# Haefele E-Commerce

# July 23, 2024

Contents

[Haefele E-Commerce 1](#_Toc172981296)

[July 23, 2024 1](#_Toc172981297)

[Business Requirements 2](#_Toc172981298)

[Target Audience 2](#_Toc172981299)

[Name of project 2](#_Toc172981300)

[Technical Specs 2](#_Toc172981301)

[Project 1 – hf.Api 3](#_Toc172981302)

[Project 2. hf.Application 4](#_Toc172981303)

[Project 3 – hf.Domain 5](#_Toc172981304)

[Project 4 - hf.Infrastructure 6](#_Toc172981305)

[Login 8](#_Toc172981306)

[User Stories 11](#_Toc172981307)

[Conclusion 12](#_Toc172981308)

[Improvements 13](#_Toc172981309)

### Business Requirements

1. Requirements for the Haefele E-Commerce are as follows:
   1. Add users to the system. Provide username and password as credentials.
   2. Allow the user to log in. The login screen also provides the user with a ‘Sign-up’ and

‘Forgot password’ link.

* 1. As an anonymous user:
     1. Allow a user to browse the product catalog without logging in.
     2. Add cart button next to each product. Selecting a product for purchase will force the user to log in.
  2. As an admin user:
     1. Display / modify and add products.
     2. Display all users on the system
  3. As a normal user:
     1. View / reprint own invoices.
     2. Select the product catalog to view products for purchase.
  4. View Cart with product name, qty, price and image.
  5. Add checkout button to the cart. Send an email to the logged in user.
  6. All data is stored in a Microsoft Sql database.

### Target Audience

Haefele Software hiring manager

### Name of project

Haefele E-Commerce

### Technical Specs

1. Server-Side Language: C#
2. Framework: .net Core 8.0
3. Unit test: xUnit
4. IDE: Visual Studio 2022
5. Frontend – Angular 18
6. UI CSS Toolkit: Bootstrap

Solution

The hf (Haefele) solution consists of several projects with the goal of providing a clean architecture. This makes testing and maintenance easier as different teams can work on different projects.

### Project 1 – hf.Api

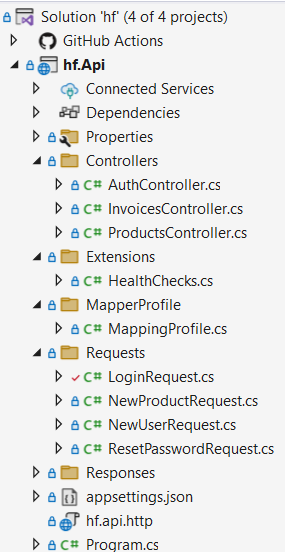


Fig. 1 - hf.Api

This project consists of the Program.cs file, the Auth, Products and Invoices Controllers. It also contains extensions to the Api project such the automapper profile file and a healthchecks class.

**Auth controller** - This controller serves as a way of adding and listing new users to the application. The login and reset-password endpoints are also contained in this controller.

The name, surname, token and user id fields are stored in the client’s localstorage. Please note that the password is hashed before saving to the database (Auth controller). Each subsequent login attempt will hash the password before comparing it to the user password stored in the db. This is simply to enhance security and does not replace client certificates. Jwt bearer token is used to authenticate subsequent requests.

**Products / Invoices controller** - These controllers are used to retrieve a list of products and invoices from the sql server datastore. Product and invoice crud operations can also be performed through these controllers.

