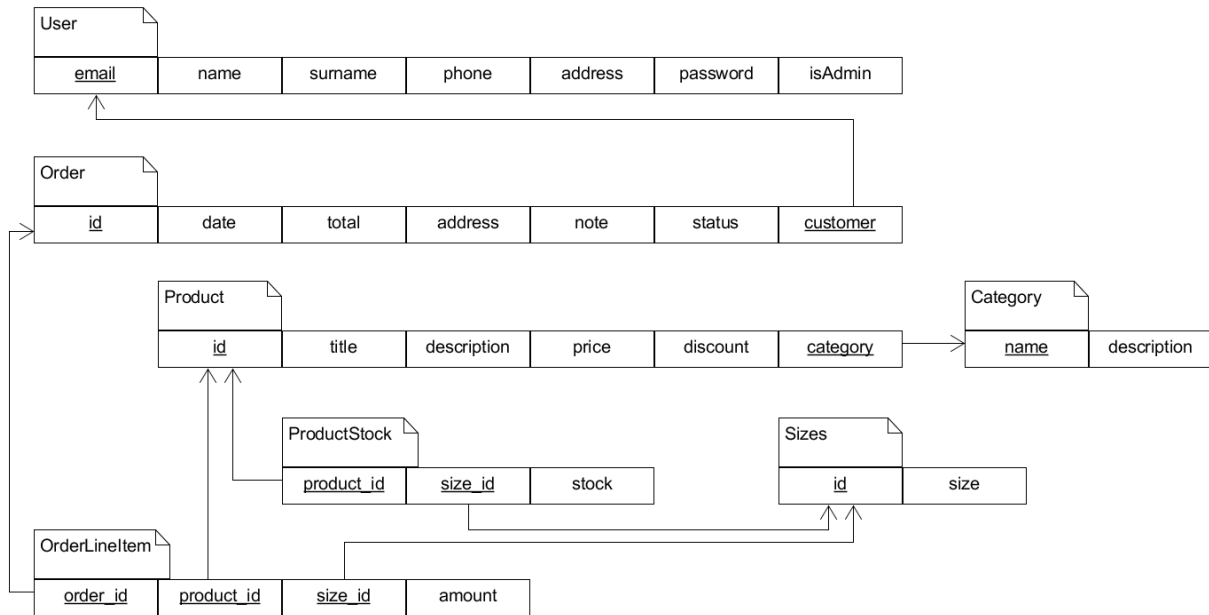
minimizing redundancy and minimizing the insertion, deletion, and update anomalies. (Elmarsri, 2011)

The first normal form (1NF) disallows composite attributes, multivalued attributes, and nested relations. Based on the 1NF, the name of the user, which is a composite attribute, was split into first name and surname. We realize that also the address is a composite attribute, but as stated before, because of time and simplicity reasons, we agreed not to normalize this attribute.

The second normal form (2NF) of a relational model makes all non-primary-key attributes fully functionally dependent on the primary key. In order to get our model to be in this form, we had to decompose ProductSize into two relations – ProductStock and Sizes. This way the ProductStock relation had a composite primary key consisting of an ID of a product and an ID of a size from the Sizes relation. Because the web shop sells mostly clothes, accessories and supplies, the sizes are mostly all the same, ranging from XS to XXL, or in a case of non-clothing products, a One size fits all, shortened as O/S. This way we got rid of data redundancy and made the ProductStock's composite primary key determine stock of the product, while the Sizes relation is a very basic relation with just an id of the size as primary key and then the name of the size.

The OrderLineItem relation also consists of composite primary key made up of three different foreign keys constraints – the id of the order, product and size, on which the amount is dependent.

After the third normal form (3NF) of relation, no non-key attribute was transitively functionally dependent on a candidate key. The 3NF normal form finalized the normalization process of our relation model, which is shown in the figure below.



After the relation model was done, we created SQL scripts to make the database. The figure below shows the SQL script for creating User and Order tables with attributes and constraints.

```

CREATE TABLE [dbo].[User]
(
    [email]    [varchar](50) NOT NULL PRIMARY KEY,
    [name]     [varchar](50) NOT NULL,
    [surname]  [varchar](50) NOT NULL,
    [phone]    [varchar](20) NULL,
    [address]  [varchar](50) NULL,
    [password] [varchar](50) NULL,
    [isAdmin]  [bit] NOT NULL DEFAULT 0
)

CREATE TABLE [dbo].[Order]
(
    [id]        [int] IDENTITY(1, 1) NOT NULL PRIMARY KEY,
    [date]      [datetime] NULL,
    [total]     [decimal] NULL,
    [address]   [text] NULL,
    [note]      [text] NULL,
    [status]    [int] NOT NULL,
    [customer]  [varchar](50) NOT NULL FOREIGN KEY REFERENCES [dbo].[User](email)
)
  
```

2.2. Webservice

This section describes the webservice, which is a system component containing the data access layer and RESTful services.

2.2.1 Data Access Layer

The data access layer, as its name suggests, is used to give the application access to the data stored in the database. This layer of our system uses DAO pattern to separate low-level data accessing API or operations from high-level business services. The pattern consists of interfaces, DAO classes implementing the interfaces and model classes.

The data handling is a sensitive and essential feature of the system, therefore it must be secure and accurate. The concurrency issue of our project occurs here alongside security features, such as authorization and prevention of SQL code injection.

ADO.NET

To access the database from our system we used the ADO.NET library, which is provided by Microsoft. The ADO.NET library separates data access from data manipulation into discrete components. Using ADO.NET we connect to a database, execute commands, and retrieve results. ADO.NET is the most direct method of data access within the .NET Framework, meaning it is low level (Microsoft, 2021). Because of this, we had to write our own SQL statements, create connections and choose when to use a transaction. As a result, ADO.NET is more versatile and gives more control of the execution to us developers, then for example, Entity Framework.

Concurrency issue

In our system, the concurrency issue occurs when a user places a new Order. When that happens, the system must ensure that the desired products are in stock, there is enough of them, and that the stock gets decreased correctly, without any other order placed at the same time interfering with it. This process happens in the data access layer, in a method called *CreateAsync*, which takes in one parameter, an Order object.

First, a new SQL Connection object is created, and the connection is opened. A new *SqlTransaction* object is created by calling the *BeginTransaction* on the connection object.

At first, when we were setting the isolation level of the transaction, we thought that using Read Committed isolation level was enough. This isolation level uses shared locks that lock the data that is being read to avoid dirty reads, but the data can still be changed before the end of the transaction, which results in non-repeatable reads or phantom data (Coulouris, 2021). We

assumed this isolation level was sufficient because the method only reads the data once, but since it uses shared locks, the stock data can be read by two concurrent transactions and overwritten while the transactions are still happening. This could result in an undesirable outcome with a negative stock amount.

After this realization, we switched to using Repeatable Read, which locks the data that is used in a query, preventing other transactions from updating it while the transaction is proceeding. This prevents non-repeatable reads, but phantom rows, which do not affect this operation, are still possible.

The *CreateAsync* method continues with the creation of *SqlCommand* object, to which the transaction is set. The first query to be executed is the insertion of the Order data into the database. When the Order is inserted using the *ExecuteScalar* method, an id of the order in the form of an integer is returned. This id is later used throughout the transaction.

