CS 319 - Object Oriented Software Engineering

Instructor: Eray Tüzün, TA: Muhammad Umair Ahmed & Elgun Jabrayilzade

BilHealth

Design Report

Iteration 2

Mehmet Alper Çetin
21902324

Vedat Eren Arıcan
22002643

Uygar Onat Erol
21901908

Recep Uysal
21803637

Efe Erkan
21902248

May 2, 2022

Contents

1 Introduction

Our project is a health center management software, built in the form a web application. This report documents the design choices that were made for the implementation of the project.

Note that all diagrams on this document are in vector format, meaning that you can zoom in without any decrease in quality.

1.1 Purpose of the System

The project is designed as a web-based application that will ease the health center management in Bilkent. The main goal of the project is to provide online attention to the patients and to have better communication betweeen doctors, staff and patients. Our system includes features that ease the interaction betweeen patients and doctors through cases which contain all relevant information for a given medical situation. Patients can open cases and request for appointments through that opened cases. Also, staff and doctors provide patients with medical services through cases.

1.2 Design Goals

1.2.1 Accessibility

The system should be accessible by all actors despite individual shortcomings, to achieve its goal. We want to design a user interface such that a person who isn't familiar with using computers can easily find their way around the website. All the buttons and labels in the system should have self-explanatory titles. The system should include boundary conditions which prevent the user from taking wrongful actions.

As a general guideline to practices that make up an accessible user interface, the Mozilla Developer Network and World Wide Web Consortium (W3C) are useful.

1.2.2 Security

The system will have a person's health details which are highly privacy sensitive. Also, the system will have confidential information of every user such as Bilkent ID, email address and password. Therefore, the system should be secure to protect all of these personal information from any unauthorized activity. No user would want such private data to be accessed by any unauthorized person.

To secure the app, the implementation should follow an access control schema as such:

- A doctor can access a patient's full profile only while the doctor is assigned to an open case with the patient.
- A nurse can access a patient's full profile only while granted permission by the patient or a staff member. This access should expire after some amount of time.
- A staff member can always access a patient's full profile. However, this access is fully tracked, also known as an *audit trail*.

From research, this schema appears to align with health information security practices around the globe.

1.2.3 Performance

The system should give the best performance even on computers with old hardware. Every user interaction should be processed and displayed in less than 1 second to provide a better experience to the user.

In order to ensure sufficient performance, potential bottleneck points should be addressed. One of the most notorious bottlenecks is database access. The system should limit round trips to the database as much as possible. When data is queried or written, it should be done with transactions that allow all of the process to be completed on a single request.

1.2.4 Scalability

The system aims to support up to 20 thousand users, which means the design should enable concurrency and high throughput.

1.2.5 Deployment

In order to facilitate the development of the project in a fast paced environment, the design takes into consideration concepts such as portability and replicability. The sought result is that developers can easily run a local version of the system while keeping things simple when the time comes to deploy to production.

1.2.6 Testability

Almost any respectable web service of today strives to achieve test driven development of some magnitude. Our project is no different and the design attempts to make it possible to incorporate unit or integration tests for the business logic, by means such as dependency injection.

1.2.7 Extensibility and Maintainability

The fast paced development environment can quickly turn the project into a nightmare of spaghetti code and architecture. To counteract such a possibility, the design aims to keep components maintainable and extensible through proven OOP structs.

1.3 Top Two Design Goals

For the implementation of the health center management software, two primary design goals were chosen: **Security** and **Scalability**.

Due to the fact that the system contains sensitive health information about the users, such as diagnosis and test results, the system should be secure to prevent unauthorized access. In addition to this, data should be stored in a secure way in order to prevent possible identity thefts. Therefore, **security** is chosen to be one of the top design goals.