### Project 2. hf.Application

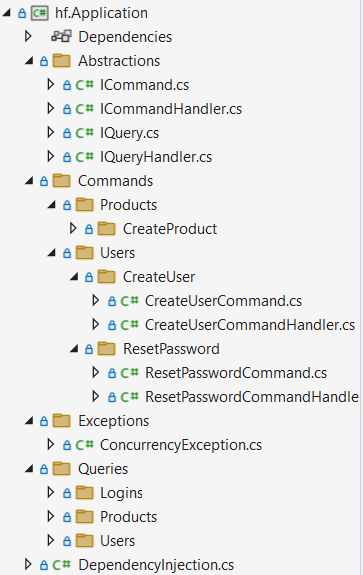


Fig 2. hf.Application

The Application project consists of four folders with subfolders and implements the cqrs pattern. The classes for implementing the commands are divided into the Products, Users and ResetPassword with more to follow. These folders contain the individual command and commandhandler classes for each user story. Queries are divided into the Logins, Products and Users folder which houses the query’s and queryhandler classes. Using the cqrs pattern we can separate the CRUD operations for better performance and separation of concerns as the application evolves over time. Both query and commandhandlers are wired up to the endpoints and can be called with the controller being location unaware.

### Project 3 – hf.Domain

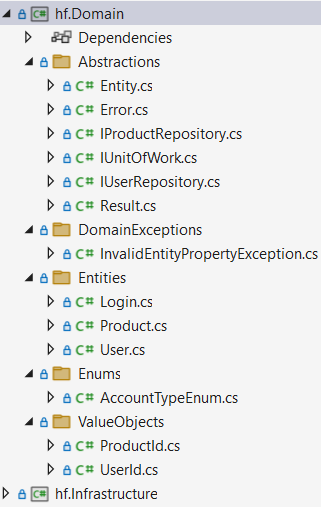


Fig 3. Hf.Domain

The Domain project is the center of the solution. It is important that it contains no references to 3rd party dependencies. All the other projects will keep a reference to the Domain project. It should only store contracts (interfaces) to services or repositories stores elsewhere as well as entities and value objects for persistence. This is also a great place to store domain exceptions.

### Project 4 - hf.Infrastructure

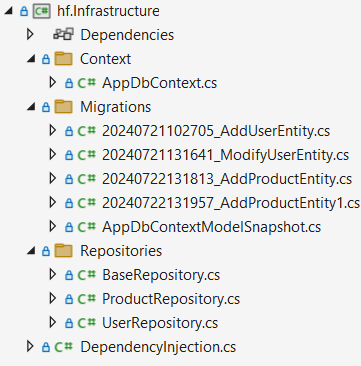


Fig 4. hf.Infrastructure

The purpose of this project is to persist and retrieve data from the database using entity framework core. The project also stores multiple repositories this injected into the different command and queryhandlers.

Fig 5 – Angular project structure.

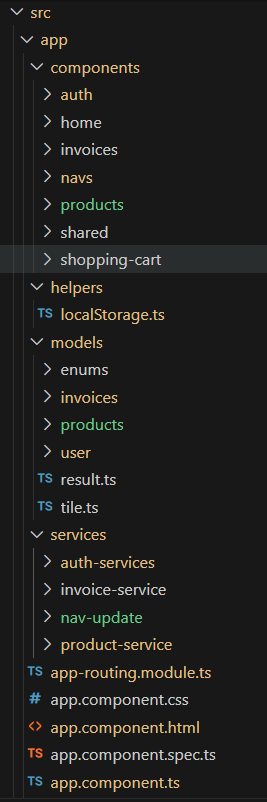


Fig 5.

As mentioned, the frontend is done in Angular 18. No specific version 18 commands were used so this should work in earlier versions of angular.

The Fig 5 shows the logical structure of the frontend –

1. Components: This houses the different components used in the project.
2. Services: there are 4 services in the Services folder that communicates with the hf.Api project by sending data in the form of parameters.
3. Models: contains the entities used in the project.

### Login

A screenshot of a login page

Description automatically generated

Fig 9. Login screen

Fig 9 shows the view that users can use to log into the application. Communication is asynchronous.