The next step is the iteration of all the products in the order, for which *LineItem* objects are used. For each of the *LineItem* objects, we check and decrease the stock of its corresponding product in the database. For this, the *DecreaseStockWithCheck* method is used, which is in the *ProductSizeStockDAO* class. To this method we pass the *SqlCommand* object, the id of the product, the id of the product's size and the quantity of the product. In the method, we first get the stock and compare its amount to the desired quantity. If the stock is less than the desired amount, a custom *ProductOutOfStockException* is thrown. On the other hand, if the stock is equal to or more than the desired quantity, the stock is then decreased.

After the stock is decreased, the products in the order need to be saved into the database. For this, we use *CreateLineItemAsync* method in the *LineItemDAO* class. Here we also pass the *SqlCommand* object, the id of the order and the *LineItem* object. At this point, we have inserted the required data into the database. After this, the transaction is committed.

After the transaction commit, we check for the *ProductOutOfStockException*. If this exception is caught, the transaction is rolled back and the exception with a custom message is thrown again, so it can pass to the upper layers of the system to the user. Then, we have a catch for the

exception, where the same step repeats. In the *finally* clause of the try catch block, the connection is closed. After the block, the id of the order is returned.

```
public async Task<int> CreateAsync(Order order)
{
    int id = -1;
    using SqlConnection connection = new SqlConnection(connectionString);

    connection.Open();
    SqlTransaction transaction = connection.BeginTransaction(IsolationLevel.RepeatableRead);

    SqlCommand command = connection.CreateCommand();
    command.Transaction = transaction;
    try
    {
        command.CommandText = "INSERT INTO dbo.[Order] (date, total, address, note, status, customer) VALUES " +
            "(@date, @total, @address, @note, @status, @customer); SELECT CAST(scope_identity() AS int)";
        command.Parameters.AddWithValue("@date", DateTime.Now);
        command.Parameters.AddWithValue("@total", order.TotalPrice);
        command.Parameters.AddWithValue("@address", order.Address);
        command.Parameters.AddWithValue("@note", (order.Note == null ? "" : order.Note));
        command.Parameters.AddWithValue("@status", order.Status);
        command.Parameters.AddWithValue("@customer", order.User.Email);
        id = (int)command.ExecuteScalar();
        order.Id = id;

        foreach(LineItem item in order.Items)
        {
            await productSizeStockDAO.DecreaseStockWithCheckAsync(command, item.Product.Id, item.SizeId, item.Quantity);
            await lineItemDAO.CreateAsync(command, id, item);
        }
        transaction.Commit();
    }
    catch (ProductOutOfStockException outOfStockEx)
    {
        transaction.Rollback();
        throw new ProductOutOfStockException($"The product's stock is less then desired! Error: '{outOfStockEx.Message}'.", outOfStockEx);
    }

    catch (Exception ex)
    {
        transaction.Rollback();
        throw new Exception($"An error ocured while creating a new order: '{ex.Message}'.", ex);
    }
    finally
    {
        connection.Close();
    }
    return id;
}
```

Security

SQL injection

To prevent SQL injection in our system, all SQL queries in the data access layer use command parameters. The command text contains parameter placeholders, which use an '@' symbol prefix on the parameter name. Each parameter is then added to the command with a real value. These placeholders are filled in with actual parameter values when the *SqlCommand* is executed.

In the figure below is an example of deleting a product from our database using the product's ID. Here we use an '@id' placeholder, which we set to the actual value of product's ID using the *AddWithValue* method.

```
SqlCommand command = new SqlCommand("DELETE FROM Product WHERE id = @id", connection)
command.Parameters.AddWithValue("@id", id);
```

Password hashing

Password hashing is a security procedure used to store passwords in a secure way that is resistant to attacks. For the password hashing we decided to use a *BCrypt* library.

BCrypt is designed to be slow and resource intensive, which is an advantage against brute-force and dictionary attacks, while being easy to use and implement. It incorporates a salt value, which is a random string of data, to further increase the complexity of the hash and make it more difficult to crack.

When a user creates an account and provides a password, the password is passed through a *HashPassword* method, shown in the figure below, to produce a unique hash value together with a salt. The hash value is then stored in the database instead of the plaintext password.

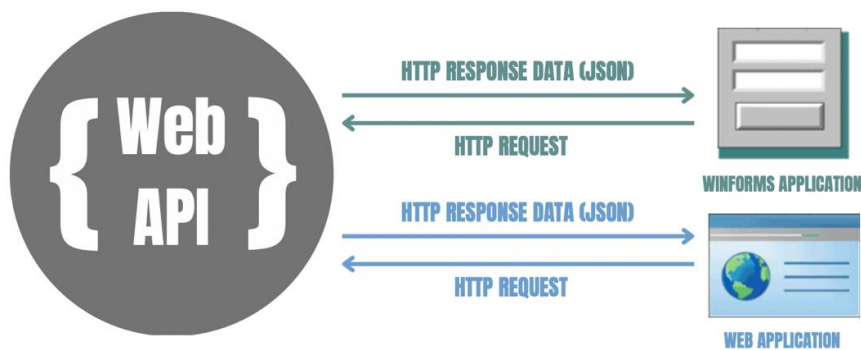
```
string passwordHash = BCryptTool.HashPassword(user.Password);
```

When the user attempts to log into the application, their password is hashed using the same method and the resulting hash is compared to the stored hash value. If the hashes match, the user is authenticated.

```
bool statement = BCryptTool.ValidatePassword(password, user.Password);
```

2.2.2 RESTful API

Our application programming interface (API) adheres to the Representational State Transfer (REST) architectural style, which defines a set of constraints for creating Web services in distributed systems. REST allows for simple interfaces to transmit data over a standardized interface (HTTP, HTTPS) without an additional messaging layer (Troelsen, 2022).



In our system, for each model class we have a controller class that contains specific HTTP methods with the selected crud functionalities for that model. An API method first calls a method in the data access layer to get the data as a model object. The model object is then mapped to a data transfer object using the AutoMapper library. The DTO is returned together with an *ActionResult* with an HTTP response status code, depending on the success of the completion of the requests. An example of an API method to get an *OrderDTO* by an *Order* id is displayed in the figure below.

```
[HttpGet("{id}")]
public async Task<ActionResult<OrderDto>> Get(int id)
{
    Order order = await _orderDataAccess.GetByIdAsync(id);
    if (order == null)
        return NotFound();

    OrderDto orderDto = _mapper.Map<OrderDto>(order);
    return Ok(orderDto);
}
```

Autmapper

Autmapper is a popular library for ASP.NET Core that simplifies the process of mapping between objects of different types (CodeMaze, 2022). It allows us to define rules for how the properties of one object should be mapped to the properties of another object, and then automatically applies these rules when performing the mapping.

In our project we implemented the AutoMapper by dependency injection and then we configured the mapping in the separate classes that are inheriting the *Profile* interface, which is part of the AutoMapper library. In most cases AutoMapper is able to map objects without pre-set configuration. An example of mapper profile and the mapping itself is shown in the figures below.

```
public class ProductProfile : Profile
{
    1 reference
    public ProductProfile()
    {
        CreateMap<ProductSizeStock, ProductSizeStockDto>()
            .ReverseMap();

        CreateMap<Product, ProductDto>()
            .ReverseMap();
    }
}

ProductDto productDto = _mapper.Map<ProductDto>(product);
```

Validation filters

Validation filters are a useful feature in ASP.NET Core that allows us to perform validation checks on incoming HTTP requests. They can be used to ensure that the request meets certain criteria, such as having required parameters or a valid format (Microsoft, 2022).