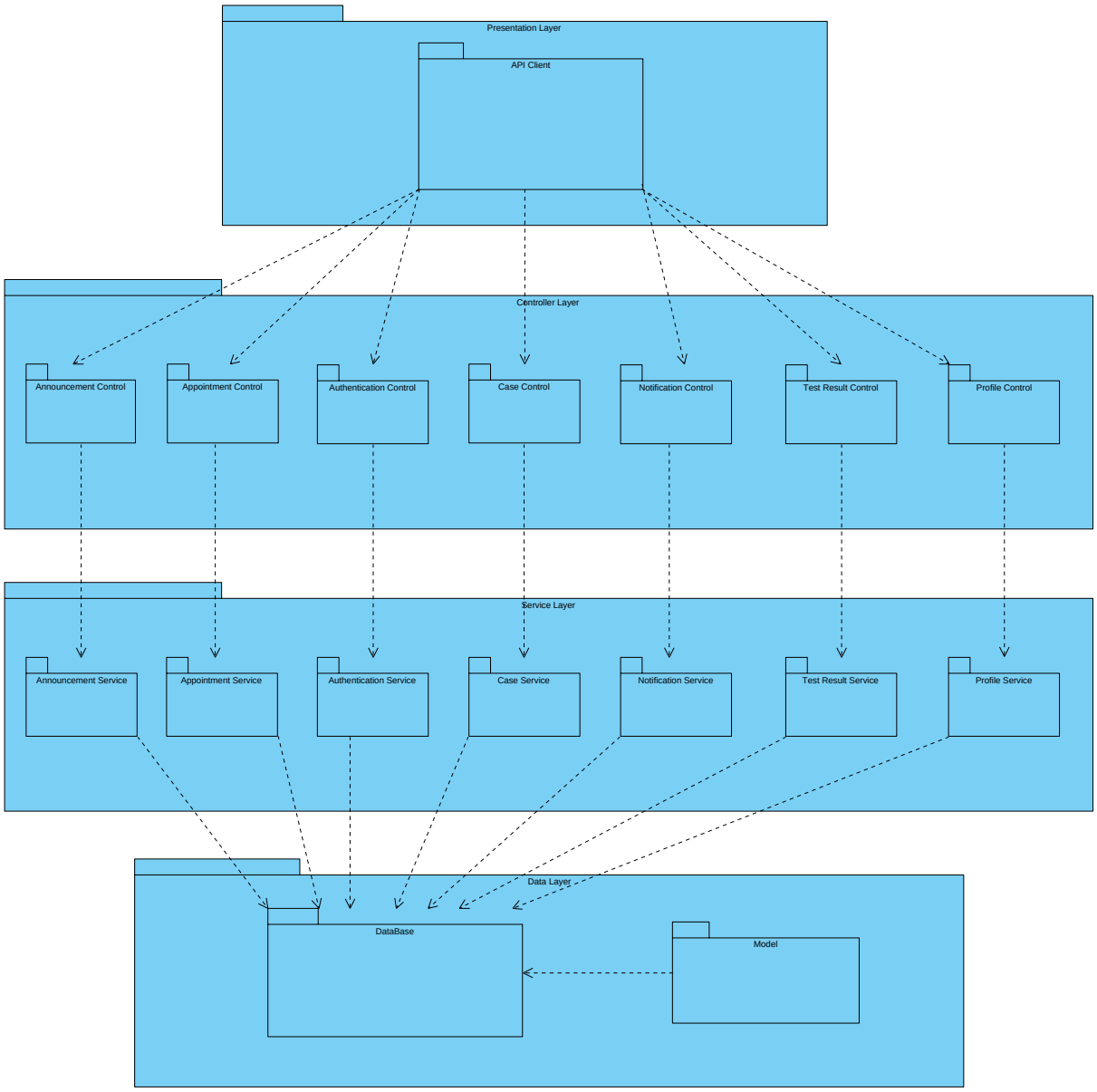
The health center management system may have up to 20.000 end users, and these users may execute operations on the system concurrently. Therefore, in order to prevent any issues, scalability is chosen to be one of the top design goals.

2 High-Level Software Architecture

2.1 Subsystem Decomposition

Since the project uses React as its front-end interface technology, the subsystems are not divided according to views. The front-end client, `APIClient`, handles all communication with the API exposed by the *controller layer*. To elaborate, this client component sends HTTP requests to the back-end endpoints according to the user's interactions with React.

