Πανεπιστήμιο Πειραιώς – Τμήμα Πληροφορικής

Πρόγραμμα Μεταπτυχιακών Σπουδών

«Πληροφορική»

Μεταπτυχιακή Διατριβή

Τίτλος Διατριβής	**Ανάπτυξη διαδικτυακής εφαρμογής με το Blazor framework και το Entity Framework Core της Microsoft** **Development of a web application based on Microsoft's Blazor framework and Entity Framework Core**
Ονοματεπώνυμο Φοιτητή	**Δημήτριος Πίτσιος**
Πατρώνυμο	**Χρήστος**
Αριθμός Μητρώου	**ΜΠΠΛ17041**
Επιβλέπων	**Ευθύμιος Αλέπης, Αναπληρωτής Καθηγητής**

Τριμελής Εξεταστική Επιτροπή

Ευθύμιος Αλέπης	Πατσάκης Κωνσταντίνος	Βίρβου Μαρία
Αναπληρωτής Καθηγητής	Αναπληρωτής Καθηγητής	Καθηγήτρια

Ημερομηνία Παράδοσης **2021 / 12**

Contents

Περίληψη 4
1. Εισαγωγή 5
1.1 Razor 5
1.2 Blazor 6
1.3 Dependency-Injection (DI) 8
1.4 Entity Framework Core 10
1.5 Project Structure 17
2. Ανάλυση Εφαρμογής 22
2.1 Root-Component *App.razor* 22
2.1.1 LogIn στην εφαρμογή μέσω του *LogInForm* Component 25
2.1.2 Navigation Menu 32
2.2 Η Σελίδα DiagnosticCenters.razor 36
2.2.1 *DataTable* component 36
2.2.2 *ModalDialog* component 41
2.2.3 Ανάλυση της λειτουργικότητας της σελίδας 46
2.3 Η Σελίδα *DiagnosticCenterRecords.razor* 51
2.3.1 Table-Per-Hierarchy (TPH) mapping 52
2.3.2 *TabContainer.razor* component 53
2.3.3 Μερικές πολυπλοκότερες λειτουργικότητες 57
Βιβλιογραφία 65

Περίληψη

Την σημερινή εποχή, η πανδημία του κορωνοϊού έχει επιβαρύνει με επιπλέον άγχος τη ζώη των ανθρώπων, που για να ανταποκριθούν στις καθημερινές τους υποχρεώσεις εκτίθενται στον κίνδυνο της μόλυνσης από τον ιό COVID-19. Προκειμένου να αισθανόμαστε ασφαλείς με τους ανθρώπους που συναναστρεφόμαστε, είναι απαραίτητη η υποβολή του καθένα σε τακτικούς ελέγχους ανίχνευσης του ιού. Λόγω των νέων αυτών δεδομένων προέκυψε η ανάγκη οργάνωσης διαγνωστικών κέντρων στους δήμους κάθε περιοχής, που θα πραγματοποιούν δωρεάν **rapid test** και μίας εφαρμογής διαδικτύου μέσω της οποίας θα δίνεται η δυνατότητα στον υπεύθυνο οργανισμό υγείας να διαχειρίζεται τα αποτελέσματα αυτών των test. Για την ανάπτυξη της εφαρμογής αυτής πρέπει να δημιουργηθεί μία βάση δεδομένων όπου θα υπάρχουν καταχωρημένοι οι χρήστες, οι οποίοι όταν αποκτούν πρόσβαση στην εφαρμογή θα τους παραχωρείται κάποιος ρόλος. Ανάλογα με τον ρόλο τους θα έχουν πρόσβαση σε διαφορετικές λειτουργικότητες της εφαρμογής. Για παράδειγμα ένας χρήστης που έχει ρόλο ‘staff’ έχει το δικαίωμα να διαχειρίζεται τις πληροφορίες του συστήματος για τους ανθρώπους που υποβάλλονται στη διαδικασία των ελέγχων, ενώ ένας άλλος χρήστης με ρόλο ‘doctor’ έχει το δικαίωμα να καταχωρεί τα αποτελέσματα στο σύστημα. Ένας επιπλέον ρόλος χρήστη που υποστηρίζει το σύστημα είναι αυτός του ‘admin’ που αναλαμβάνει την διαχείριση των χρηστών αλλά και των γεωγραφικών δεδομένων που αφορούν τις πόλεις και τις περιοχές όπου τα διαγνωστικά κέντρα οργανώνουν τη δράση τους.

Η παρούσα εργασία αφορά λοιπόν μία εφαρμογή διαδικτύου (web application), που βασίζεται στο ASP.NET Core framework της Microsoft, το οποίο είναι open-source και δίνει τη δυνατότητα ανάπτυξης και εκτέλεσης εφαρμογών ανεξαρτήτως λειτουργικού συστήματος (Windows, Mac, Linux). Μερικά ακόμα πλεονεκτήματα που προσφέρει το framework είναι ο μηχανισμός Razor και η τεχνολογία του Blazor, που είναι αλληλένδετη με τον εν λόγω μηχανισμό. Ένα τρίτο πλεονέκτημα είναι ένα απλό και εύχρηστο dependency injection σύστημα (DI-container). Στο κεφάλαιο της εισαγωγής που ακολουθεί, γίνεται μία αναφορά κάποιων βασικών στοιχείων για τις τεχνολογίες που αναφέρθηκαν ώστε να τεθεί μία κοινή βάση προκειμένου να αναλύθει στη συνέχεια ο πηγαίος κώδικας που ορίζει τον τρόπο λειτουργίας της εφαρμογής.

1. Εισαγωγή

1.1 Razor

Ο μηχανισμός Razor επιτρέπει την ανάμιξη C# κώδικα και HTML, χρησιμοποιώντας το σύμβολο "@" για την μετάβαση από HTML σε C#. Για παράδειγμα, με τον παρακάτω κώδικα παράγονται οι γραμμές ενός HTML στοιχείου του είδους table:

```
<table>
       <tbody>
       @foreach (var user in users)
       {
            <tr>
                <td>@user.Name</td>
            </tr>
       }
       </tbody>
</table>
```

Παρατηρούμε ότι κάθε φορά που θέλουμε να παρεμβάλλουμε C# κώδικα ανάμεσα στην HTML σήμανση, εισάγουμε το σύμβολο "@". Μπορούμε επομένως να εισάγουμε στο αρχείο μίας ιστοσελίδας, C# εκφράσεις (expressions), που ακολουθούν το σύμβολο "@", των οποίων το αποτέλεσμα αφού υπολογισθεί εμφανίζεται ως HTML. Εάν ένα C# expression είναι σύνθετο τότε αυτό πρέπει να τοποθετηθεί μέσα σε παρενθέσεις:

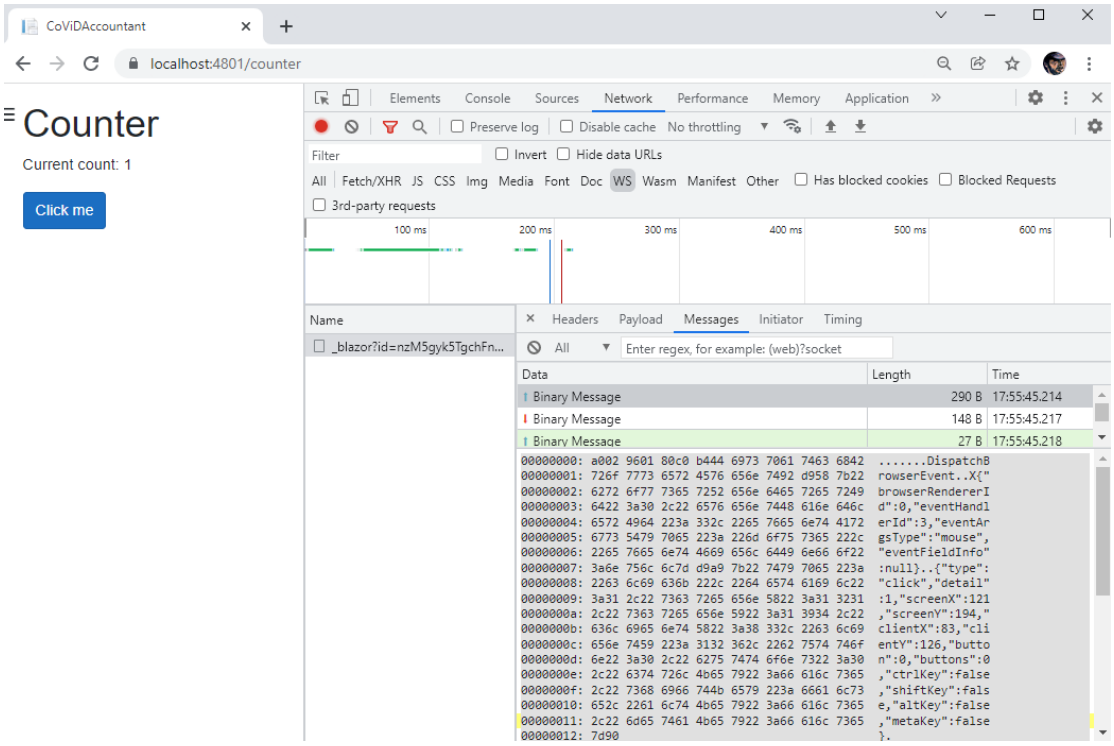
```
@{ var numbers = Enumerable.Range(1, 10); } // Get numbers from 1 - 10

@foreach (var number in numbers)
{
       @(number * 10)
}
```

Σημείωση: Μία δήλωση (statement) πρέπει να περικλείεται από αγκύλες, ενώ για τις δομές επανάληψης (ή ελέγχου) δεν είναι απαραίτητο.

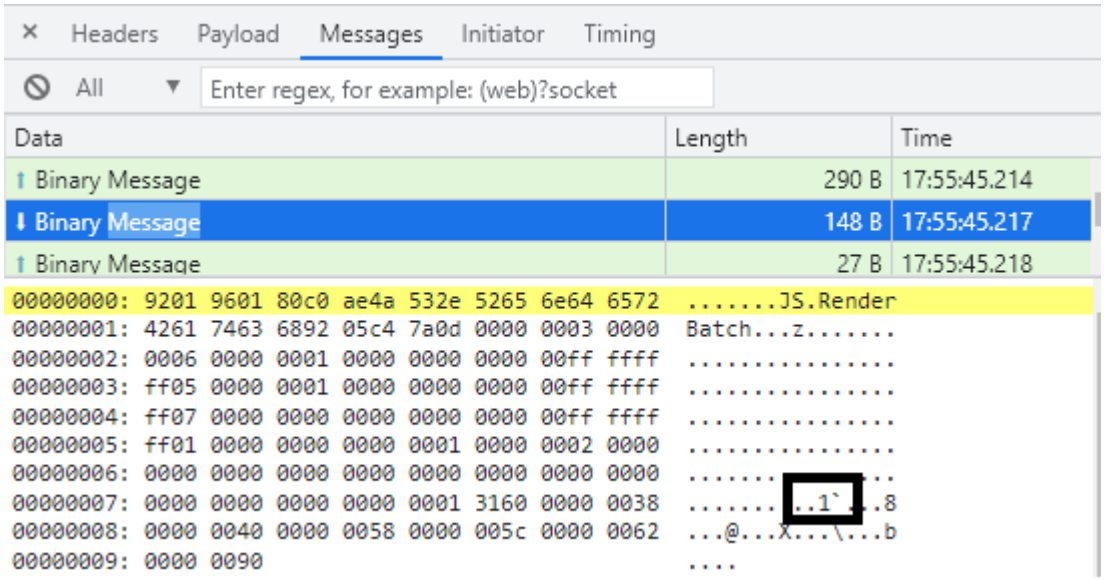
1.2 Blazor

Τα αρχεία που υποστηρίζουν την Razor σύνταξη έχουν κατάληξη .cshtml (razor pages) ή .razor (razor components). Τα components (.razor) αποτελούν τα δομικά στοιχεία της εφαρμογής μας και γενικότερα κάθε εφαρμογής που αναπτύσσεται χρησιμοποιώντας την τεχνολογία του Blazor. Ένα ***component*** είναι ένα τμήμα της διεπαφής της εφαρμογής με τον χρήστη, όπως μία σελίδα, ένας διάλογος ή μία φόρμα, που περιέχει όπως είδαμε παραπάνω λογική εκτέλεσης κώδικα, όπως οι δομές ελέγχου (if-else) και επανάληψης (foreach, for, while), καθιστώντας έτσι μία ιστοσελίδα δυναμική. Το Blazor είναι λοιπόν ένα framework που προσφέρει την δυνατότητα δημιουργίας πλούσιων και διαδραστικών διεπαφών (UI), χρησιμοποιώντας ως γλωσσα προγραμματισμού την C# αντί της Javascript, σε συνδυασμό με την γλώσσα σήμανσης HTML και μορφοποίησης αυτής μέσω CSS.

Υπάρχουν δύο μοντέλα ανάπτυξης εφαρμογών μέσω του Blazor. Το πρώτο βασίζεται στο νέο *'πρότυπο'* της WebAssembly (WASM), μέσω του οποίου τα components μίας εφαρμογής εκτελούνται στον browser (client-side). Το δεύτερο μοντέλο βασίζεται στην τεχνολογία του SignalR, που είναι μία web-socket τεχνολογία μέσω της οποίας επιτυγχάνεται real-time επικοινωνία μέσω του client και του server όπου «τρέχει» (έχει εγκατεστημένο το .NET runtime) μια ASP.NET Core εφαρμογή. Πιο συγκεκριμένα, οι ενέργειες που πραγματοποιούνται client-side από τους χρήστες μιας εφαρμογής, ενεργοποιούν κάποια events που αποστέλλονται μέσω της SignalR σύνδεσης στον server όπου εκτελείται ο αντίστοιχος κώδικας. Κατόπιν μέσω της ίδιας σύνδεσης αποστέλλεται στον client ένα δυαδικής μορφής μήνυμα που περιγράφει τις αλλαγές που πρέπει να γίνουν στο UI. Η ανανέωση του UI της εφαρμογής, κατόπιν ενός event, δεν είναι ολική, δεν πραγματοποιείται δηλαδή επαναφόρτωση (refresh) της σελίδας αλλά υπολογίζεται το μικρότερο σύνολο από τις αλλαγές που απαιτούνται στο DOM (Document Object Model) για να ανανεωθεί το UI στον browser. Βλεπουμε στις παρακάτω εικόνες τα μηνύματα που αποστέλλονται τόσο από τον client όσο και από τον server, κατά την πραγματοποίηση του onclick-event στη σελίδα με route "/counter" (Counter.razor), που είναι ένα από τα προϋπάρχοντα αρχεία του template ενός BlazorServer project.