In our application we implemented the validation filter by inheriting the *IActionFilter*.

```
public class ValidationFilter : IActionFilter
{
    0 references
    public void OnActionExecuting(ActionExecutingContext context)
    {
        var param = context.ActionArguments.SingleOrDefault(p => p.Value is IEntity);
        if (param.Value == null)
        {
            context.Result = new BadRequestObjectResult("Object is null");
            return;
        }
        if (!context.ModelState.IsValid)
        {
            context.Result = new UnprocessableEntityObjectResult(context.ModelState);
        }
    }
}
```

In the filter we validate whether the required query parameter is inherited from the *IEntity* interface, which all our DTO classes are. Then we check if the query parameter meets the criteria of the validation attributes that we set for certain DTO class properties. The criteria of the validation attributes are set up corresponding to data types of columns that we use in the database.

We applied validation filter locally on the methods that are taking some arguments that need to be validated. Example is below:

```
[HttpPost]
[ServiceFilter(typeof(ValidationFilter))]
0 references
public async Task<ActionResult<string>> Post([FromBody] UserDto newUserDto)
{
    User user = _mapper.Map<User>(newUserDto);
    string email = await _userDataAccess.CreateAsync(user);
    if(email == null || email.Equals("")) { return BadRequest(); }
    return Ok(email);
}
```

The filter is added to service container in the *Program* class of the *WebApi* project as follows:

```
//Register scoped filters
builder.Services.AddScoped<ValidationFilter>();
```

2.2.3. REST CLIENT

In the client part of our client-server system, we also utilize RestSharp to create Rest Client, which requests data from the Rest API (RestSharp.dev, n.d.). This process is an important aspect, because it allows the separation of the user interface concerns from the data storage concerns, which adheres to the one of the six REST constraints, by enforcing the principle of separation of concerns.

In order to receive all the data needed in the client from the server a Data transfer object (DTO) design pattern is used. This allows us to transfer data from different domain objects in a single DTO. DTOs also give us the ability to hide particular properties that the client side is not supposed to see and remove circular dependencies.

```

namespace WebApiClient.DTOs
{
    public class ProductDto
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Description { get; set; }
        public decimal Price { get; set; }
        public string Category { get; set; }
        public IEnumerable<ProductSizeStockDto> ProductSizeStocks { get; set; }
    }
}

```

To implement the DTO design pattern in our system, we generated DTO classes from the properties of our model classes. Displayed in the figure above is one of the examples of the DTOs benefit of joining multiple datapoints into one object, which happens with the *ProductDTO*, as the object consists of the Product class properties with a list of product sizes and their stock. This was done because each product can have multiple sizes and each size has its own stock.

To enhance readability, we stored these classes in dedicated folders in the web service project and in the web application project.

2.3. Webpage

In this section, we focus on the webpage, which represents the presentation tier in our project.

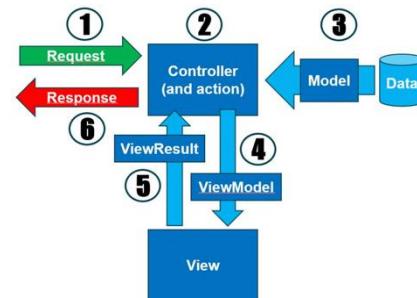
Our webpage was built using ASP.NET MVC.

2.3.1. MVC

MVC is an architectural pattern which separates the web application into three parts – Model, View and Controller.

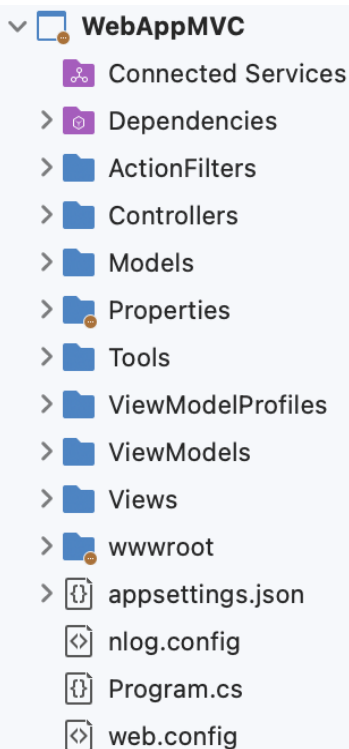
Model stands for data, which are retrieved from the database. View represents the User Interface, which is made up of HTML, CSS and razor syntax for simple

communication with the controller layer. The controller handles HTTP requests from the user



and renders the View accordingly, creating a response that is then sent back to the browser (Microsoft, 2022).

In our project, we have decided to implement strongly-typed Views, which bind a specific ViewModel class to the corresponding View. To do this, we used AutoMapper to map from our DTO model classes to specific ViewModels. This happens in the View Model Profile classes and the Controller classes.



```
public class UserProfile : Profile
{
    public UserProfile()
    {
        CreateMap<UserDto, UserDetailsVM>();
        CreateMap<UserDto, UserEditVM>()
            .ReverseMap();
    }
}
```

```
public async Task<ActionResult> Edit(string email)
{
    var userDto = await _userClient.GetByEmailAsync(email);
    UserEditVM userEditVM = _mapper.Map<UserEditVM>(userDto);
    return View(userEditVM);
}
```

In order to create a strongly-typed View, we need to specify the type of ViewModel being passed to the View by using the @model directive. The example bellow shows the *User/Edit* View.

```
@model WebAppMVC.ViewModels.UserEditVM
```

```
<h1 class="account-title">@Html.DisplayFor(model => model.PageTitle)</h1>
```

We decided to implement this solution because it brings the following advantages (DotNetTutorials, n.d.):

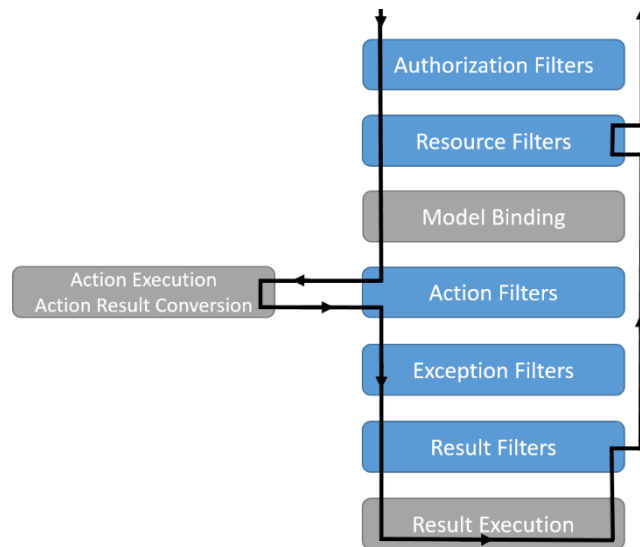
1. Compile-time error checking
2. Visual studio IntelliSense support, for faster coding
3. Misspelling a property name inside the View will result in a compile-time error, rather than a runtime one

This is not implemented for all Views, such as the *About* and *Index* Views in the *HomeController*, or the *Index* view in the *CartController*, as they do not bind to a specific ViewModel class. In this case we are using ViewData to pass data from the controller to the View, which can be referred to as using a loosely typed view.

