Crișan Alexandra-Bianca
gr. 258/1

Wishlistly application – Documentation

1. *Purpose of the application*

Wishlistly is an app that allows users to track items in their wishlist – create, edit and delete certain items. After a user registers, an empty wishlist is automatically created for them and they can start editing it.

2. *Web server*

a. .NET 5

The application was created using an existing template in Visual Studio, for ASP .NET Core and React.js. For the server part, .NET 5 was used.

b. CQRS pattern of Microservices

As a pattern of Microservices, Command Query Responsibility Segregator (CQRS) was used to implement the desired functionality. This means the application is divided into two parts: **Commands**, that handle updates of the data and **Queries**, that read the data. By doing this split, we can use different and simpler models for reading and writing data, and we can prevent update commands from causing possible conflicts.

Some examples of implemented Commands and Queries can be seen in the next images:

Figure 1 - Commands

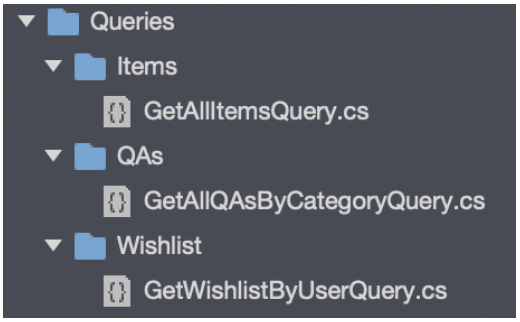

Figure 2 – Queries

c. Secure REST services

The web server exposes a RESTful API through its controllers that use the well-known HTTP verbs and HTTP protocol.

The ASP .NET Web Api provides a built-in authorization filter in the form of an attribute, [**Authorize**], which is placed in every controller to restrict api access to authenticated users (see Fig. 1). These controllers must derive from ControllerBase, a class that provides us with multiple useful properties and methods, such as the [**ApiController**] attribute. This attribute, on one hand automatically triggers an HTTP 400 response if any model validation does not pass, and on the other hand requires explicit routing using the [**Route**] attribute.

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class ItemsController : ControllerBase
{
    private readonly IItemsService _itemsService;

    public ItemsController(IItemsService itemsService)
    {
        _itemsService = itemsService;
    }

    [HttpGet]
    public async Task<ActionResult> GetAllAsync()
    {
        var items = await _itemsService.GetAllAsync();
        return Ok(items);
    }

    [HttpPost]
    public async Task<ActionResult> AddItemAsync(ItemViewModel itemViewModel)
    {
        await _itemsService.AddItemAsync(itemViewModel);
        return NoContent();
    }
```

Figure 1 – Authorized REST controller

Besides that, the app uses the **ASP .NET Core Identity** framework that manages and stores user accounts with the use of an Entity Framework Core data model. Users, passwords, profile data, roles, claims, tokens, everything is taken care of, creating a secure web application.

d. Server side notifications

To achieve server side notifications, **SignalR** was used, which is a Microsoft ASP .NET library that allows real-time web functionality (the ability to have server-side code push content to the connected clients as it happens, in real time).

It is very easy to install using the Package Manager in Visual Studio, with just a command we can already use the package called Microsoft.AspNet.SignalR.Core.

As for the implementation, SignalR uses hubs to connect to an api with the client web app so we just need to define a method that will be used by a client and the library will do the rest. This code will send a message to all the clients that are listening to the event called "ReceiveMessage".