The service layer is called by the controller layer to execute the business logic and database access required to perform a given action. The data layer consists of the domain models, powered by the Entity Framework Core library.



2.2 Hardware-Software Mapping

The project does not have any extraordinary hardware requirements. As it is a web application, it requires the users to have a device using which they can connect to the web server of the application over the internet. The main target devices for a supported user experience are desktop computers, laptops, and mobile phones.

In order to run the project the intended way, the host system's hardware must support stable containerization through Docker. It is possible to reconfigure certain entrypoints of the project to support bare-metal execution too, at the cost of system replicability and practicality. Similar to most other web services, the project is officially supported to run on a Linux environment, which may be of consideration when choosing hardware.

The server on which the system runs should have the following specifications at a minimum:

- A server-grade CPU such as Intel Xeon, with clockrates above 2.0 GHz
- 16-32 GB of RAM to support many users and interactions at once
- 1-2 TB of storage to support the database and the test result store

2.3 Persistent Data Management

The project makes use of PostgreSQL as its persistent data storage solution. The project does not have specific requirements that directly justify the use of PostgreSQL, however, this database system is well rounded enough to save the development process from any potential drawbacks. The `psql` command line tool is easy to use and supports our workflow in implementing database migrations. Also, the Docker images for the database are robust and simple, which has allowed the project to be supported by a containerized database.

On top of the actual database, the project uses EF Core as an object relational mapper (ORM), which is Microsoft's official library for data persistence. We use the recommended *code-first* approach of database management, which means our PostgreSQL tables are constructed by EF Core with respect to our entity definitions in C# code. The EF Core toolchain generates migrations, which it also converts to SQL scripts, using which a database can be configured to host the project in moments.

All of the domain models seen on the class diagrams are persisted to the database.

2.4 Access Control and Security

The project is secured by a typical authentication and authorization flow, supported by our identity provider of choice, ASP.NET Core Identity. Authentication is conducted over HTTPS with a user's username and password. The controller endpoints of the project are exposed to users within specific role types, which constitutes our authorization process. The authentication is persisted on the user's device in the form of browser cookies, enabling user session capability.

The project is mostly exclusive to authenticated users, meaning that there are not many substantial actions a guest user can take. Moreover, since users are divided across roughly

5 roles, most HTTP endpoints concerning management of the system are forbidden to access by non-staff users.

Lastly, user details are stored in a database with their passwords having been hashed by the *bcrypt* algorithm, which is among the top choices in secure password storage techniques at this time.