Authorization

The important aspect of web application development is ensuring that only authorized users have access to certain resources or functionality. ASP.NET Core provides several options for implementing authorization in a web application.

One of them is to use the built-in authorization attributes. It's safe and easy to implement, which is why we chose it for



our project.

We applied the *Authorize* attribute, which is written in the line above the method in square brackets, to all action methods that needed protection against un-authorized requests. Example can be seen in the action method *Details()*, which is returning confidential information

about

user.

Here we applied the *Authorize* attribute locally just for this method.

```
[Authorize]
public async Task<ActionResult> Details()
{
    string email = User.FindFirst(ClaimTypes.Email).Value;

    UserDto userDto = await _userClient.GetByEmailAsync(email);
    UserDetailsVM userDetailsVM = _mapper.Map<UserDetailsVM>(userDto);

    return View(userDetailsVM);
}
```

The request from the user first runs through the authorization filter, which has the highest priority of all filters in ASP.NET Core.

Here it is determined whether the user is authorized for the given request. In our case, the authorization filter is checking whether the data from a cookie are identical with the data that are saved, when we issued the cookie for an authenticated user.

Protection against cross-site request forgery

Anti-forgery tokens, also known as Cross-Site Request Forgery (CSRF) tokens, are an important security feature that helps protect web applications against malicious attacks. These tokens are used to verify that a particular request to a web application was intended by the user, rather than being an unauthorized request from a third party.

We use the *ValidateAntiForgeryToken* attribute to apply anti-forgery protection to a particular action. This attribute checks for a valid anti-forgery token in the request and ensures that it matches the token that was previously generated by the server. If the token does not match, the request is considered to be a forgery and is rejected.

```

[HttpPost]
[Authorize]
[ValidateAntiForgeryToken]
public async Task<ActionResult> UpdatePassword(UserEditVM user)
{
    if (user.Email != User.FindFirst(ClaimTypes.Email).Value)
        return View("/Views/Shared/ActionForbidden.cshtml");

    UserDto userDto = _mapper.Map<UserDto>(user);
    userDto.Email = User.FindFirst(ClaimTypes.Email).Value;

    await _userClient.UpdatePasswordAsync(userDto);
    await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);

    return RedirectToAction("Login", "Authentication");
}

```

In the image above, the implementation of *ValidateAntiForgeryToken* is shown in code. We also included *@Html.AntiForgeryToken()* method in the forms in all razor sharp views that use anti forgery validation.

Caching

In-memory caching is a useful technique for improving the performance of ASP.NET Core web applications. It involves storing data in memory for quick access, rather than fetching it from a slower data store such as a database or external web service.

In our application we are caching categories of the products, here is an example (to see it more detailed, please zoom in):

```

public ProductController(IProductClient prodClient, ICategoryClient catClient, IMapper mapper, IMemoryCache cache)
{
    _cache = cache;
    _productclient = prodClient;
    _categoryclient = catClient;
    _mapper = mapper;
}

```

First, we inject the *IMemoryCache* into the *ProductController*. Then, in the listing action method, we check if the *categories* data is available in the cache. If the data is present in the cache, we retrieve that. If the data is not available in the cache, we fetch the data from the database and at the same time populate it into the cache. Additionally, we are setting a sliding expiration time of 12 hours using the *MemoryCacheEntryOptions*.

```

//If possible get from cache
if (_cache.TryGetValue(categoryListCacheKey, out IEnumerable<CategoryVM> categories))
{
    productIndexVM.Categories = categories;
}
//Else get from DB
else
{
    try
    {
        //NOTE: Semaphore is here because we don't want a case when two user get data from DB and then write it in cache at the same time.
        await semaphore.WaitAsync();

        //Get categories from DB
        IEnumerable<CategoryDto> categoriesDB = await _categoryclient.GetAllAsync();
        IEnumerable<CategoryVM> categoriesVM = categoriesDB.Select(categoryDto => _mapper.Map<CategoryVM>(categoryDto));

        productIndexVM.Categories = categoriesVM;

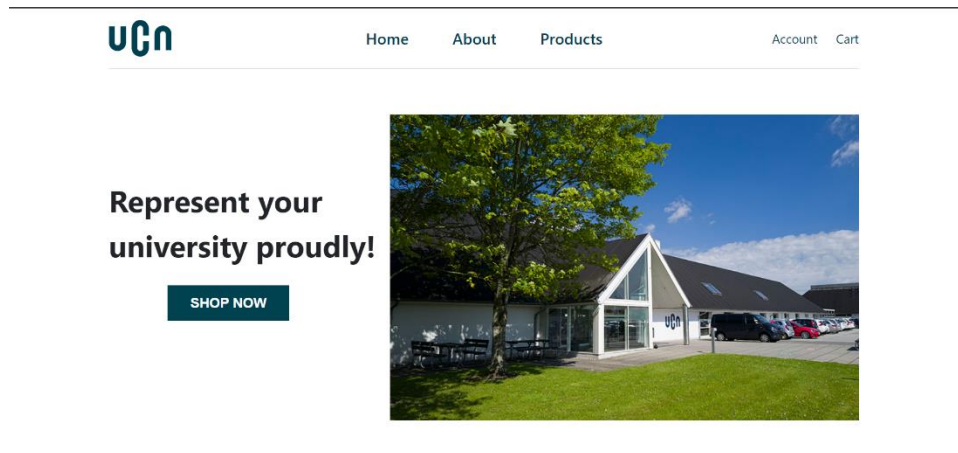
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromHours(12))
            .SetAbsoluteExpiration(TimeSpan.FromDays(2))
            .SetPriority(CacheItemPriority.Normal)
            .SetSize(1024);
        _cache.Set(categoryListCacheKey, categoriesVM, cacheEntryOptions);
    }
    finally
    {
        semaphore.Release();
    }
}
return View(productIndexVM);

```

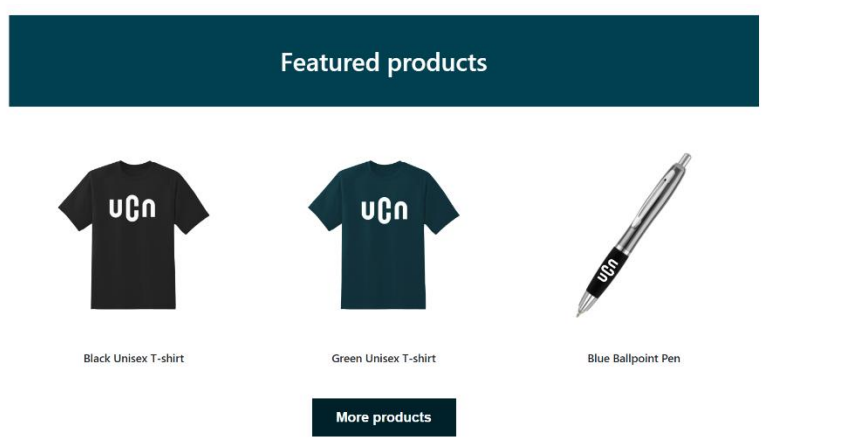
Even though the *IMemoryCache* is thread-safe, it is prone to race conditions. For instance, if the cache is empty and two users try to access data at the same time, there is a chance that both users may fetch the data from the database and populate the cache. This is not desirable. To solve these kinds of issues, we implement a locking mechanism for the cache.