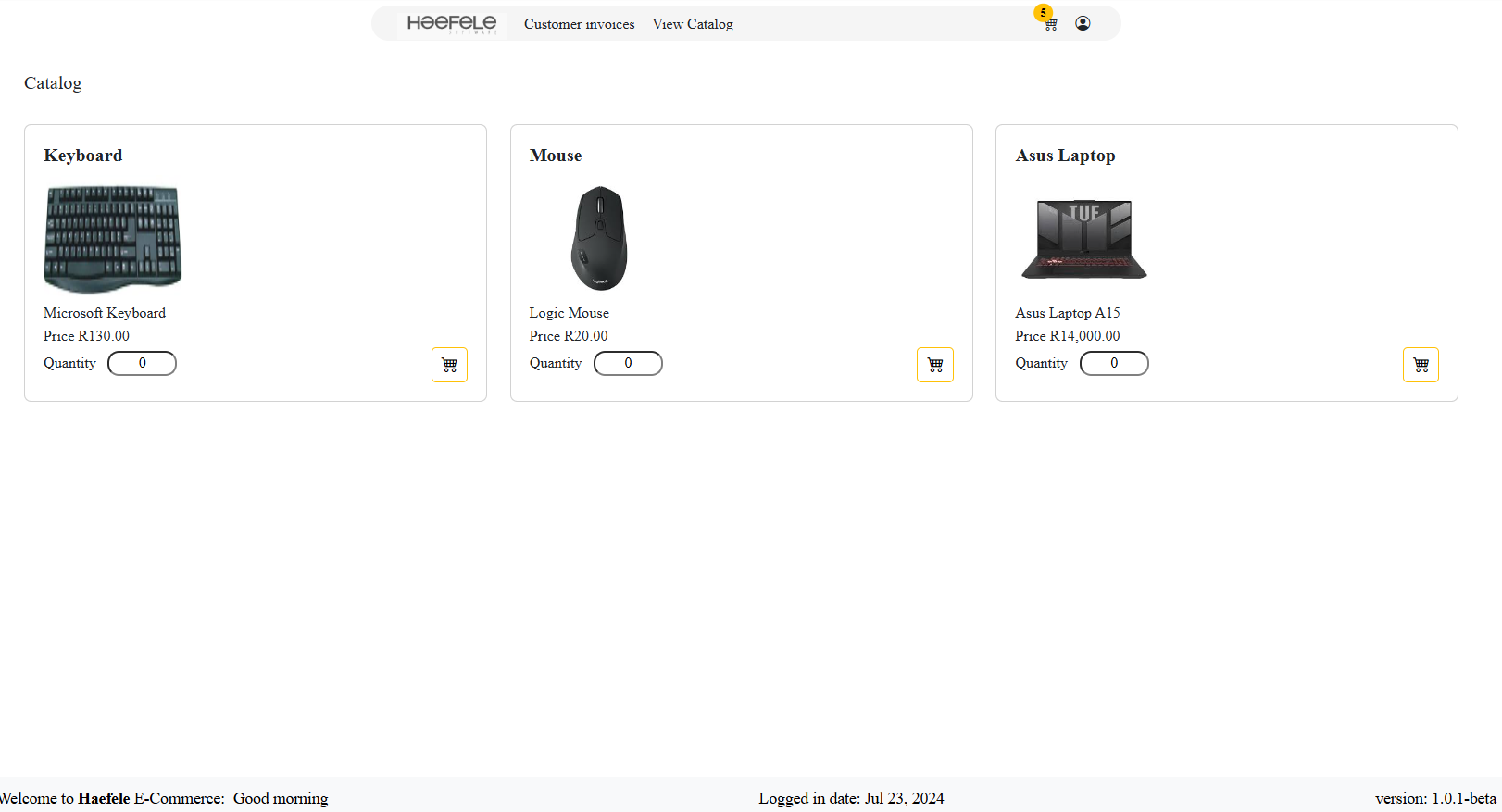


Fig 10. Product listing

After logging into the application, the user is taken to the product listing. As the requirements stated, a shopping cart button must be next to each product. Furthermore, the product is displayed with an id, Name, Image, Price and Qty.