$1.2 - Εικόνα 1 - Μήνυμα από τον client στον server κατόπιν του onclick event



$1.2 - Εικόνα 2 - Μήνυμα από τον server στον client με τις αλλαγές που πρέπει να εφαρμοστούν στον browser

Τα δύο μοντέλα που αναφέραμε λέγονται Blazor WebAssembly και Blazor Server αντίστοιχα. Παρόλο που το WASM μοντέλο δεν θα μας απασχολήσει, καθώς η εφαρμογή που αναπτύχθηκε βασίζεται στο Server μοντέλο, αξίζει να αναφέρουμε ότι η WebAssembly είναι μια χαμηλού επιπέδου γλώσσα που αναπτύσσεται ως *'πρότυπο'* (Web-Standard) της Κοινοπραξίας Παγκοσμίου Ιστού (W3C) και υποστηρίζεται από όλους τους σύγχρονους browsers (Mozilla, Edge, Safari, Chrome). Έτσι υπάρχει πλέον η δυνατότητα και σε αλλές

γλώσσες προγραμματισμού (εκτός της Javascript), αφού μεταγλωτιστούν σε WebAssembly, να εκτελούνται με πολύ καλές επιδόσεις στον browser.

1.3 Dependency-Injection (DI)

Μία **ASP**.NET Core εφαρμογή υποστηρίζει το dependency-injection design-pattern που ικανοποιεί την dependency-inversion ή Inversion of Control (IoC) σχεδιαστική αρχή. Σύμφωνα με αυτή, μία κλάση που για να επιτελέσει τη λειτουργία της στηρίζεται σε άλλες κλάσεις (instance-variables), δεν πρέπει να εξαρτάται από την υλοποίησή τους αλλά να στηρίζεται σε αφαιρέσεις τους (π.χ. interface). Το ίδιο λοιπόν το framework (ASP.NET Core) εξυπηρετεί στην επίλυση αυτών των εξαρτήσεων μεταξύ των κλάσεων, παρέχοντας σε μία εξαρτώμενη κλάση τα απαραίτητα στιγμιότυπα των μελών της μέσω του constructor της. Για παράδειγμα βλέπουμε στην εικόνα που ακολουθεί, τον constructor της κλάσης του controller που είναι υπεύθυνος για το login των χρηστών της εφαρμογής. Το login ενός χρήστη εξαρτάται από διάφορες άλλες κλάσεις οι οποίες δημιουργούνται καθώς περνούνται ως παράμετροι στον constructor της εξαρτώμενης κλάσης (UsersController). Έτσι λοιπόν με τον όρο dependency-injection, εννοούμε την διαδικασία με την οποία παρέχονται σε ένα τμήμα κώδικα, τα προαπαιτούμενα resources. Ο πάροχος αυτών των resources καλείται **IoC container** ή **DI provider**.

Σημείωση: Οι κλάσεις *UserManager<T>* και *SignInManager<T>* αποτελούν μέρος του **Identity** framework, που εξυπηρετεί λειτουργίες σχετικές με την διαχείριση των χρηστών μιας εφαρμογής, όπως η δημιουργία ή η εύρεση ενός χρήστη, η ανάθεση ρόλου σε χρήστη, η δημιουργία προσωπικού κωδικού κτλ.

```
[Route("api/users")]
1 reference
public class UsersController : ControllerBase
{
    private readonly IDataProtector dataProtector;
    private readonly UserManager<User> userManager;
    private readonly SignInManager<User> signInManager;
    private readonly CoViDAccountantDbContext dbContext;

    0 references
    public UsersController(
        IDataProtectionProvider dataProtectionProvider,
        UserManager<User> userManager,
        SignInManager<User> signInManager,
        CoViDAccountantDbContext dbContext)
    {
        dataProtector = dataProtectionProvider.CreateProtector("SignIn");
        this.userManager = userManager;
        this.signInManager = signInManager;
        this.dbContext = dbContext;
    }

    [AllowAnonymous]
    [HttpGet("login")]
    0 references
    public async Task<IActionResult> Login(string t)
    {
```

$1.3 - Constructor Injection

1.4 Entity Framework Core

Ένα από τα κυριότερα κομμάτια μίας εφαρμογής είναι αυτό της επικοινωνίας με τη βάση δεδομένων και για αυτόν το σκοπό έχει δημιουργηθεί η βιβλιοθήκη Entity Framework Core (ή EF Core). Αυτή η εξαιρετικά χρήσιμη αλλά και πολύπλοκη βιβλιοθήκη, εξυπηρετεί στην σύνδεση του κόσμου μίας σχεσιακής βάσης δεδομένων με αυτόν του αντικειμενοστρεφούς προγραμματισμού, αποτελεί όπως λέμε έναν *object-relational mapper (O/RM).* Στον παρακάτω πίνακα βλέπουμε την αντιστοιχία μεταξύ των κόσμων αυτών.

Relational database	.NET software
Table	.NET class
Table columns	Class properties/fields
Rows	Elements in .NET collections—for instance, `List`
Primary keys: unique row	A unique class instance
Foreign keys: define a relationship	Reference to another class
SQL—for instance, `WHERE`	.NET LINQ—for instance, `Where(p => …`

$1.4 - EF Core mapping

Το EF Core προσφέρει τόσο την δυνατότητα της δημιουργίας μιας βάσης δεδομένων για την λειτουργία μιας εφαρμογής (code-first) όσο και την δυνατότα χρήσης μίας ήδη υπάρχουσας βάσης, που δεν έχει δημιουργηθεί μέσω της βιβλιοθήκης (database-first). Για την συγκεκριμένη εργασία έχει ακολουθηθεί η πρώτη προσέγγιση. Τα δύο βασικά μέρη αυτής της διαδικασίας είναι:

- Οι κλάσεις που θέλουμε να αντιστοιχηθούν με πίνακες στη βάση δεδομένων και αποτελούν το προγραμματιστικό μοντέλο της.
- Μία κλάση που κληρονομεί από την κλάση *DbContext* του EF Core, μέσω της οποίας διαμορφώνεται το σχήμα της βάσης που δημιουργείται. Μέσω αυτής της κλάσης πραγματοποιείται οποιαδήποτε τροποποίηση των εγγραφών της βάσης, κατά την εκτέλεση της εφαρμογής.

Βλέπουμε στην παρακάτω εικόνα τα μέλη αυτής της κλάσης-κλειδί της εφαρμογής που αντιστοιχούν στους πίνακες της βάσης που δημιουργείται.

```
public class CoViDAccountantDbContext : Microsoft.AspNetCore.Identity.EntityFrameworkCore.IdentityDbContext
{
    0 references
    public CoViDAccountantDbContext(DbContextOptions opts) : base(opts) { }

    2 references
    public DbSet<City> Cities { get; set; }
    1 reference
    public DbSet<District> Districts { get; set; }
    7 references
    public DbSet<DiagnosticCenter> DiagnosticCenters { get; set; }
    6 references
    public DbSet<Vaccine> Vaccines { get; set; }
    1 reference
    public DbSet<Person> Persons { get; set; }

    1 reference
    public DbSet<Record> Records { get; set; }
    4 references
    public DbSet<Vaccination> Vaccinations { get; set; }
    4 references
    public DbSet<CovidTest> CovidTests { get; set; }
```

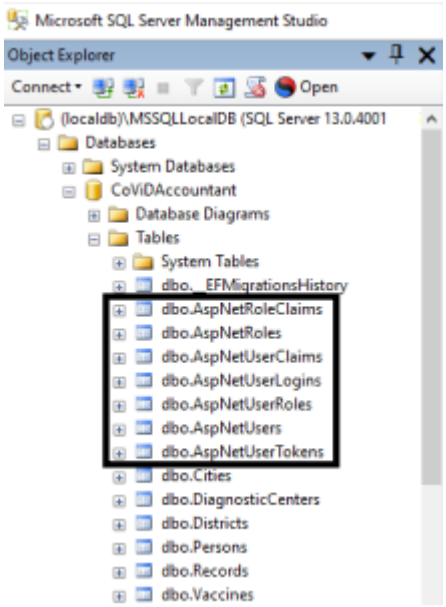
Σημείωσεις :