2.3.2. Website design

For the design of the website, we chose a simple, yet informative and modern look. We used the official colors of UCN to keep the same feel and identity throughout the whole website. The homepage contains a menu at the top of the page, a SHOP NOW button, to encourage users to shop and see all the products.



After scrolling to the second part of the home page, the user lands on a Featured products page, where three featured products are shown, which link the user right to the product's details page. At the bottom of the featured product is More products button, which takes the user to the products page.



2.3.3. Cart

Because HTTP is a stateless protocol, its requests don't preserve data. In order to have some data available between multiple requests throughout a session, we had to use a cookie. The cookie was used for storing contents of the user's cart, as the user can dynamically change the contents of their cart at any moment, while they browse the website. The cookie is used in a session state of the application. The session is created in the MVC layer in Program.cs class, which is displayed in the figure below.

```
builder.Services.AddSession(options =>
{
    options.IdleTimeout = TimeSpan.FromSeconds(3600); // one hour
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

When the session is created, a few options are set, such as the *IdleTimeout* of the session. The *IdleTimeout* property indicates how long the session can be idle, before its contents are abandoned. We set the timeout to 3600 seconds, which equals to one hour. We also set the *IsEssential* property to true. This ensures that the cookie is essential to the function of the application and can bypass consent policy check. This cookie does not collect any data about user, only the content of their cart, which is necessary for any order to be placed by a user. After the application is built, we call the *UseSession()* method on it, to enable sessions for it.

```
public static class CartTool
{
    private const string shoppingCartKey = "shopping_cart";

    public static OrderCreateVM GetCart(this HttpContext context)
    {
        var cart = context.Session.Get<OrderCreateVM>(shoppingCartKey);

        if (cart == null)
            cart = new OrderCreateVM() { Items = new List<LineItemVM>() };

        if (cart.Items == null)
            cart.Items = new List<LineItemVM>();

        cart.TotalPrice = cart.Items.Sum(item => item.Price * item.Quantity) + 35;

        return cart;
    }

    public static void SaveCart(this HttpContext context, OrderCreateVM cart)
    {
        context.Session.Set<OrderCreateVM>(shoppingCartKey, cart);
    }
}
```

To use the cookie and the data stored in it, we created a static class called *CartTool*, displayed in the figure above. The class contains two methods, one for getting the content of the cookie – *GetCart()*, and another for saving the content of the cookie – *SaveCart()*. For storing of the cart, we use the *OrderCreateVM* model, as one of its properties is a list of line items. Additionally, this View model is used in the *Order/Create* View. These methods are extension methods of the *HttpContext* class, so when accessing the methods, we only call the method on the *HttpContext* class as *HttpContext.GetCart()*.

Ref: Rick Anderson, Kirk Larkin, and Diana LaRose (09/30/2022) Session and state management in ASP.NET Core.

2.4. Desktop client app

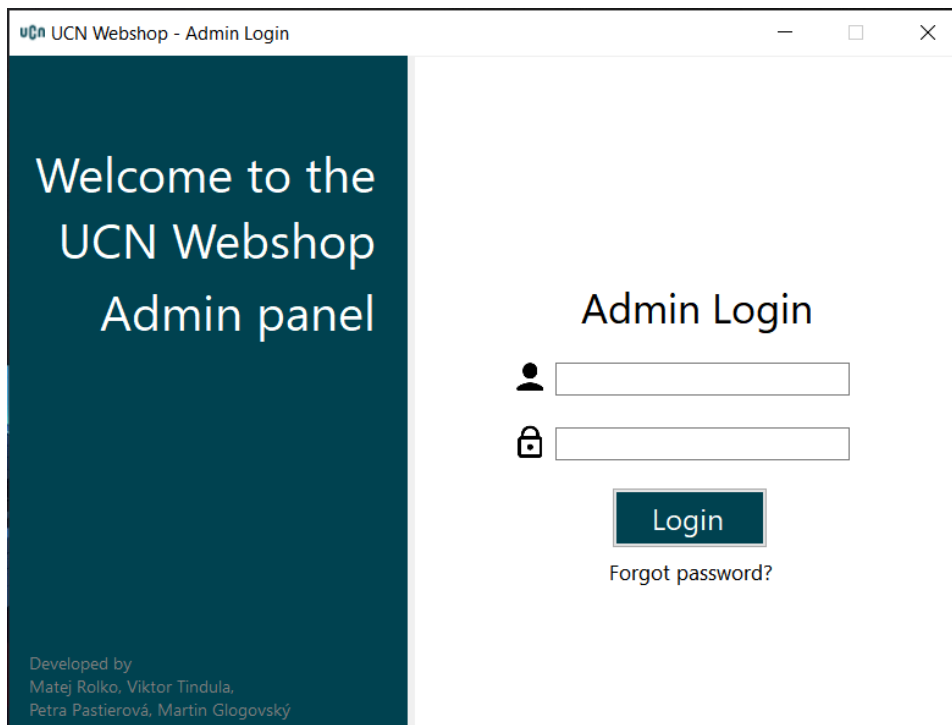
For the dedicated client app, we created a simple Windows Forms application, that is used to show basic data for the administrators of the web shop.

Windows Forms, also known as WinForms, is a free open-source Graphical User Interface library included in the .NET framework.

When a user opens the application, they land on a login screen. Here, the user must enter their login details, to be able to access the application. The login details are checked in the database and the user must be saved as an administrator. If that is not the case, the user will not be let into the application. The login mechanism uses a token, which is locally cached upon the login of the user and sent with all the requests to the database for authorization.

The main feature of the application, based on the main user stories, is to see all the products, be able to edit or delete them and to see the list of orders.

In the figure below is the login screen of the dedicated client application.



2.5. Other system functionalities

Error handling

Exception filters in ASP.NET Core are a feature that allow developers to specify logic that will execute when an exception occurs during the execution of a controller action.

We implemented the exception filter as an attribute that we applied locally on the controller class in the web page:

```
[ServiceFilter(typeof(ExceptionFilter))]  
1 reference  
public class AuthenticationController : Controller  
{
```

And globally to all the controllers in the web API:

```
//Add controllers with option with global filters  
builder.Services.AddControllers(options =>  
{  
    options.Filters.Add<ExceptionFilter>();  
});
```

One of the primary benefits of exception filters is that they allow developers to handle common errors in a centralized and consistent manner (Red gate, 2014). This can be particularly useful for handling errors such as invalid input or resource not found, which may occur frequently in a web application (CodeMaze, 2022).

In addition to handling specific errors, our exception filters can also perform logging of the exception by using a logging middleware.

Logging middleware

Logging middleware is a type of software component that is designed to record and log information about the processing of requests and responses in a web application. Logging middleware can be useful for a variety of purposes, including debugging, performance monitoring, and security.








In our application we inject the logging middleware to the exception filter class from where the exceptions are logged and saved locally on pre-configured location.