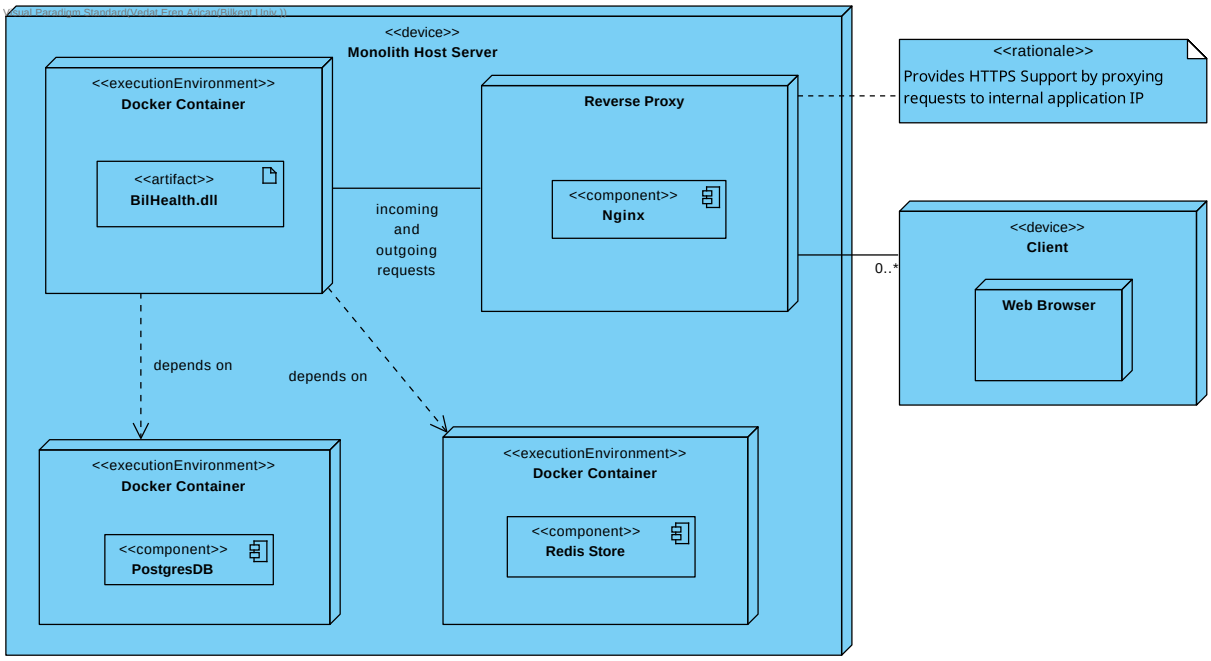
Table 1: Access Matrix

Object \ Actor	Patient	Nurse	Doctor	Staff	Admin
Announcements	Read	Read	Read/Write	Read/Write	All
Appointments	Request	-	Deny	Approve/Deny	All
Authentication	Login	Login	Login	Login/Register	All
Cases	Open/Message	-	Close/Message	Open/Close	All
Appointment Visits	-	Add	-	-	All
Triage Requests	Request	Request	-	Approve/Deny	All
Profiles	Read/Write Own	Read Whitelisted	Read Associated	Read/Write All	All
Test Results	Read Own	Read Whitelisted	Read Associated	Read/Write All	All

2.5 Boundary Conditions

2.5.1 Initialization

To initialize the application, the admins need to first perform schema migrations on the database. Once the database is ready, the rest of the system can be initialized through the `docker-compose` tool.



2.5.2 Termination

The application is terminated through the `docker-compose` tool. There are no special steps to be taken beyond commanding Docker to bring down the application containers.

2.5.3 Failure

The application does not crash in the event of failure. The implementation is careful to catch exceptions in appropiate places, and log the incidents if need be. Furthermore, the ASP.NET Core framework catches and reports any rare exceptions that may go unnoticed.

Also, database access should be conducted through *transactions*, which allow multiple operations to be made dependent on each other such that if one operation fails, the rest are not executed. This is ideal for data reliability and integrity in the event of failure.

3 Low-Level Design

3.1 Object Design Trade-Offs

- **Maintainability** versus **Performance:** The system will have objects for almost every part of the project because it is coded using OOP. This may cause a decrease in performance, but it is good for maintainability as it provides developers the mechanism to make changes easily.