1. Στο εξής θα αναφερόμαστε στην κλάση *CoViDAccountantDbContext* ως application's DbContext.
2. Αναφέραμε προηγουμένως ότι το application's DbContext πρέπει να κληρονομεί από την κλάση *DbContext* του EF Core, αλλά στην περίπτωση μας κληρονομεί από την *IdentityDbContext* που κληρονομεί από την *IdentityUserContext* , επειδή χρησιμοποιούμε όπως είδαμε το Identity framework.

```
namespace Microsoft.AspNetCore.Identity.EntityFrameworkCore
{
    public abstract class IdentityDbContext<TUser, TRole, TKey, TUserClaim, TUserRole, TUserLogin, TRoleClaim, TUserToken> : IdentityUserContext
        where TUser : IdentityUser<TKey>
        where TRole : IdentityRole<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserRole : IdentityUserRole<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TRoleClaim : IdentityRoleClaim<TKey>
        where TUserToken : IdentityUserToken<TKey>
    {
        public IdentityDbContext(DbContextOptions options);
        protected IdentityDbContext();

        public virtual DbSet<TUserRole> UserRoles { get; set; }
        public virtual DbSet<TRole> Roles { get; set; }
        public virtual DbSet<TRoleClaim> RoleClaims { get; set; }

        protected override void OnModelCreating(ModelBuilder builder);
    }
}
```

```
namespace Microsoft.AspNetCore.Identity.EntityFrameworkCore
{
    //
    // Summary:
    //     Base class for the Entity Framework database context used for identity.
    //
    // Type parameters:
    //   TUser:
    //     The type of user objects.
    //
    //   TKey:
    //     The type of the primary key for users and roles.
    //
    //   TUserClaim:
    //     The type of the user claim object.
    //
    //   TUserLogin:
    //     The type of the user login object.
    //
    //   TUserToken:
    //     The type of the user token object.
    public abstract class IdentityUserContext<TUser, TKey, TUserClaim, TUserLogin, TUserToken> : DbContext
        where TUser : IdentityUser<TKey>
        where TKey : IEquatable<TKey>
        where TUserClaim : IdentityUserClaim<TKey>
        where TUserLogin : IdentityUserLogin<TKey>
        where TUserToken : IdentityUserToken<TKey>
    {
        public IdentityUserContext(DbContextOptions options);
        protected IdentityUserContext();

        public virtual DbSet<TUser> Users { get; set; }
        public virtual DbSet<TUserClaim> UserClaims { get; set; }
        public virtual DbSet<TUserLogin> UserLogins { get; set; }
        public virtual DbSet<TUserToken> UserTokens { get; set; }

        protected override void OnModelCreating(ModelBuilder builder);
    }
}
```

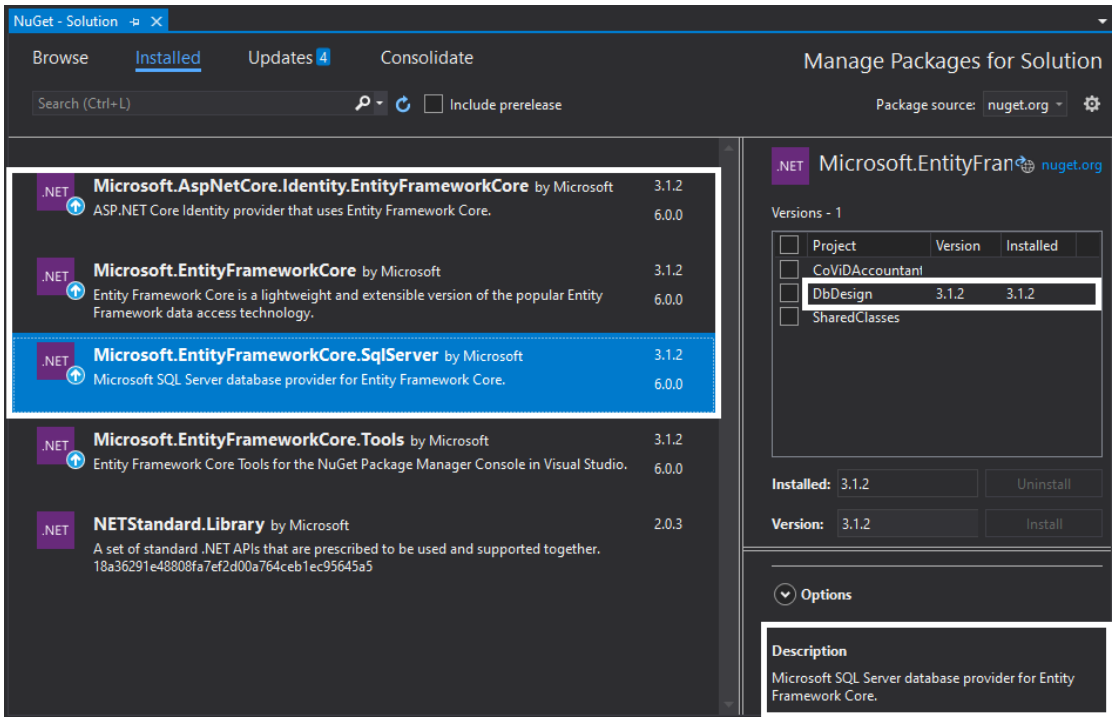
Σημείωση : Τα μέλη των παραπάνω κλάσεων του Identity-framework, αντιστοιχούν στους πίνακες που σχετίζονται με τους χρήστες και τα δικαιώματα που θα έχουν όταν έχουν ταυτοποιηθεί από την εφαρμογή κατά την σύνδεσή τους, και κατασκευάζονται κατά την δημιουργία της βάσης.



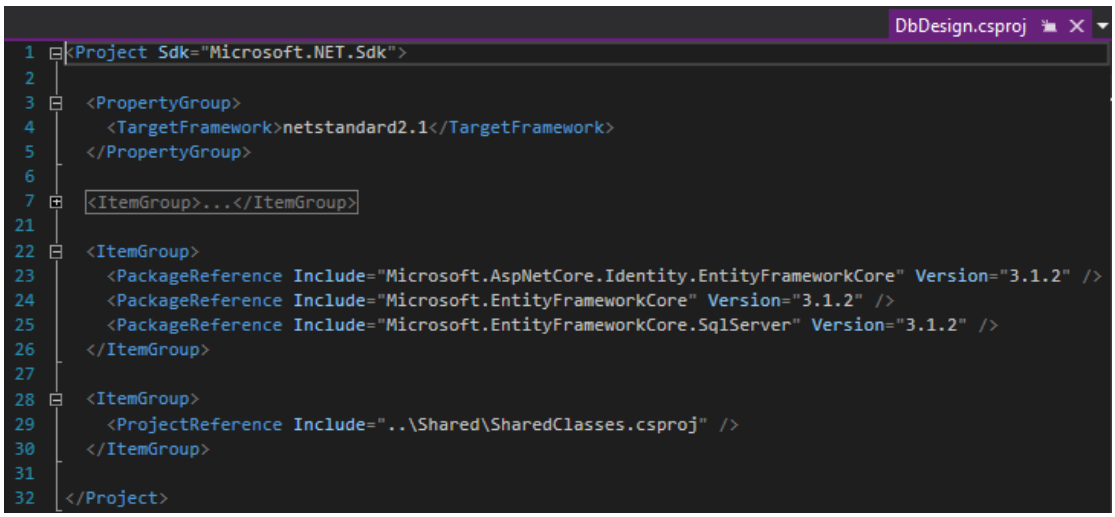
Βλέπουμε μέσω του διαχειριστικού συστήματος βάσεων (DBMS) που έχουμε εγκατεστημένο, τους πίνακες της βάσης δεδομένων που δημιουργείται από το EF Core.