```
public ExceptionFilter(ILoggerManager logger)
{
    _logger = logger;
}

0 references
public override void OnException(ExceptionContext filterContext)
{
    _logger.LogInfo(filterContext.Exception.Message);
}
```

The location of the folder where new reports about the exceptions will be created can be configured in the *nlog.config* file.

Every day a new file is created, and it stores the exceptions from that day. The folder looks as follows:

 2022-11-11_logfile	11/11/2022 12:42	Text Document
 2022-11-12_logfile	12/11/2022 15:38	Text Document
 2022-11-13_logfile	13/11/2022 09:28	Text Document
 2022-11-14_logfile	14/11/2022 12:51	Text Document
 2022-11-15_logfile	15/11/2022 13:33	Text Document
 2022-11-18_logfile	18/11/2022 11:55	Text Document
 2022-11-20_logfile	20/11/2022 13:00	Text Document

Asynchronous programming

Asynchronous programming is a useful technique for improving the performance of ASP.NET Core applications. We decided to implement asynchronous programming, because it allowed us to write code that can perform tasks concurrently, rather than sequentially, which can help to improve the responsiveness of our application and make better use of available resources.

In order to achieve fully asynchronous flow of the program, we have made all the methods asynchronous. Here is an example of asynchronous code in our application:

```

[HttpPost]
public async Task<ActionResult<int>> Post([FromBody] OrderDto newOrderDto)
{
    int id = await _orderDataAccess.CreateAsync(_mapper.Map<Order>(newOrderDto));

    if (id <= 0)
        return BadRequest();

    return Ok(id);
}

```

The `async` keyword is used to indicate that a method contains asynchronous code, and the `await` keyword is used to pause the execution of the method until a task is completed.

By using asynchronous programming, we improved the performance of our application by allowing it to continue processing other tasks while it is waiting for the results of long-running tasks. This helps to improve the user experience and make our application more efficient.

Authentication

The authentication in our project was needed in order to be able to authorize requests coming in from users. There are several ways to implement authentication in ASP.NET Core. One common approach is to use the built-in authentication middleware, which supports a variety of authentication schemes such as cookie-based authentication, OAuth, and OpenID Connect.

Another approach is to implement custom authentication using the ASP.NET Core Identity framework, which provides a set of APIs and libraries for implementing user authentication and authorization in a web application. It allows developers to create custom login forms, manage user accounts and passwords, and implement role-based access control (Microsoft, 2022). We decided for the second approach because it was easier to implement, and it fulfills all the requirements we needed.

```

[ServiceFilter(typeof(ValidationFilter))]
[HttpPost]
public async Task<ActionResult> Login([FromBody] LoginModelDto loginData)
{
    User? user = await _userDataAccess.LoginAsync(loginData.Email, loginData.Password);

    if (user != null)
    {
        var secretKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("superSecretKey@345"));
        var signinCredentials = new SigningCredentials(secretKey, SecurityAlgorithms.HmacSha256);
        var tokenOptions = new JwtSecurityToken(
            issuer: "https://localhost:5001",
            audience: "https://localhost:5001",
            claims: new List<Claim>{
                new Claim(ClaimTypes.Email, user.Email),
                new Claim("address", user.Address),
                new Claim(ClaimTypes.MobilePhone, user.PhoneNumber),
                new Claim(ClaimTypes.Name, user.Name),
                new Claim(ClaimTypes.Surname, user.Surname),
                new Claim(ClaimTypes.Role, user.IsAdmin ? "Admin" : "Casual"),
            },
            expires: DateTime.Now.AddMinutes(20),
            signingCredentials: signinCredentials
        );
        return Ok(new
        {
            token = new JwtSecurityTokenHandler().WriteToken(tokenOptions),
            expiration = tokenOptions.ValidTo
        });
    }
    return Unauthorized();
}

```

In the *Login()* action method above, which is in the *AuthenticationController* in the web API, are receiving the user's login data and comparing it with the one from the database. If the user is recognized, a *JwtSecurityToken* with setup and user information is issued and returned together with the *Ok* status code.

The issued *JwtSecurityToken* is returned to the other *Login()* action method in the Web app MVC. Here, the data from the token are parsed and sent in the request header back to the user.


```

[HttpPost]
public async Task<IActionResult> Login([FromForm] LoginVM loginInfo)
{
    if (!ModelState.IsValid)
    {
        return View(loginInfo);
    }
    LoginModelDto loginModelDto = _mapper.Map<LoginModelDto>(loginInfo);
    string result = await _client.LoginAsync(loginModelDto);
    string? TokenString = (string?)JObject.Parse(result)["token"];

    if (TokenString != null)
    {
        JwtSecurityToken Jst = new(TokenString);

        List<Claim> theApiClaims = (List<Claim>)Jst.Claims.ToList();
        theApiClaims.Add(new Claim("token", TokenString));

        var claimsIdentity = new ClaimsIdentity(theApiClaims, "Login");
        var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);

        await HttpContext.SignInAsync(CookieAuthenticationDefaults.AuthenticationScheme,
            claimsPrincipal);
        return RedirectToAction("Index", "Home");
    }
    else
    {
        return View(loginInfo);
    }
}

```

Dependency injection

Dependency injection is a software design pattern that allows a class to receive its dependencies from the outside rather than creating them itself. This is useful because it allows for greater flexibility and modularity in the design of an application. It also makes it easier to test classes in isolation, since their dependencies can be mocked or stubbed.

ASP.NET Core includes built-in support for dependency injection and in our project, we aimed to implement dependency injection in all possible places where it would make sense.

One of many examples of dependency injection in our application can be found in the *OrderController* in the front-end.

```

private readonly IOrderClient _client;
private readonly IMapper _mapper;

0 references
public OrderController(IOrderClient client, IMapper mapper)
{
    _client = client;
    _mapper = mapper;
}

```

Here, in the constructor of the *OrderController* we inject the needed services, *IOrderClient* and *IMapper*, and then we assign them as private readonly fields of the *OrderController* class. To build these dependencies, we use program class as a configuration container for all injected services (both in front-end and back-end).

```

//Order client
string orderUrl = "https://localhost:44346/api/v1/orders";
IOrderClient orderClient = new OrderClient(orderUrl);
builder.Services.AddScoped<IOrderClient>((cs) => orderClient);

```

Here we create a new instance of the *OrderClient* and we set it as an according service provider for the *IOrderClient*.

Then we add the *IOrderClient* to the application builder services. Notice the method *AddScoped()*, which means that the added service object will be same within a request, but different across different requests.