```
public class MessageHub : Hub
{
    public async Task SendMessage(string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", message);
    }
}
```

Figure 2 - SignalR Message Hub

In this case, the server sends a notification to the client if it detects that the Wishlist is already opened somewhere else, so the user knows.

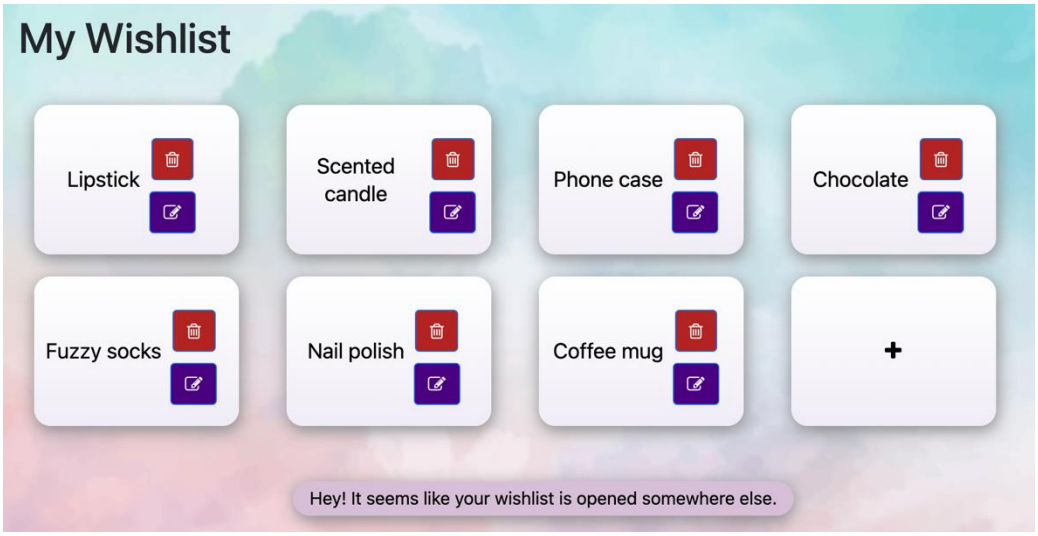

Figure 3 - Notifications

It is worth mentioning that SignalR and the hub need to be registered in the Startup class.

3. *Web app*

a. React.js

As mentioned before, the app was created using the Visual Studio template for .NET and React.

b. Microfrontends

The frontend part of the app was split into 2 applications with the purpose of them being more independent, not only easier to understand but also to develop and test. It was divided by the features of the app, which are **Wishlist** and **Help**. Therefore there is one micro-frontend that deals with a user's wishlist, and another one that presents the Help section containing FAQ's.

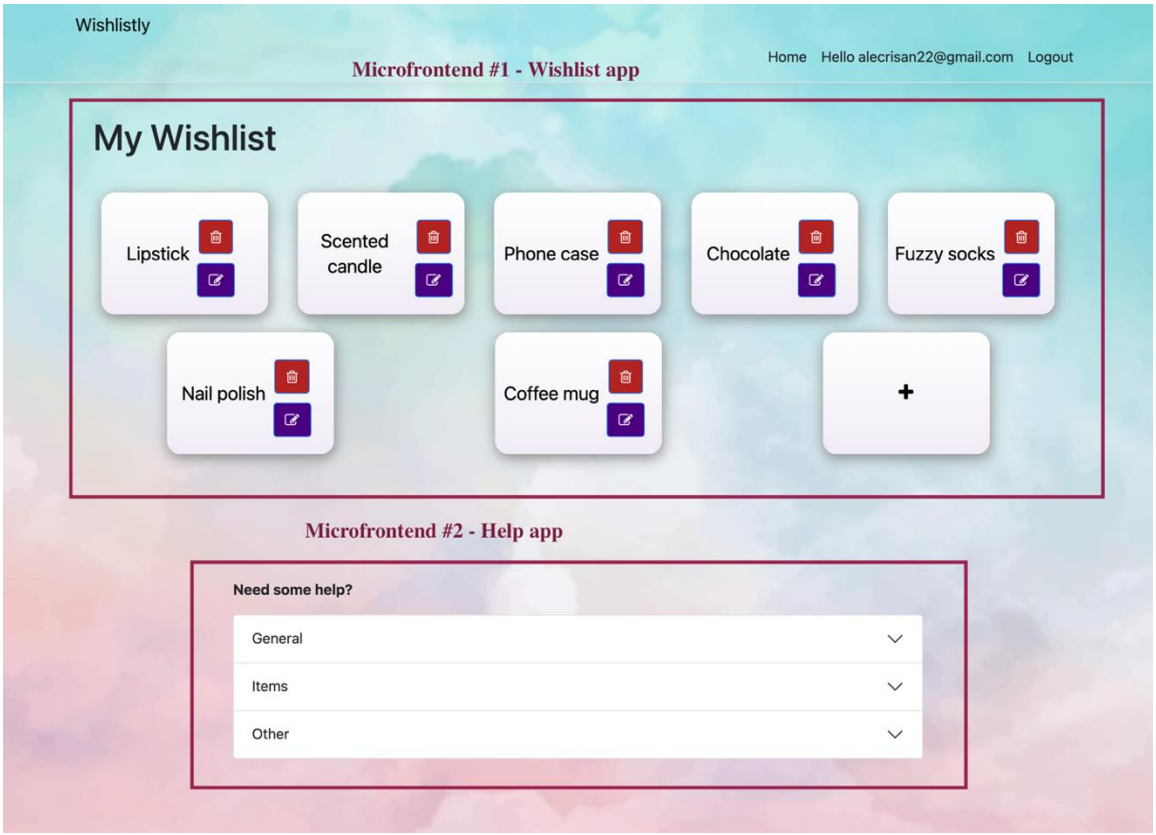

Figure 4 - Microfrontends

c. Webpack module federation

In order to implement the microfrontends, webpack 5's module federation was used, a JavaScript architecture that allows sharing the code and dependencies between 2 applications.

Note: The main client app configuration was auto-generated by the Visual Studio template that uses the create-react-app script, which unfortunately does not support Webpack 5 yet. However, webpack 5 was manually installed to be able to add module federation.

The webpack.config.js file needs some extra configuration besides the simple webpack part of it, using a ModuleFederationPlugin function as such:

Figure 5 - Webpack config Wishlist microfrontend

Figure 6 - Webpack config Help microfrontend

Important elements

Name: we will communicate with other apps through this name

Filename: we use it as an entry file

Shared: we use it to specify which dependencies will be shared with other apps (! if we don't type "singleton: true" each app will run on a separate React instance)

Remotes: determines from which apps we will receive a component/page or the entire app – in this case, from the Help application

Exposes: allows us to share a component/page/entire app – in this case, the Help component

d. Notifications

For the client part of the notifications the **SignalR JavaScript library** was used so that it can call the server-side hub code. Again, this was also easy to install since it is a npm package so the only command needed was "npm install @microsoft/signalr".

As I mentioned before, after the client establishes the connection, it listens to the event "ReceiveMessage" from the server-side and shows whatever message it gets.

