# BETS E-Commerce

# June 06, 2022

### Business Requirements

1. Requirements for the BETS api are as follows:
   1. Add user to the system. Provide email and password as credentials.
   2. Allow user to log in.
   3. Display a list of products.
   4. Add cart button next to each product.
   5. View Cart with product name, qty, price and image.
   6. Add checkout button to the cart. Send an email to the logged in user.
   7. Store data in sql database.

### Target Audience

BETS hiring manager

### Name of project

BETS E-Commerce

### Technical Specs

1. Server-Side Language: C#
2. Framework: .net Core 5.0
3. Unit test: xUnit
4. IDE: Visual Studio 2019
5. Frontend – Angular 12
6. UI CSS Toolkit: Bootstrap / PrimeNg

Solution

The BETS 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 – BE.Api

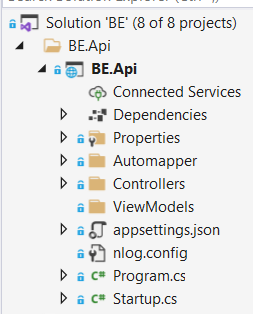


Fig. 1 - BE.Api

This project consists of the Startup.cs file and the Auth, Login, Products and ShoppingCart Controllers.

**Auth controller** - This controller serves as a way of adding a new user to the application.

**Products controller** - This controller is used to retrieve a list of products from the sql server. This list is then displayed on the client web page.

**Login controller** – allows a user to log into the application. The name, surname, token and user id fields are stored in 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.

**Shopping Cart Controller –** is used to receive the shopping cart information from the client. The details are then stored in the database.

### Project 3. BE.Contracts

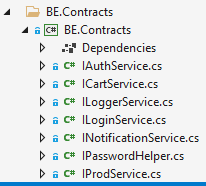


Fig 2. BE.Contracts

Four contracts are used in this project.

1. IAuthService is implemented as a service to add users to the application. See the BE.Services project.
2. IProdService is implemented as a service to query and list products.
3. ILoggerService is used to implement the NLog logger package.
4. ICartService is implemented as a service to save cart details to the database. An email is also sent to the logged in user.
5. ILoginService is implemented as a service to allow a user to be logged into the app. Some details are stored on the client side for customization.
6. IPasswordHelper is implemented as a helper service with the only function to hash the user password. The concrete implemented can be easily swopped out for another implementation.
7. INotificationService implemented as a helper service with the only function to send an email with the cart details to the customers. The concrete implementation (emailservice) can be changed to use any other notification medium such as whatapp or sms’es without changing much of the overall logic.

These interfaces are injected through dependency injection and makes it easier to perform testing. Implementation can be easily swopped out just by defining another concrete implementation in the DI container.

### Project 3 - BE.IoC

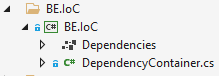


Fig 3. BE.IoC

The Di project registers all the interfaces for dependency injection. This function is normally done inside the Startup.cs class but externalizing this into a separate project, it keeps the Startup.cs cleaner.

### Project 4 - BE.Services

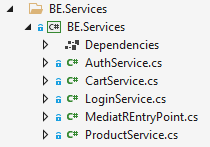


Fig 4. BE.Service

The service project is used to separate the different functions into separate logical components. The service classes implement the corresponding interfaces in BE.Contracts project which is then injected into the controllers.

### Project 5 - BE.Test

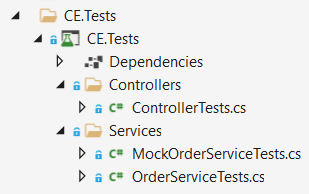


Fig 5. BE.Test

The last project is an Xunit test project for running unit tests. The application is developed using a TDD approach.

### Project 6 – BE.Infrastructure

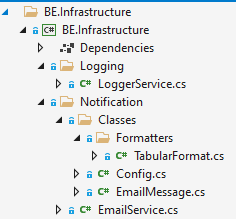


Fig 6 – BE.Infrastructure

This project includes all the cross-cutting concerns that are shared across the solution. As noted previously, the LoggerService is implemented by using NLog but can be changed to use a custom solution. The email service is used by the CartService to send payment confirmation email to the user but can also be used from any other component for a different purpose.

### Project 7 – BE.Common

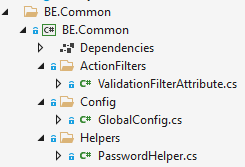


Fig 7. BE.Common

BE.Common contains different functionality that can also be used throughout the solution such as JWT configuration.