```

//AutoMapper config
builder.Services.AddAutoMapper(typeof(Program));

```

The *IMapper* is added in the extension method of *IServiceCollection* *AddAutoMapper()*.

3. Testing

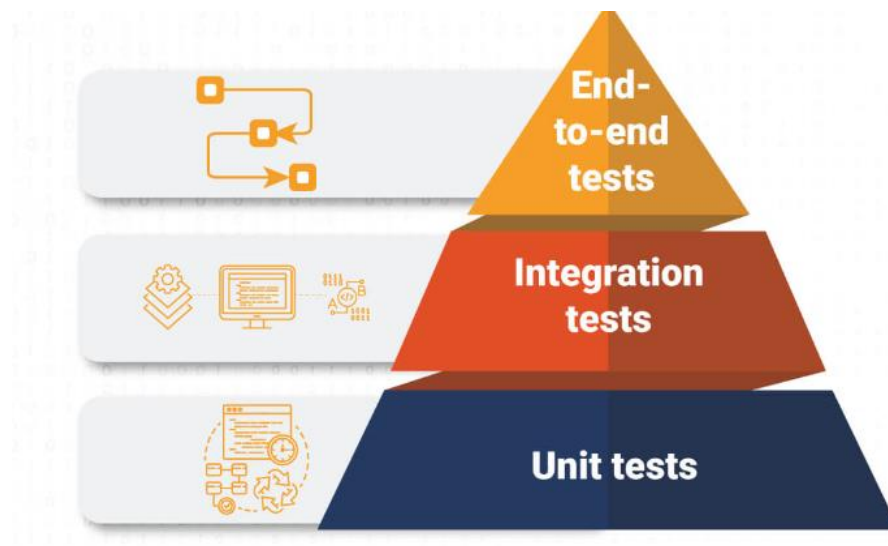
Testing is an important aspect of software development, as it helps ensure that an application is functioning correctly and meeting the requirements set forth in the design phase. In an ASP.NET Core application, testing can be performed at various levels, including unit testing, integration testing, and end-to-end testing (Medium.com, 2021).

Unit testing involves testing individual units of code, such as methods or classes, in isolation from the rest of the application. This allows developers to verify that each unit is functioning as expected and that any interactions between units are handled correctly.

Integration testing involves testing the integration of different units of the application, such as a controller and a database. This helps ensure that the different parts of the application are working together as intended.

End-to-end testing involves testing the application as a whole, from start to finish. This can involve testing the application's user interface, as well as its integration with external systems.

The testing pyramid is a graphical representation of the different types of tests that should be included in a software development project. According to the testing pyramid, there should be a larger number of unit tests than integration tests, and a larger number of integration tests than end-to-end tests. This is because unit tests are faster and more isolated than integration tests, and integration tests are faster and more isolated than end-to-end tests.



Although we understand that it's generally a good idea to have a combination of both types of tests in a test suite, in our project we focused mainly on the integration testing.

The reason for this is mainly because all parts of our software are highly dependent on each other and therefore integration testing was more relevant in a sense of value for our project and time efficiency.

Testing of the data access layer

For the purpose of testing the data access layer (DAL) we created a separate local test database. There are several benefits to using a test database when testing the DAL:

1. A test database allows us to test the DAL with real data, rather than using mock data. This can help us ensure that the DAL is able to handle different types of data and can help us uncover issues that might not be apparent when using mock data.
2. A test database allows us to test the DAL in a more realistic environment. When using a test database, we can test the DAL with a database schema and data that is similar to the production environment.
3. A test database can help us test the DAL's performance and scalability. By using a test database, we can simulate different workloads and test the DAL's ability to handle large amounts of data or high levels of concurrency.

In the figure below, there is an example of the tests created using Xunit testing framework.

```
//GetAll
[Fact]

public async Task User_GetAll_Success()
{
    //ARRANGE

    //ACT
    IEnumerable<User> result = await _UserDataAccess.GetAllAsync();
    //ASSERT
    Assert.True(result.Count() > 0);
}

//GetByEmailAsync
[Fact]

public async Task User_GetByEmailAsync_Success()
{
    //ARRANGE
    //ACT
    var result = await _UserDataAccess.GetByEmailAsync(_existingUser.Email);
    //ASSERT
    Assert.NotNull(result);
}
```

When testing the DAL, we tested mainly for successful tests scenarios. If we had more time, we would also test other possible scenarios. All the scripts that are needed for the construction of the test database are available in the folder in the *BackendIntegrationTest* project.

Testing of the web API

In the integration tests of the web API's controllers, we also tested for other possible scenarios, as it was very important for the flow of our program. Below is a snapshot of the test code, where we are testing custom error responses that the method should return in case of a problem.

```
[Fact]
0 | 0 references
public async Task Post_Incorrect_Data_In_Product_Should_Return_UnprocessableEntity422()
{
    //ARRANGE
    ProductDto entity = new();
    entity.Name = "";
    var json = JsonConvert.SerializeObject(entity);
    var content = new StringContent(json, Encoding.UTF8, "application/json");
    //ACT
    var response = await _client.PostAsync($"{productUrl}", content);
    //ASSERT
    HttpStatusCode statusCode = response.StatusCode;
    Assert.Equal(HttpStatusCode.UnprocessableEntity, statusCode);
}

[Fact]
0 | 0 references
public async Task Post_Wrong_Data_Type_Should_Return__BadRequestObjectResult400()
{
    //ARRANGE
    var json = JsonConvert.SerializeObject("");
    var content = new StringContent(json, Encoding.UTF8, "application/json");
    //ACT
    var response = await _client.PostAsync($"{productUrl}", content);
    //ASSERT
    HttpStatusCode statusCode = response.StatusCode;
    Assert.Equal(HttpStatusCode.BadRequest, statusCode);
}
```

Testing of the web page

The web page is the part of the code which is the least tested. It is also covered only by the integration tests in which are testing just for successful response status code.

```
//INDEX
[Fact]
0 references
public async Task Index_Returns_Success()
{
    //ARRANGE
    string url = "https://localhost:7183/Product";
    //ACT
    var response = await _client.GetAsync(url);
    //ASSERT
    response.EnsureSuccessStatusCode();
    Assert.Equal("text/html; charset=utf-8",
        response.Content.Headers.ContentType.ToString());
}
```

It is worth mentioning that in testing of the web page and the web API, we are using *Microsoft.AspNetCore.Mvc.Testing* package that allows us to fully inherit all the configuration from the *Program* class, so we are able test with all dependencies that are present in the production version of our application.