Στο σημειωμένο πλαίσιο περιέχονται οι πίνακες που δημιουργούνται λόγω του ότι το application's DbContext κληρονομεί από το IdentityDbContext του Identity-framework.

Για να εγκαταστήσουμε στην εφαρμογή μας (solution) το EF Core μέσω του Visual Studio, πλοηγούμαστε ως εξής, στο menu-bar επιλέγουμε Tools > NuGet Package Manager > Manage NuGet Packages for Solution.



Στο δεξί μέρος της εικόνας παρατηρούμε ότι το solution αποτελείται από τρία projects εκ των οποίων μόνο το DbDesign έχει εγκατεστημένο το NuGet πακέτο που είναι επιλεγμένο. Το ίδιο ισχύει και για τα πρώτα δύο NuGets, κάτι που φαίνεται και αν επιλέξουμε στον SolutionExplorer το DbDesign-project, οπότε παίρνουμε την παρακάτω εικόνα.

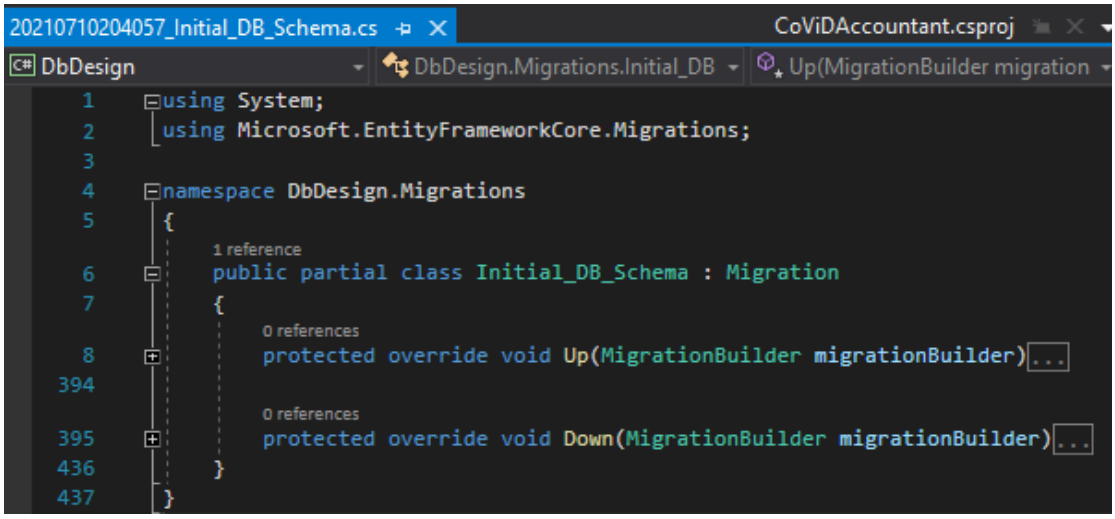


Το DbDesign-project αποτελεί ένα **class-library** (.NET Standard) που περιέχει την κλάση του application DbContext και τα μοντέλα από τα οποία ορίζουν τη δομή της βάσης δεδομένων. Αποτελεί εν ολίγοις το Business-Logic κομμάτι της εφαρμογής. Το SharedClasses-project είναι και αυτό ένα class-library, που περιέχει μία κλάση που εισήχθησε για να εξυπηρετήσει ένα συγκεκριμένο component (.razor αρχείο) του CoViDAccountant-project, και που υπάρχει ως αναφορά (reference) και στο DbDesign-project, όπως βλέπουμε και από την παραπάνω εικόνα. Το CoViDAccountant-project, που ακολουθεί το BlazorApp-template, έχει αναφορές και των δύο class-libraries και επομένως η λειτουργικότητα που παρέχεται από τα NuGets που έχουν αναφορές αυτά τα επιμέρους projects, είναι διαθέσιμη και σε αυτό το project.

CoViDAccountant.csproj

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="3.1.2">
      <PrivateAssets>all</PrivateAssets>
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
    </PackageReference>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\EF\DbDesign.csproj" />
    <ProjectReference Include="..\Shared\SharedClasses.csproj" />
  </ItemGroup>

</Project>
```

Είπαμε ότι το EF Core δημιουργεί τη βάση δεδομένων της εφαρμογής μέσω του application DbContext και των κλάσεων-μοντέλων που βρίσκονται στο DbDesign-project αλλά δεν έχουμε αναφέρει καθόλου πως γίνεται αυτό. Αυτο λοιπόν επιτυγχάνεται μέσω των migrations, τα οποία δημιουργούνται μέσω της Package Manager Console (Tools > Manage NuGet Packages > Package Manager Console), δίνοντας την εντολή *Add-Migration* ακολουθούμενης από ένα όνομα εντός εισαγωγικών, ενδεικτικό των τροποποιήσεων που θα πραγματοποιηθούν στη βαση. Όποτε υπάρχει κάποια διαφοροποίηση στην κλάση του application DbContext, η εντολή αυτή δημιουργεί κάποια αρχεία που περιέχουν τον κατάλληλο κώδικα C# μέσω του οποίου θα γίνουν οι επιθυμητές αλλαγές στη βάση. Κάθε τέτοιο αρχείο περιέχει δύο μεθόδους, την **Up** και την **Down**, που όπως γίνεται κατανοητό από τα ονόματα τους, η μεν εφαρμόζει τις αλλαγές στη βάση όταν τρέχουμε την εντολή *Update-Database* και η δε αναιρεί τις αλλαγές αυτές μέσω της εντολής *Remove-Migration*.



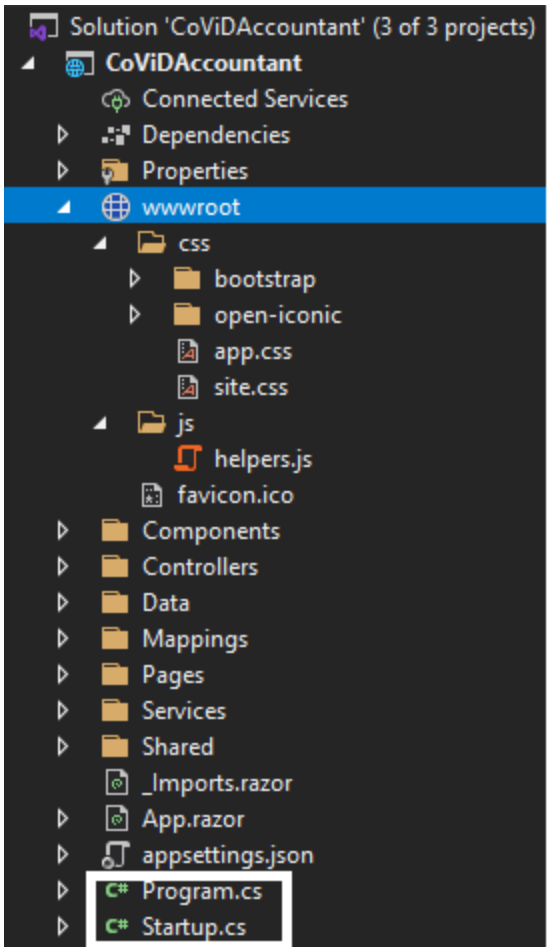
Σημείωση: Μία τεχνική λεπτομέρεια που αφορά την εντολή Remove-Migration είναι ότι πρέπει πρώτα να τρέξουμε την εντολη Update-Database “<penultimate-migration-name>” για να επαναφέρουμε τη βάση στην προηγούμενη της κατάσταση πριν μπορέσουμε να αναιρέσουμε ένα migration.

Για να μπορούμε να κάνουμε μετατροπές στη βάση μέσω του Package Manager Console και των εντολών που αναφέραμε πρέπει να έχουμε εγκατεστημένο στο κύριο project (Blazor) το Microsoft.EntityFrameworkCore.Tools NuGet πακέτο.

Ενδεικτικά βλέπουμε την εντολή έντός της μεθόδου Up, που δημιουργεί τον συνδετικό πίνακα μεταξύ των χρηστών και των ρόλων τους.