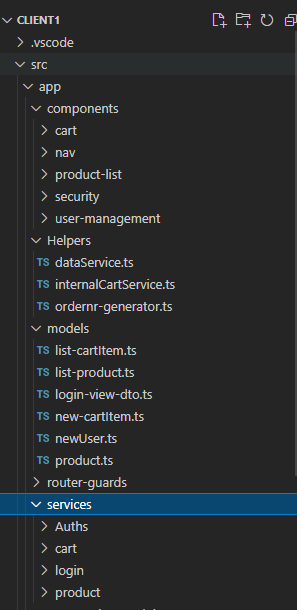


Fig 8 – Angular project structure.

As mentioned, the frontend is done in Angular 12. No specific version 12 commands were used so this should work in earlier versions of angular.

The Fig 8 shows the logical structure of the frontend –

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

### Login

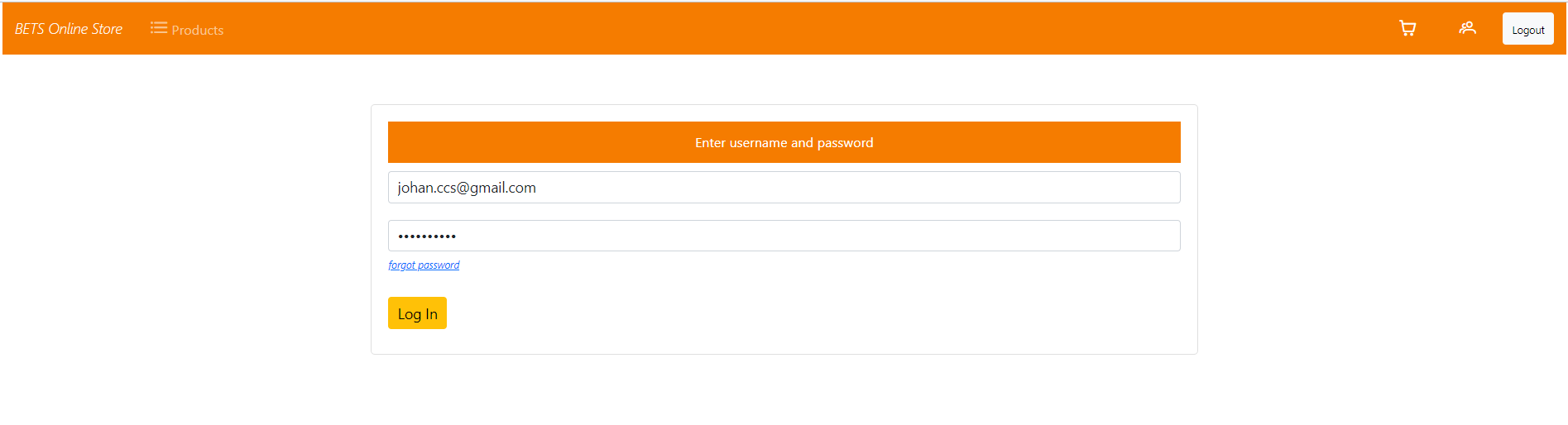


Fig 9. Login

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

Graphical user interface

Description automatically generated

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.

Graphical user interface

Description automatically generated

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.

Graphical user interface, application

Description automatically generated

Fig 12. Shopping Cart

After the user selects the items to purchase, the app will route to the shopping cart when the user clicks on the Checkout button

Graphical user interface, application

Description automatically generated with medium confidence

Fig 11. Email sent to customer

Fig 11 shows the output of all the orders with a status of IN\_PROGRESS. In the console application, I used the injected IOrderService interface to show the use of a different pattern. The logic is still reused from the Api implementation.

Two new classes (DisplayFormattedData.cs and DisplayFormattedProductData.cs) are added to abstract the display logic away from the Runtime class.

Diagram

Description automatically generated

Fig 13. Get orders activity diagram

## Conclusion

The construction of the solution is hopefully done in a way that supports loose coupling and high cohesion.

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. Filters 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. Dividing the solution into separate projects, make the application more testable and maintainable.
6. Please note that all requests are handled asynchronously.
7. The solution is based on TDD.

To get the solution running:

1. Start the api solution. The default url is <http://localhost:5000> and <https://localhost:5001> for https.
2. After opening the angular solution – run the npm install command. This will install all the dependencies needed.
3. Run the front end with ng serve. The default url is <http://localhost:4200>.
4. Testing can also be performed with the use of swagger. Go to url <https://localhost:5001/swagger/index.html>
5. Alternatively – import the the swagger.json (included) into postman. This will setup all the existing endpoints in postman.