```
public class TestProductsController : IClassFixture<WebApplicationFactory<Program>>
```

4. Conclusion

In order to reach a conclusion, we must inspect the question we asked ourselves at the beginning of the project.

“How to create a working distributed system using the .NET Framework, while establishing that the system works across the network on all supported platforms and ensuring atomicity and in-real-time synchronization?”

To answer this question, we need to answer three additional questions first, that go more into detail about our system.

“How to solve concurrency issues that occur with collaborative processes when several users interact with the system?”

The concurrency issue in the project occurs when multiple users try to place an order with the same product(s) at the same time. This becomes a problem in the case of insufficient stock,

because it could lead to negative stock values in the database. To solve this issue, we decided to implement *RepeatableRead* isolation level in our transaction.

“How to establish proper system performance, ensuring that all system components communicate with each other correctly?”

One of the main practices of EXtreme Programming is Test-driven development. Even though we only had time to implement it partially in our sprints, testing made a significant difference when applied. It helped us check proper system functionality and determine whether we were meeting the requirements we set out to accomplish.

“How to create user-friendly UI in the webpage and the dedicated client application?”

We aimed to create a seamless user experience on the front-end of our application. The focus was on making the website and the dedicated client application minimalistic, concise and easy to navigate, even for a first-time user. We wanted to create that familiar UCN feel, which is why we used UCN’s official brand book for the colors and logos, to preserve the brand identity.

Conclusion

This semester was our first experience with using the .NET framework and fast-paced agile development. It took us some time to familiarize ourselves and keep up with the rapid introduction of new concepts into our studies, which felt really challenging at first, however as we progressed in our project, we started seeing all the ways in which .NET is a tool that works for the developer. Throughout the project we have learned to utilize the technologies in our favor and have gotten a deeper understanding of the individual system components and data passage between them. As a group, we feel like the product we delivered is solid, and meets the criteria and the learning objectives for this semester.

5. Group evaluation

We, as team members, have worked together on every project since the first semester, during which time we have gotten to know each other and fell into a familiar work dynamic. What was unfamiliar however, was the introduction of agile methodology into our work.

In sprint zero, we had enough time to finish everything we set to do and even had a chance to do some extra work, like creating the solution based on the architectural requirements for the project. However, in sprints one and two, we were unable to correctly estimate the time needed to finish all selected backlog items. Even though this was predicted as one of the most probable risks in our risk analysis, it still challenged us to think of a way to get back on track. By sprint three, we had a better grasp on how to work to successfully meet the deadline and in sprint four we missed it just barely, due to the unforeseen complications. Working with agile methodology while dealing with a very challenging assignment was difficult for all members of the group, each of us had tight schedules out of school and project development, so sometimes it was hard to align our schedules to meet all together. Also, the time frame for a system with such a wide scope was relatively small.

This was our first time working with Scrum, so it is not surprising it did not go exactly as planned, but that is all part of the learning experience. We now feel confident saying that if we were to do this again, we would be able to estimate the workload and plan for the sprints much better. We learned for example, that it is better to think smaller and plan for what we know we can finish in time than to “reach for the sky” and then underdeliver.

Despite everything, we think the team worked well together and was able to deliver on the product. It would help to have concrete roles set in place when it came to Scrum master and Product owner, mainly for consistency reasons, but we can acknowledge that it was important for the learning process that we swapped the roles out between each team member.

Appendices

A. List of references

Coulouris, G., Dollimore, J., Kindberg, T. & Blair, G. (2012). *Distributed Systems*. (5th edition). Addison-Wesley.

Technopedia Inc. (2021). *Three-Tier Architecture*. Retrieved December 19, 2022 from <https://www.techopedia.com/definition/24649/three-tier-architecture>

Terra, J. (2022). *What is Client-Server Architecture? Everything You Should Know*. Retrieved December 19, 2022 from <https://www.simplilearn.com/what-is-client-server-architecture-article>

Elmarsji, R., Navathe, S. (2011). *Fundamentals Of Database Systems*. (6th edition). Pearson.

Microsoft (2021). *ADO.NET Overview*. Retrieved December 19, 2022 from <https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview>

Troelsen, A. (2022). *Pro C# 10 with .NET 6 - Foundational Principles and Practices in Programming*. (11th edition). Apress.

CodeMaze (2022). *Getting Started with AutoMapper in ASP.NET Core*. Retrieved December 20, 2022 from <https://code-maze.com/automapper-net-core/>

Microsoft (2022). *Understanding Action Filters (C#)*. Retrieved December 20, 2022 from <https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/understanding-action-filters-cs>

RestSharp.dev (n.d.). *RestSharp Recommended usage*. Retrieved December 18, 2022 from <https://restsharp.dev/usage.html>

Microsoft (2022) *Overview of ASP.NET Core MVC*. Retrieved December 20, 2022 from <https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-7.0>

DotNetTutorials (n.d.). *Strongly Typed Views in ASP.NET MVC*. Retrieved December 17, 2022 from <https://dotnettutorials.net/lesson/strongly-typed-view-asp-net-mvc/>

CodeMaze (2022). *Global Error Handling in ASP.NET Core Web API*. Retrieved December 20, 2022 from <https://code-maze.com/global-error-handling-aspnetcore/>

Red Gate (2014). *Handling Errors Effectively in ASP.NET MVC*. Retrieved December 20, 2022 from <https://www.red-gate.com/simple-talk/development/dotnet-development/handling-errors-effectively-in-asp-net-mvc/>

Microsoft (2022). *Use cookie authentication without ASP.NET Core Identity*. Retrieved December 19, 2022 from <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/cookie?view=aspnetcore-7.0>

Medium.com (2022). *Testing ASP.NET Core Web APIs — A Detailed Guide*. Retrieved December 19, 2022 from <https://medium.com/technology-earnin/thorough-testing-of-asp-net-core-or-any-web-api-a87bd0585f9b>

B. Group contract

C. University College of Northern Denmark

D. CSC-CSD-S212

E. October 26, 2022

F. 3rd Semester Project
G. Group Cooperation Agreement

H. This 3rd Semester Project Group Cooperation Agreement dated October 26, 2022, is by and between the group members of Group 1 of CSC-CSD-S212. The agreement is designed to ensure a smooth cooperation during the Third Semester Project. It should increase the chances of success of the project. For this purpose, a number of rules were appointed which must be respected by all group members. Therefore, the group members agreed as follows:

I. 1. Group members

J. (a) The Group consists of 4 members; Martin Glogovský, Petra Pastierová, Matej Rolko and Viktor Tindula.

K. 2. Assignment details

L. (a) The deadline for this project and the date of hand-in is December 20, 2022 at 14:00.

M. (b) The supervisors of the assignment are Jens Henrik Vollesen, Gianna Belle and Jakob Farian Krarup.

N. 3. Communication

O. (a) For sharing files and documents file-sharing platforms, such as GitHub and Microsoft Teams are used.

P. (b) For communication purposes online platforms, such as Discord and Facebook Messenger are used.

Q. (c) Working time for major tasks is every workday between 8:30 a.m. to 4:00 p.m.

R. (d) Within collaboration hours, members are obligated to respond to messages within 15 minutes, unless given an appropriate reason not to.

S. (e) Members shall physically meet and collaborate on each day of the project unless given an appropriate reason not to.

- T. (f) If members must complete their tasks outside collaboration hours, they shall update the rest of the group about the completion of said task. If they are certain they are unable to finish before the next collaboration session, they must inform the other members immediately.

U. 4. Division of Work

- V. (a) The members agree to split work between them in accordance with the project plan.

W.5. Absence from physical meetings and shortcoming of delegated tasks

- X. (a) If a member does not show up to an agreed meeting, the supervisor is contacted unless their reason is decided to be sufficient by other group members present.
- Y. (b) If a member fails to complete their delegated task no later than 12 hours after its expected time, the next course of action is decided on by the other members and may be elevated to the supervisor.
- Z. (c) If a member's work is impeded by illness, or other sufficient reason, they are obligated to inform the others as soon as they are able to and shall make an effort to do partial work remotely, if at all possible. Their current tasks may then be delegated between all able-to-work group members.
- AA. (d) If a member's dedication to their tasks, work and collaboration efforts are not found satisfactory by the rest of the group, they may receive a strike. After receiving the previously agreed upon number of strikes (2), the situation may be elevated to the supervisor and further actions may be taken in collaboration with said supervisor.

BB. 6. Confirmation

- CC. (a) Submission of this form to the supervisor means that all members have agreed to the contents of this document and consider it binding.