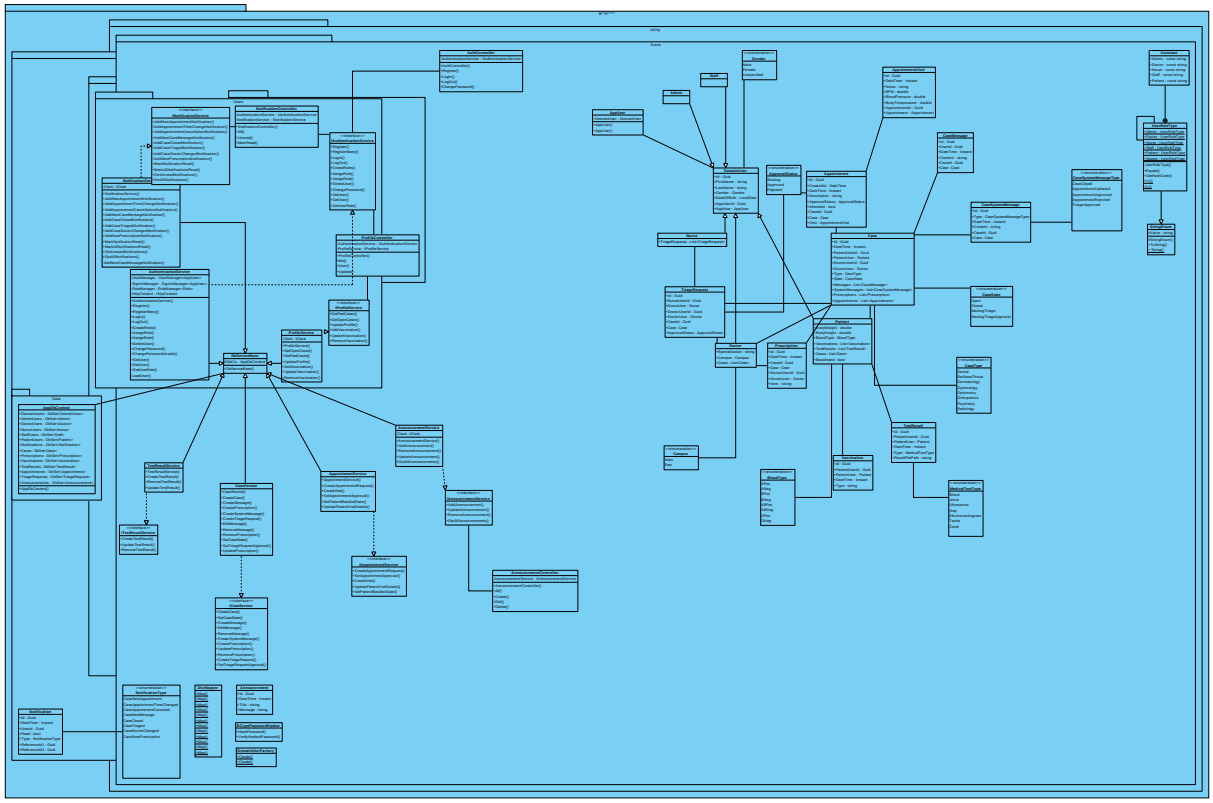
- **Memory** versus **Performance:** The system will be created using OOP, so there will be many objects and classes involved during the execution of a given action on the application. This may cause some diminish in performance as memory is used by more objects and a bottleneck occurs.

- **Security** versus **Accessibility:** The system will not have security measures beyond a typical username and password. This decreases the security of the authentication

process, but is better for accessibility purposes as users will not require extraordinary steps to access the system.

- **Functionality** versus **Extensibility:** As the system gains more functionality, despite the use of appropiate design patterns and practices, interfaces may become more rigid. This may cause the application to be less extensible.

3.2 Final Object Design



3.3 Packages

Below is a list of our namespaces with brief descriptions. Note that, to preserve the anonimity of the report, we've replaced the root namespace with a placeholder "`Project`".

- `Project.Utility`:
  This namespace contains the utility functions which were able to be decoupled from the project at large. In doing so, it can be used reliably across other namespaces.

- `Project.Utility.Enum`:
  This namespace contains the *enum* and *enum class* definitions that are used in many parts of the project.

- `Project.Services`:
  This namespace contains the main business logic interface through which boundary and entity systems are connected.

- `Project.Services.Users`:

This namespace carries the same responsibility as its parent, but specifically for user-related entities.

- `Project.Model`:
  This namespace contains the entities that are persisted into a database. The main units of data are located here.

- `Project.Model.Identity`:
  This namespace contains entities that are specifically user information holding objects.

- `Project.Model.Dto`:
  This namespace contains *record* objects that are used to transfer data to and from the front-facing controllers. It is also used for some internal communication of unpersisted data. In other words, these are temporary representations of the actual data entities to be persisted.

- `Project.Data`:
  This namespace contains the database context built from EF Core's repository-like architecture.

- `Project.Controllers`:
  This namespace contains the front-facing API endpoints through which external communication with clients take place.

3.3.1 External Packages

- `Microsoft.AspNetCore`:
  This package is the framework supporting the entire web application.