```
migrationBuilder.CreateTable(
       name: "AspNetUserRoles",
       columns: table => new
       {
              UserId = table.Column<Guid>(nullable: false),
              RoleId = table.Column<Guid>(nullable: false)
       },
       constraints: table =>
       {
              table.PrimaryKey("PK_AspNetUserRoles",
                                  x => new { x.UserId, x.RoleId });
              table.ForeignKey(
                     name: "FK_AspNetUserRoles_AspNetRoles_RoleId",
                     column: x => x.RoleId,
                     principalTable: "AspNetRoles",
                     principalColumn: "Id",
                     onDelete: ReferentialAction.Cascade);
              table.ForeignKey(
                     name: "FK_AspNetUserRoles_AspNetUsers_UserId",
                     column: x => x.UserId,
                     principalTable: "AspNetUsers",
                     principalColumn: "Id",
                     onDelete: ReferentialAction.Cascade);
       });
```

Παρατηρούμε ότι περιέχει οδηγίες για το όνομα, τις κολώνες και τους περιορισμούς του πίνακα, όπως ότι τα foreign-keys στους πίνακες των χρηστών (AspNetUsers) και των ρόλων (AspNetRoles) δεν επιτρέπεται να είναι NULL, ότι δεν επιτρέπονται πολλαπλές εγγραφές με το ίδιο UserId και RoleId (το πρωτεύων κλειδί είναι σύνθετο) και ότι σε περίπτωση διαγραφής κάποιου χρήστη ή ρόλου να διαγράφονται όλες οι εγγραφές αυτού του πίνακα που περιείχαν το Id αυτού του χρήστη (UserId) ή το Id αυτού του ρόλου (RoleId). Υπενθυμίζουμε σε αυτό το σημείο ότι, για την συγκεκριμένη εφαρμογή, όλοι οι πίνακες που δημιουργούνται στη βάση, αποτελούν μέλη του application's DbContext, με τύπο DbSet<T>. Οι συγκεκριμένοι (AspNetUsers, AspNetRoles, AspNetUserRoles κ.τ.λ.) είναι μέλη του *IdentityDbContext*.



1.5 Project Structure

Σε μία ASP.NET Core εφαρμογή όλα τα στατικά αρχεία που στέλνονται μέσω ενός HTTP request πρέπει να βρίσκονται στον φάκελο wwwroot (web-root folder). Έχουμε δημιουργήσει τον 'js' φάκελο για τα Javascript αρχεία και έχουμε προσθέσει στον φάκελο css το αρχείο app.css. Το αρχείο Program.cs περιέχει την *Main* μέθοδο που είναι το σημείο εκκίνησης της εφαρμογής. Εκεί δημιουργείται ένα αντικείμενο τύπου IHost που αποτελεί τον ακρογωνιαίο λίθο της εφαρμογής καθώς είναι υπεύθυνο για την εκκίνησή της. Πιο συγκεκριμένα η Main μέθοδος ενός Blazor Server App project περιεχεί μία μόνο εντολή:

```
CreateHostBuilder(args).Build().Run();
```

Παρακάτω βλέπουμε την static μέθοδο *CreateHostBuilder* :

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                {
                    webBuilder.UseStartup<Startup>();
                });
```

```
public class Startup
{
    0 references
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    2 references
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references
    public void ConfigureServices(IServiceCollection services)...

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)...
}
```

$1.5 - Startup class

```
// Summary:
//     Initializes a new instance of the Microsoft.Extensions.Hosting.HostBuilder class
//     with pre-configured defaults.
//
// Parameters:
//   args:
//     The command line args.
//
// Returns:
//     The initialized Microsoft.Extensions.Hosting.IHostBuilder.
//
// Remarks:
//     The following defaults are applied to the returned Microsoft.Extensions.Hosting.HostBuilder
//     • set the Microsoft.Extensions.Hosting.IHostEnvironment.ContentRootPath to the
//     result of System.IO.Directory.GetCurrentDirectory
//     • load host Microsoft.Extensions.Configuration.IConfiguration from "DOTNET_"
//     prefixed environment variables
//     • load host Microsoft.Extensions.Configuration.IConfiguration from supplied command
//     line args
//     • load app Microsoft.Extensions.Configuration.IConfiguration from 'appsettings.json'
//     and 'appsettings.[Microsoft.Extensions.Hosting.IHostEnvironment.EnvironmentName].json'
//     • load app Microsoft.Extensions.Configuration.IConfiguration from User Secrets
//     when Microsoft.Extensions.Hosting.IHostEnvironment.EnvironmentName is 'Development'
//     using the entry assembly
//     • load app Microsoft.Extensions.Configuration.IConfiguration from environment
//     variables
//     • load app Microsoft.Extensions.Configuration.IConfiguration from supplied command
//     line args
//     • configure the Microsoft.Extensions.Logging.ILoggerFactory to log to the console,
//     debug, and event source output
//     • enables scope validation on the dependency injection container when Microsoft.Extensions.
//     is 'Development'
public static IHostBuilder CreateDefaultBuilder(string[] args);
```

Η γραμμή που είναι σημειωμένη με μπλε ρίγα αναφέρει πως το μέλος Configuration της κλάσης Startup (με τύπο *IConfiguration*) κατασκευάζεται κατά την κλήση της μεθόδου CreateHostBuilder (κατά την εκκίνηση της εφαρμογής) διαβάζοντας από το αρχείο *appsettings.json.* Η κλάση Configuration που χρησιμοποιεί η κλάση Startup αποτελεί ένα dependency, καθώς η κλάση Startup για να επιτελέσει τη λειτουργία της χρειάζεται ένα αντικείμενο τύπου *IConfiguration*. Η κλάση αυτή παρέχεται αυτόματα στον constructor της