```
connection.on("ReceiveMessage", (message) => {
  const li = document.createElement("li");
  li.textContent = `${message}`;
  li.className = 'notification';
  document.getElementById("messageList").appendChild(li);
});
```

Figure 7 - Receiving messages from the hub

4. *Integration*

The server-side notifications implementation with SignalR makes use of an integration pattern called **Publish Subscribe Pattern**. This means that the clients subscribe to an event and the server broadcasts a message if the event occurs to the clients who subscribed to it. So esentially SignalR allows the server to send messages to clients without them needing to request the messages.

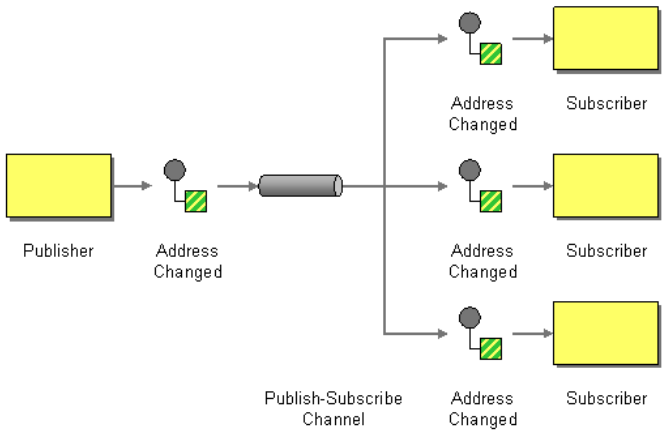

Figure 8 - Publish Subscribe Pattern

5. *Containers to deploy the solution*