- `Microsoft.EntityFrameworkCore.Design`:
  This package allows the EF Core migration tool to work on the project.

- `Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore`:
  This package allows debugging EF Core migrations.

- `npgsql.EntityFrameworkCore.PostgreSQL`:
  This package is the database driver providing EF Core its interoperation with PostgreSQL.

- `Microsoft.AspNetCore.Identity.EntityFrameworkCore`:
  This package provides authentication and authorization of users through integration with EF Core.

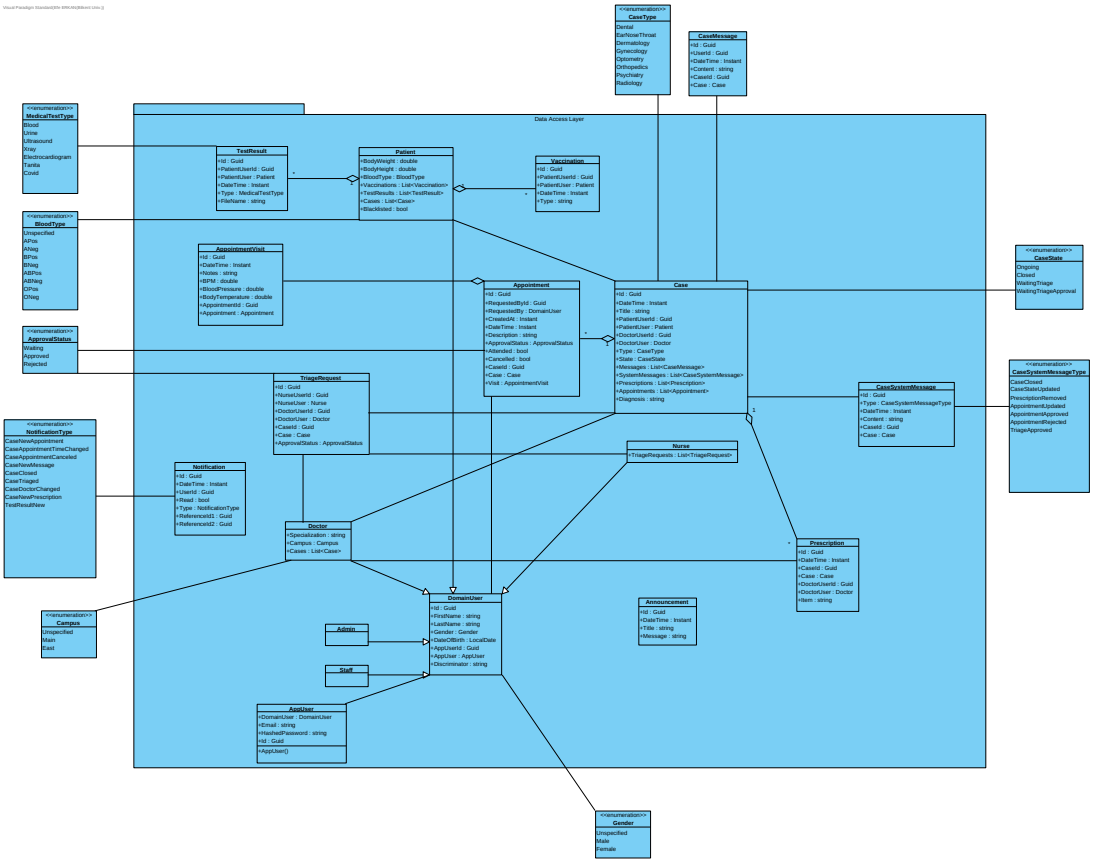
- `BCrypt.Net-Next`:
  This package provides the *bcrypt* algorithm which is currently among the best in securely hashing user passwords.

- `Microsoft.AspNetCore.Mvc.NewtonsoftJson`:
  This package is the most popular JSON library for .NET, used as the default solution in most cases.