Startup κλάσης μέσω του dependency-injection design-pattern που υλοποιεί το ASP.NET Core framework. Οι κλάσεις που παρέχονται μέσω αυτού του μηχανισμού (IoC container / DI provider), στους constructors των κλάσεων που τις έχουν ως dependency, λέγονται **services** και χωρίζονται σε δύο κατηγορίες. Η μία κατηγορία είναι τα ***framework*** *services* που αφορούν τις κλάσεις που είναι μέρος του ASP.NET Core και του Blazor framework. Η άλλη είναι τα ***application*** *services* που αφορούν τις κλάσεις που είναι προσαρμοσμένες στις ανάγκες μιας εφαρμογής και δημιουργούνται από τους προγραμματιστές της. Για να παρέχεται λοιπόν στον constructor μίας κλάσης, αυτόματα κάποια κλάση που χρειάζεται, πρέπει πρώτα να έχει καταχωρηθεί ως service. Η καταχώρηση αυτή γίνεται μέσω της μεθόδου *ConfigureServices* (της Startup κλάσης), που περιλαμβανει μία παράμετρο τύπου *IServiceCollection*.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddRazorPages();
        services.AddServerSideBlazor();
        //services.AddSingleton<WeatherForecastService>();

        services.AddScoped<UIService>();
        services.AddScoped<DiagnosticCenterManagementService>();
        services.AddScoped<GenericOperationsService>();

        services.AddDbContext<CoViDAccountantDbContext>(options =>
        {
                options.UseSqlServer(
                        Configuration.GetConnectionString("CoViDAccountantConn"));
        });

        services.AddIdentity<User, Role>(options => …)
                .AddEntityFrameworkStores<CoViDAccountantDbContext>()
                .AddDefaultTokenProviders();

        services.ConfigureApplicationCookie(options => …)
        services.AddAuthentication();
        services.AddAuthorization();
}
```

Οι μεθόδοι *AddRazorPages* και *AddServerSideBlazor* καταχωρούν διάφορα διαθέσιμα framework-services, ενώ στη συνέχεια καταχωρούνται ως services οι κλάσεις *UIService*, *DiagnosticCenterManagementService* και *GenericOperationsService* που ανήκουν στην κατηγορία των application-services. Ακολουθεί η καταχώρηση του *CoViDAccountantDbContext* (application's DbContext) μέσω της μεθόδου *AddDbContext<T>*, όπου T είναι μία παράμετρος που κληρονομεί από το `Microsoft.EntityFrameworkCore.DbContext`. Η μέθοδος αυτή δέχεται μία παράμετρο τύπου *Action<DbContextOptionsBuilder>*, που εκφράζει μία συνάρτηση που έχει μία παράμετρο τύπου *DbContextOptionsBuilder* και επιστρέφει *void.* Για παράδειγμα, η μέθοδος που ακολουθεί μπορεί να περαστεί ως παράμετρος στην μέθοδο *AddDbContext<DbContextOptionsBuilder* >.

```
public void ActionMethod (DbContextOptionsBuilder options)
{
        options.UseSqlServer(Configuration.GetConnectionString(”ConnStrName”));
}
```

Το ConnectionString ανακτάται από το framework-service με τύπο IConfiguration, που κατασκευάζεται όπως είπαμε από το αρχείο appsettings.json, μέσω της *GetConnectionString,* στην εκκίνηση της εφαρμογής. Για το λόγο αυτό, στο appsettings.json πρέπει να υπάρχει μία δήλωση όπως η παρακάτω:

```
  "ConnectionStrings": {
    "CoViDAccountantConn":
       "Server=(localdb)\\MSSQLLocalDB;Database=CoViDAccountant;Trusted_Connection=True;"
  }
```

Τέλος, ακολουθούν η καταχώρηση του Identity συστήματος και του Authentication - Authorization.

```
<Router AppAssembly="@typeof(Program).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)"/>
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>
```

$1.5 - Αρχική μορφή του App.razor κατά τη δημιουργία του Blazor-Server-App project

Όλα τα αρχεία στον φάκελο Pages ξεκινούν με το *razor-directive* **@page** ακολουθούμενο από κάποιο path. Για παράδειγμα το Counter.razor που είδαμε στην εικόνα 1 της παραγράφου 1.2 ξεκινά ως εξης: *@page "/counter"*. Το App.razor αποτελεί το “root-component” της εφαρμογής, όπως θα δούμε αμέσως παρακάτω. Περιέχει το Router component το οποίο είτε απεικονίζει κάποια σελίδα για την οποία ταιριάζει το path ενός request που γίνεται από τον client, είτε ενημερώνει τον client ότι δεν βρέθηκε σελίδα.

```
@page "/"
@namespace CoViDAccountant.Pages
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@{
    Layout = null;
}

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>CoViDAccountant</title>
    <base href="~/" />
    <link rel="stylesheet" href="css/bootstrap/bootstrap.min.css" />
    <link href="css/site.css" rel="stylesheet" />
    <link href="css/app.css" rel="stylesheet" />
</head>
<body>
    <component type="typeof(App)" render-mode="ServerPrerendered" />

    <div id="blazor-error-ui">
        <environment include="Staging,Production">
            An error has occurred. This application may no longer respond until reloaded.
        </environment>
        <environment include="Development">
            An unhandled exception has occurred. See browser dev tools for details.
        </environment>
        <a href="" class="reload">Reload</a>
        <a class="dismiss">✕</a>
    </div>

    <script src="_framework/blazor.server.js"></script>

    <script src="js/helpers.js"></script>
</body>
</html>
```

$1.5 – Pages_Host.cshtml

Το αρχείο _Host.cshtml, που βρίσκεται στον φάκελο Pages, είναι η σελίδα που φορτώνεται αρχικά, κατά την εκκίνηση της εφαρμογής. Βλέπουμε ότι σε αυτό το αρχείο ορίζεται η θέση του App.razor. Επιπλέον εδώ φορτώνεται το ***blazor.server.js*** javascript-αρχείο μέσω του οποίου, εγκαθίσταται η SignalR σύνδεση με τον server και πραγματοποιείται η ανταλλαγή των μηνυμάτων που είδαμε στις εικόνες της παραγράφου 1.2. Τέλος, βλέπουμε ότι εδώ πρέπει να περιληφθούν όλα τα css και javascript αρχεία από τον φάκελο *wwwroot* που θέλουμε να χρησιμοποιούνται από την εφαρμογή μας. Να επισημάνουμε σε αυτό το σημείο ότι μία διαφορά των razor-pages (αρχεία με .cshtml extension) και των blazor-components (αρχεία με .razor extension) είναι ότι οι μεν είναι στατικές σελίδες και κάθε φορά που πηγαίνουμε σε μία razor σελίδα πραγματοποιείται επαναφόρτωση της σελίδας στον server, ενώ τα blazor components ανανεώνονται μερικώς μέσω της SignalR σύνδεσης που έχει δημιουργηθεί.

2. Ανάλυση Εφαρμογής

2.1 Root-Component *App.razor*

```
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(Program).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData"
                                DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    <CoViDAccountant.Components.NotAuthorized />
                </NotAuthorized>
                <Authorizing>
                    <h1>Authenticating...</h1>
                </Authorizing>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

$2.1 - 1 - App.razor

Θα ξεκινήσουμε την ανάλυση της εφαρμογής από τον authentication μηχανισμό της. Με τον όρο authentication εννοούμε ότι για να αποκτήσει κάποιος χρήστης πρόσβαση σε μία σελίδα της εφαρμογής πρέπει πρώτα να έχει ταυτοποιηθεί. Βλέπουμε στην παραπάνω εικόνα το App.razor αρχείο όπου, σε σύγκριση με το αρχικό που είδαμε στην παραγραφο 1.5 της εισαγωγης, έχει προστεθεί το component *CascadingAuthenticationState.* Τα *Router* και *Found*/*NotFound* components προϋπήρχαν, ενώ το component *RouteView* έχει αντικατασταθεί από το component *AuthorizeRouteView*, που μας δίνει την δυνατότητα να χειριστούμε τις περιπτώσεις που ο χρήστης δεν είναι είτε authenticated είτε authorized. Στην περίπτωση που ο χρήστης έχει ταυτοποιηθεί αλλά δεν έχει τα κατάλληλα δικαιώματα για να αποκτήσει πρόσβαση στο περιεχόμενο μίας σελίδας, απεικονίζεται το *NotAuthorized* component το οποίο περιλαμβάνει ένα μήνυμα προς το χρήστη. Αυτή η λειτουργία επιτυγχάνεται σε συνδυασμό με το ***@attribute [Authorize]*** *directive*, εάν αυτό παραληφθεί το *AuthorizeRouteView* δεν αποκλείει την πρόσβαση σε μία σελίδα. Στην παρακάτω εικόνα βλέπουμε το NotAuthorized.razor που βρίσκεται στον φάκελο Components. Το component *AuthorizeRouteView* παρέχει στο εμφωλευμένο *NotAuthorized* component μία παράμετρο τύπου Task<AuthenticationState> από την οποία μπορεί να ανακτηθεί το authentication-status ενός συνδεδεμένου χρήστη. ‘Ετσι, στην περίπτωση που ένας χρήστης δεν είναι authenticated, μέσω της εντολής

```
navigationManager.NavigateTo("/account/login?returnUrl=" +
                             System.Net.WebUtility.UrlEncode(uri.PathAndQuery));
```