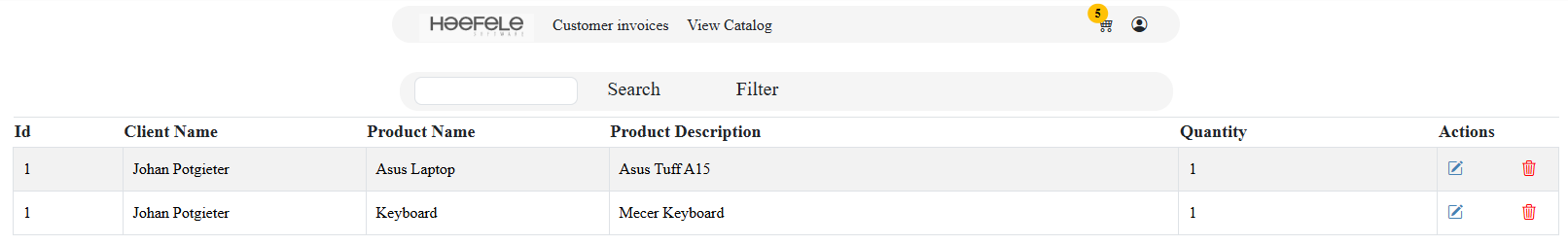


Fig 11.

Fig 11 displays how the ui changes as the user selects the different items.

Using Angular allows for dependency injection on the client side as well as an mvc dependency design pattern to logically divide the logic into separate modules. This use allows for the reuse of components.