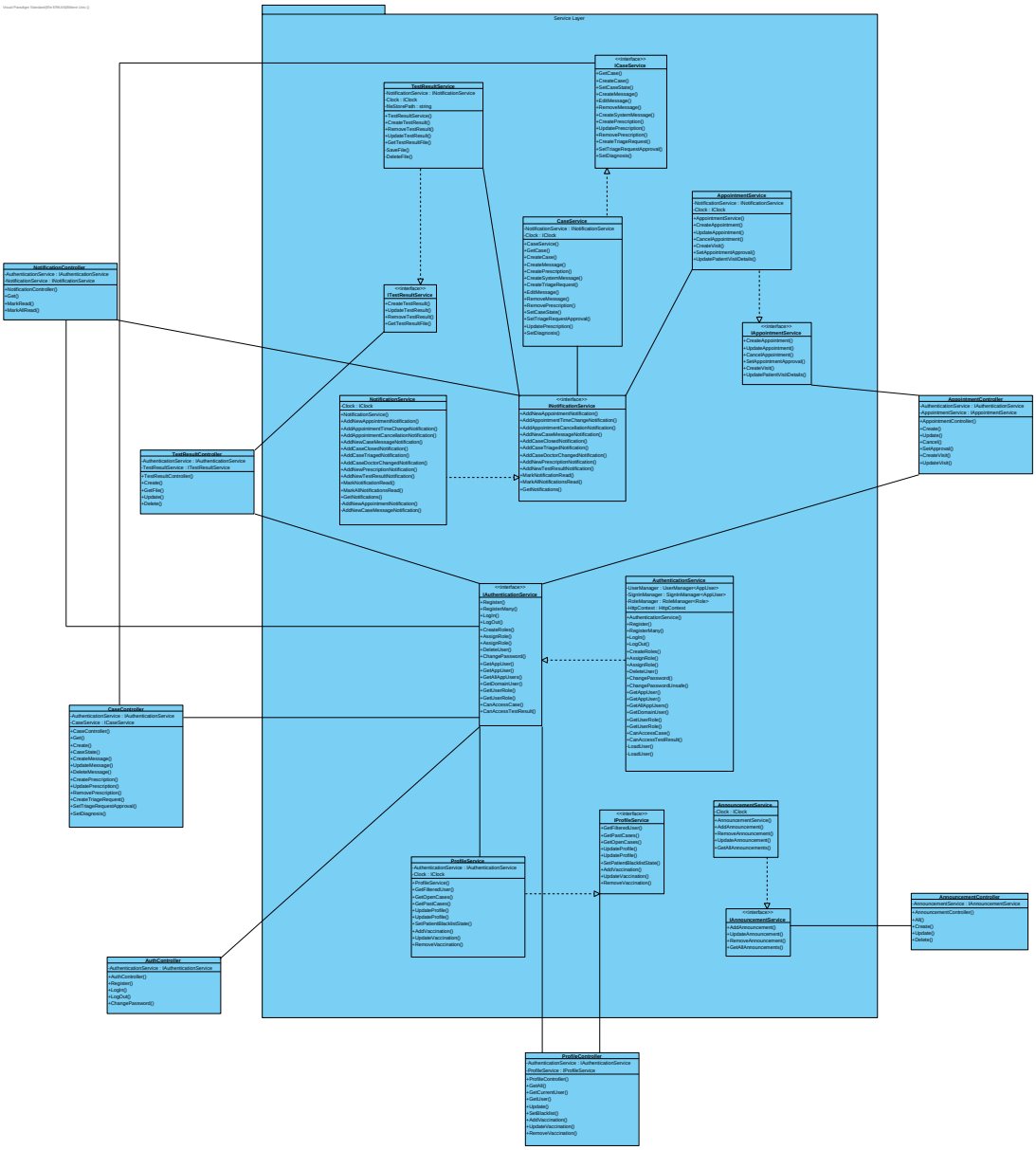
- `Microsoft.AspNetCore.SpaProxy`:
  This package allows the .NET build process to launch and proxy with the client application.