Docker is one of the most popular containerization platforms, allowing developers to isolate their apps from their environment into containers that bundle software, libraries and everything needed to run successfully. It basically takes away the "it works on my machine" issue. ☺

So naturally, this application was deployed using Docker and Docker-Compose. The server and both of the microfrontends each have their own Dockerfile and can be deployed separately but also together using Docker-Compose. In the root of the project there is a *docker-compose.yml* file that consists of all the services and the paths to their dockerfiles so that the system can be started altogether.

Before running the apps in containers, images were built using the command "*docker image build -t <name-of-image> .*"

To run each container separately, there is a README file with the instructions for each one.

To start the whole system, go into the root of the app which has the docker-compose file and run *"docker-compose up"*. The app will we available at https://localhost:5001.

6. *UML, c4 models describing the system*

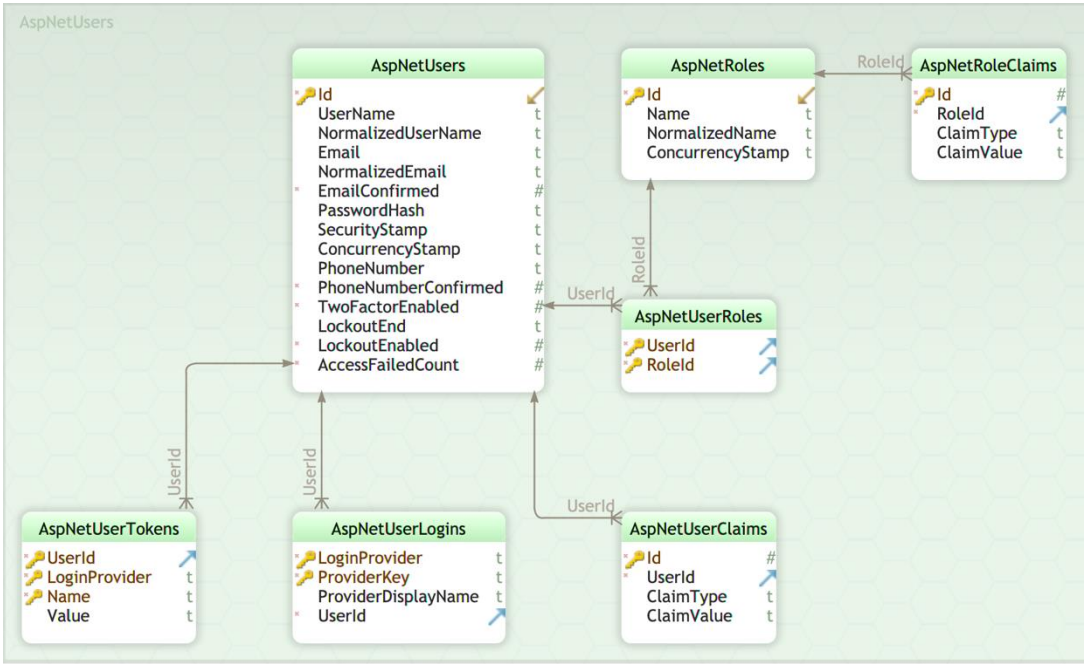

Figure 10 - DB Diagram part 1 - the tables generated by Identity

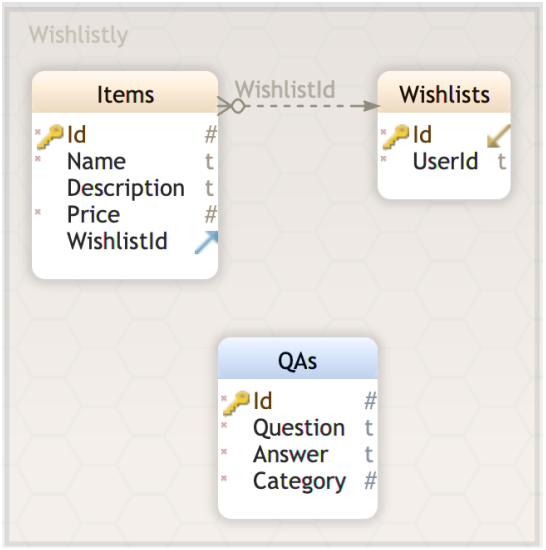

Figure 9 - DB Diagram part 2

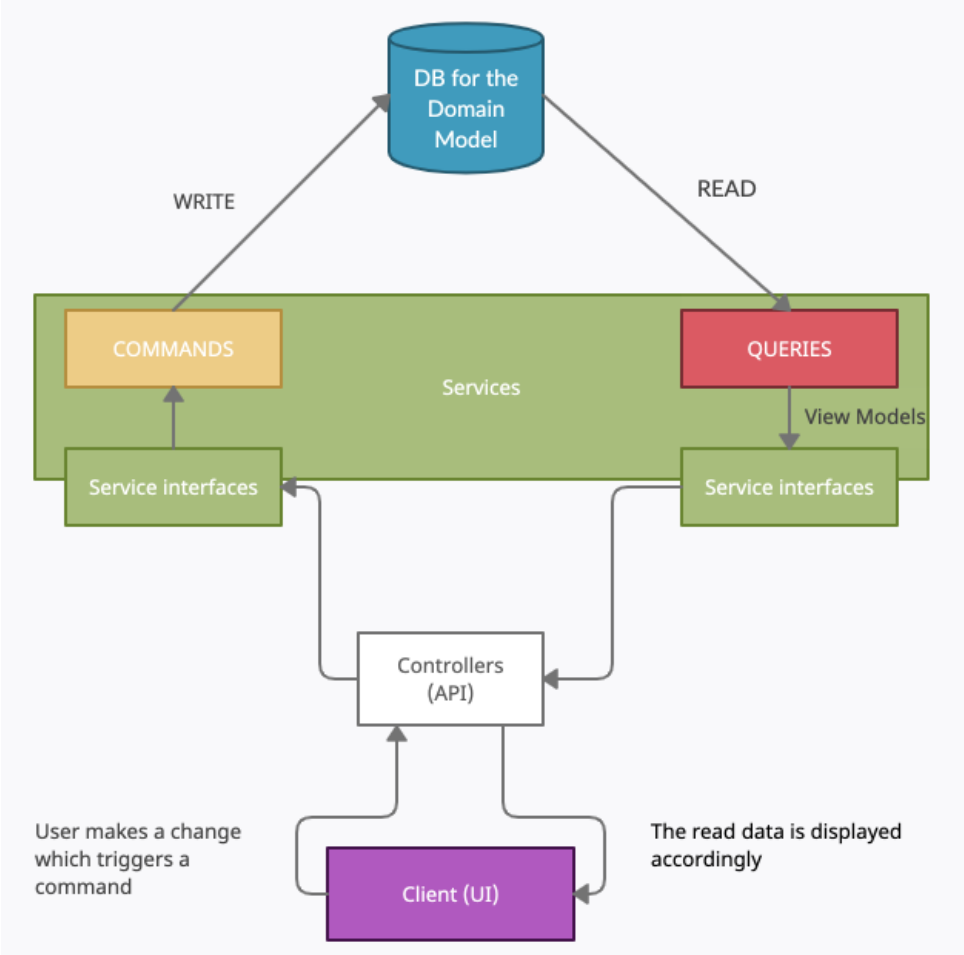

Figure 12 - CQRS architecture

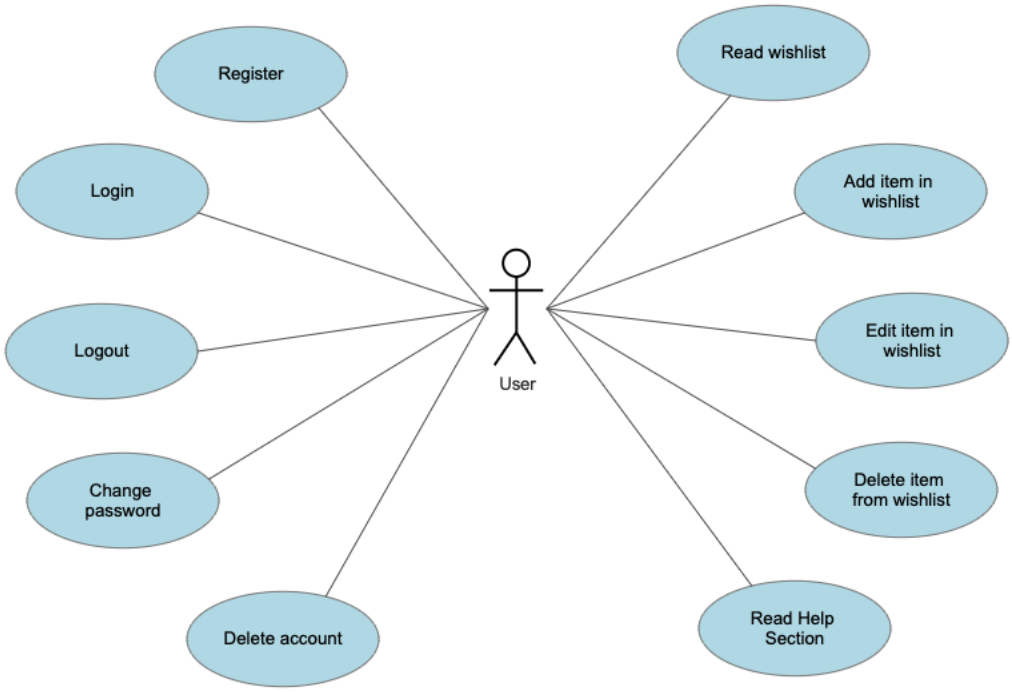

Figure 11 - Use case diagram

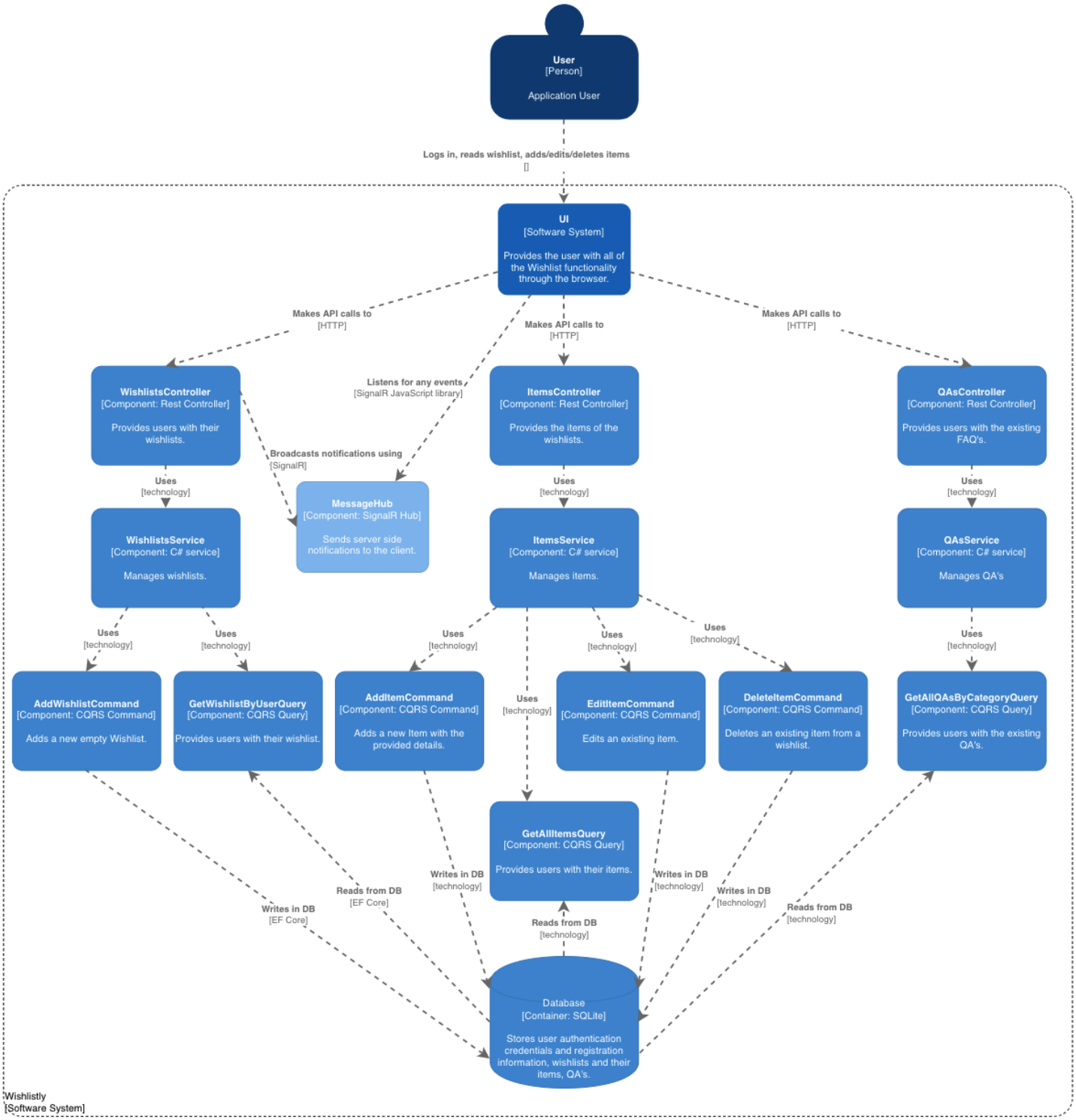

Figure 13 - C4 model of the architecture

7. *Other tools used that weren't mentioned*

Github – a git system was used to keep track of the changes and organize the application. Repository: https://github.com/alecrisan/Wishlist

Entity Framework Core – an object-relational mapper (ORM) that performs the work required to map between objects defined through code and data stored in relational data sources.

Database Management System (DBMS) – SQLite

When working with EF Core, the most popular choice for a DBMS is SQL Server but due to the fact that this application was developed on a MacOS, a more cross-platform tool was needed, thus the choice of SQLite. This library offers a great range of advantages, such as: it is serverless, zero configuration, self contained and transactional. However, it still has some limitations, for example the ALTER TABLE command is not yet implemented.
As a tool for designing and editing the database file, **DB Browser for SQLite** was used.

Visual Paradigm, Creately and Smart Draw for the diagrams