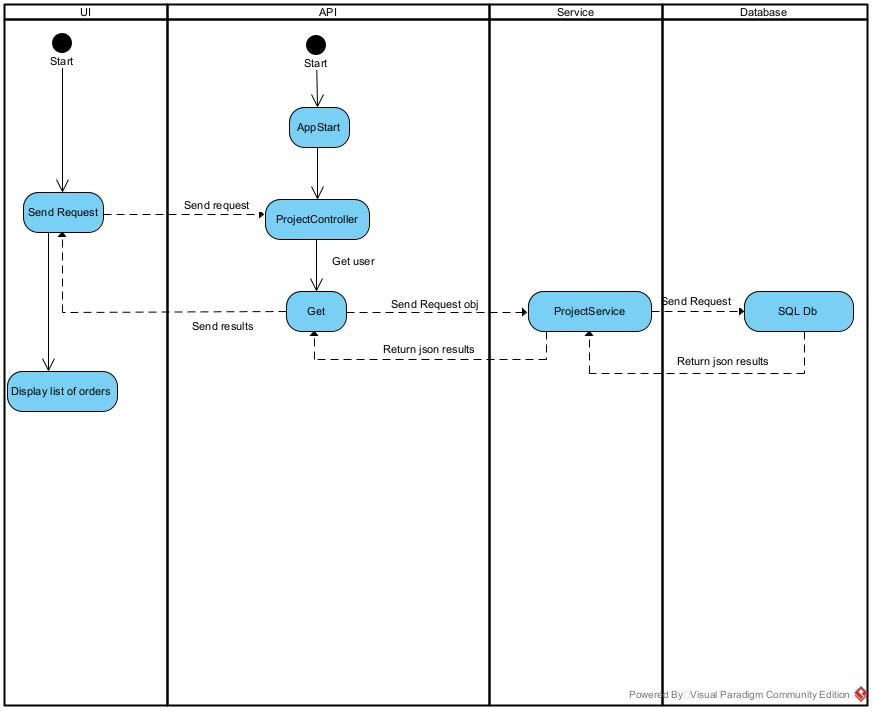


Fig 12. Login procedure

### User Stories

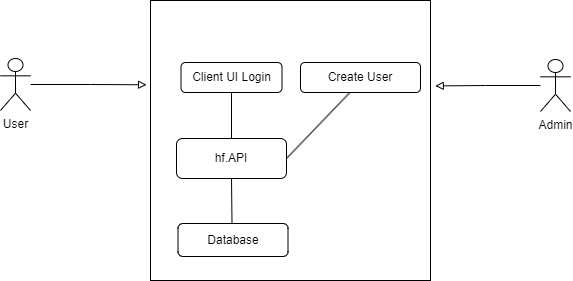


Fig 13. Log in user story

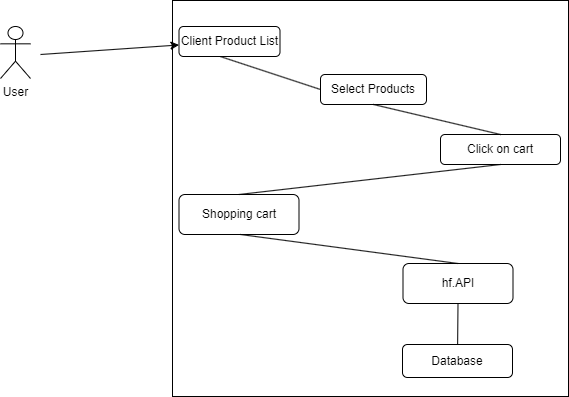


Fig 14. Select products

## Conclusion

The construction of the solution is done in a way that supports loose coupling and high cohesion. Please note that this solution architecture is not needed for every project. If the project is likely going to evolve over time and grow in size and complexity, separation of concerns and design patterns such as cqrs become more important for the solution maintenance. Below is a list of focus items:

1. Class, method and variable names are given meaningful values. The only time where x is used as variable is as a loop counter variable.
2. The controllers do not contain any logic. Global exception handers can be used to remove redundant code such as try catch.
3. Fields are declared with a leading \_ to indicate the role it plays and can be found at the top of each class.
4. Any injected dependency (services) is declared as read-only to keep it immutable.
5. Repositories and child classes are declared as sealed to prevent unneeded inheritance.
6. Where possible, repository, command and queryhandler classes are declared as internal.
7. Each project has its own dependency injection file which is declared as an extension class of the IServiceCollection. This allows the dependencies to be registered in each separate project.



The scope of the concrete implementations of services and repositories are declared as internal. Outside code cannot directly instantiate and call these repositories. Using this procedure, loose coupling is even more improved and the program.cs is kept clean. This also applies to the command and queryhandlers in the application layer. Each dependency injection class is registered in the api project.

1. Dividing the solution into separate projects makes the application more testable and maintainable.
2. Entity classes are immutable as each property cannot be directly changed from outside the class. Initial instantiation is done by using the constructor. Any modification thereafter uses modification methods.
3. Please note that all requests are handled asynchronously using tasks.
4. Communication between controller endpoints and repositories are handled through the cqrs pattern.

A diagram of a database

Description automatically generated

1. Frontend routes are protected by auth guards and JWT token which will force a login procedure if an anonymous user clicks on a menu item which is only meant for authenticated users. If a login is successful, a jwt token will be attached to each subsequent request to the backend. Note that the token will expire after set period of time.

# Improvements

Below is a list of improvements that still needs to be added:

1. Observability and monitoring. Tools to monitor and graph the application’s performance and logfiles need to be added to the application such Prometheus and Grafana.
2. Better unit test and integration test coverage.
3. Better ci/cd pipelines needed. The ones currently included are basic and are there to show the intent on a better devops implementation.
4. Make better use of entityframework core. Can use batching when more invoices and customers are loaded on the system.
5. As the front-end application becomes more complex and grows, lazy loading can be implemented. This allows the application functionality to be divided into different modules. Each feature module is only loaded into memory when accessed. At the moment, we only have one main module that will load all the components into memory.
6. For database connection settings in the hf.Api, use user secrets to obfuscate the database credentials. Currently the credentials are visible in the appsettings.json file for simplicity but it poses a security risk.
7. Supply Jwt signingkey through environment variables.

How to run the solution

1. In the hf.backend solution, open the package manager and. Change the project to hf.Infrastructure. If the migration folder exists, delete it. Run the command ‘add-migration’ + plus migration name. This will build the project and create migrations. After build success – run update-database. This should create the database. Alternatively, run the attached script in the docs folder. Use bonzibuddy as username and password as password.
2. Start the api solution. The default url is <http://localhost:5000> and <https://localhost:5001> for https.
3. After opening the client solution – run the npm install command. This will install all the dependencies needed.
4. Run the front end with ng serve. The default url is <http://localhost:4200>.
5. Testing can also be performed with the use of swagger. Go to url <https://localhost:5001/swagger/index.html>.