3.4 Class Diagrams of Layers

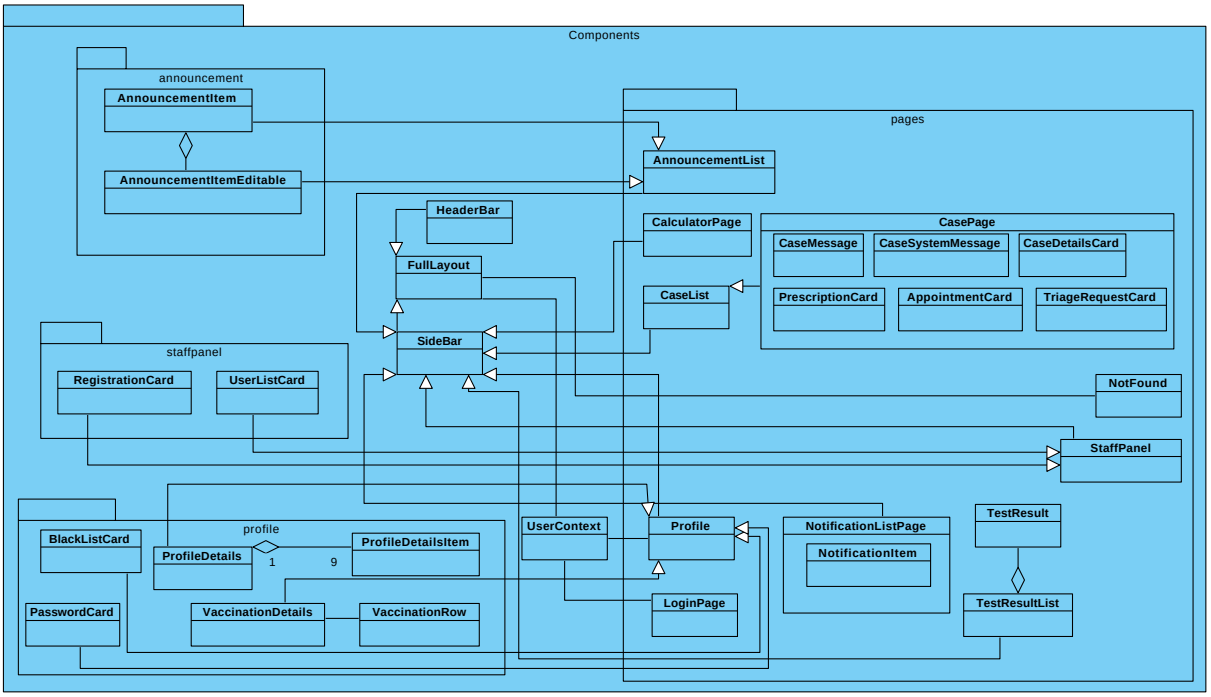
3.4.1 Data Access Layer



3.4.2 Business Logic Layer



3.4.3 User Interface Layer



4 Glossary

- **EF Core:** Microsoft's object relational mapper solution.
- **ORM:** Abbreviation for **O**bject **R**elational **M**apper.
- **.NET:** Microsoft's giant framework for C# applications.
- **ASP.NET:** Microsoft's giant framework for web-based C# applications.
- **PostgreSQL:** A popular relational database management system.
- **JSON:** Abbreviation for **J**ava**S**cript **O**bject **N**otation.
- **Hashing Passwords:** Cryptographically obscuring passwords such that they are practically irrecoverable.
- **Repository:** A very popular design pattern to provide an abstraction to persistent data.
- **API Endpoint:** The application programming interface endpoint through which communication with the system can occur.
- **Application Entrypoint:** The sum of the build process and configuration of the initialization of the system.
- **HTTP:** The most commonly used protocol over which web servers and clients communicate.
- **HTTPS:** Wraps the HTTP protocol into a cryptographically secured transmission protocol.
- **Enum:** A special type of programming entity that can be used to enumerate hard-coded types, mostly mapped to integers.
- **Enum Class:** An enum-like class adding specialized capability to the enum concept.
- **Record:** A C# reference type with value-based equality.
- **Docker:** A software virtualization product.
- **Dependency Injection:** A software design pattern that inverses the dependency control by injecting dependencies into a dependent object.