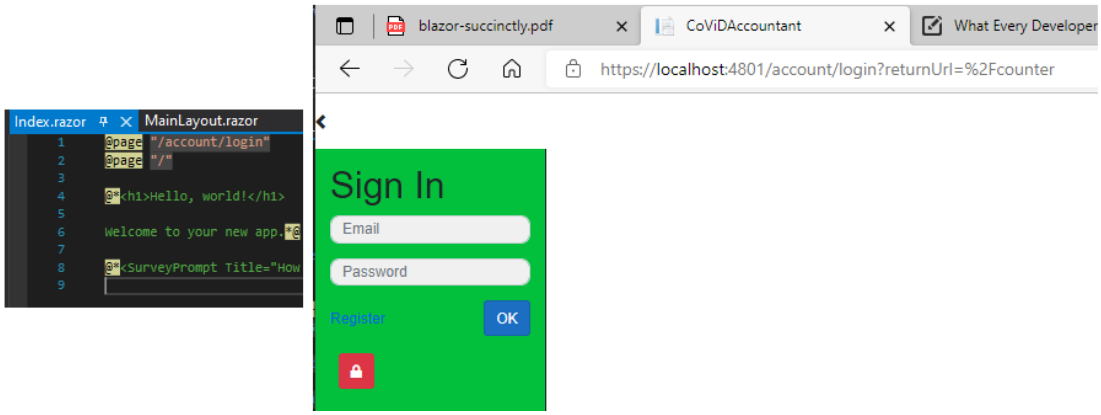
τον οδηγούμε στην αρχική σελίδα της εφαρμογής, που είναι το αρχείο Index.razor και το μόνο που περιέχει είναι τα page-directives @page "/account/login" και @page "/"

```
@inject AuthenticationStateProvider authenticationStateProvider
@inject NavigationManager navigationManager

@if (_notAuthorized)
{
    <section class="section">
        <div class="container">
            <div>
                <h1>Authorization Failure!</h1>
                <p>You're not authorized to see this page.</p>
            </div>
        </div>
    </section>
}

@code {
    [CascadingParameter] Task<AuthenticationState> _authStateTask { get; set; }

    private bool _notAuthorized;

    protected override async Task OnInitializedAsync()
    {
        //var authState = await
              authenticationStateProvider.GetAuthenticationStateAsync();
        var authState = await _authStateTask;
        var authenticated = authState.User.Identity.IsAuthenticated;
        if (!authenticated)
        {
            // If the user is not authenticated redirect them
            // to the sign in page
            var uri = new Uri(navigationManager.Uri);
            navigationManager.NavigateTo("/account/login?returnUrl=" +
                            System.Net.WebUtility.UrlEncode(uri.PathAndQuery));
        }
        else
        {
            // If the user is signed in, authorization failed
            _notAuthorized = true;
        }
    }
}
```

$2.1 - 2 - NotAuthorized.razor

Το τμήμα του URL που ακολουθεί το "?" λέγεται *query-component* ή *query-string* και είναι ο συνήθης τρόπος για να δηλώνουμε παραμέτρους, δηλαδή key-value τιμές. Στην περίπτωση που ένα query-string αποτελείται από πολλές παραμέτρους, αυτές θα πρέπει να διαχωρίζονται με το σύμβολο "&" (ή το ";"). 'Ενα τέτοιο προσαυξημένο URL δεν επηρεάζει τον μηχανισμό του routing, ο οποίος αγνοεί το query-string που περιέχει και προβάλλει τη σελίδα Index. Στην παρακάτω όμως εικόνα βλέπουμε ότι στη σελίδα Index υπάρχει μία φόρμα για να κάνουμε login – είναι το *LogInForm* component στον φάκελο Components. Αυτό συμβαίνει διότι το *LogInForm* βρίσκεται στο *NavMenu* component το οποίο με τη

σειρά του βρίσκεται στο *MainLayout* component. Τα αρχεία NavMenu.razor και MainLayout.razor βρίσκονται στον φακελο *Shared*. Έχουμε συναντήσει ξανά τον **τύπο** του MainLayout στο *App* component, όπου χρησιμοποιείται ως η ***DefaultLayout*** παράμετρος του *AuthorizeRouteView*.



$2.1 - 3 - Index.razor

Το αρχείο MainLayout.razor ορίζει το καλούπι για όλες τις σελίδες της εφαρμογής που δεν προσδιορίζουν κάποιον άλλο τύπο ως layout. Κάθε **blazor component** (.razor) ισοδυναμεί με μία κλάση με όνομα αυτό του .razor αρχείου. Ένα component - μια κλάση ή ένας τύπος - για να μπορεί να χρησιμοποιηθεί ως layout πρέπει να κληρονομεί από την κλάση ***LayoutComponentBase*** που περιέχει ένα μέλος τύπου *RenderFragment* με το όνομα *Body*. Το Body αντιστοιχεί στο περιεχόμενο των σελίδων που δεν ορίζουν κάποιον συγκεκριμένο τύπο layout, οπότε λαμβάνουν το MainLayout ως default μέσω του App component.

```
@inherits LayoutComponentBase

<div class="main">
        <NavMenu />

        <div class="content px-4">
             @Body
        </div>
</div>
```

$2.1 - 4 - MainLayout.razor

2.1.1 LogIn στην εφαρμογή μέσω του *LogInForm* Component

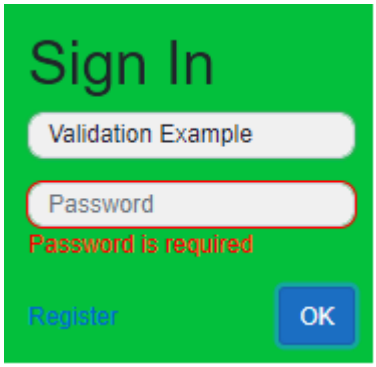
```
<div class="p-3" style="margin: 0 auto;"
     @attributes="@(Hidden ? Attrs.Hidden : Attrs.Empty)">

    <h1 class="title">Sign In</h1>

    <EditForm Model="@_userModel" OnValidSubmit="@Submit">
        <DataAnnotationsValidator />

        <div class="form-group">
            <InputText class="form-control" @bind-Value="@_userModel.Email"
                       placeholder="Email" />
            <ValidationMessage For="() => _userModel.Email" />
        </div>
        <div class="form-group">
            <InputText class="form-control" type="password"
                       @bind-Value="@_userModel.Password"
                       placeholder="Password" />
            <ValidationMessage For="() => _userModel.Password" />
        </div>
        @if (_register)
        {
            <div class="form-group">
                <InputText class="form-control" type="password"
                           @bind-Value="@_userModel.PasswordCrossCheck"
                           placeholder="Retype Password" />
            </div>
        }

        <div class="form-group row mx-auto align-items-center">
            <a href="" class="color-info" @onclick="RegisterLoginSwitch">
                @(!_register ? "Register" : "Login")
            </a>
            <div class="col"></div>
            <button class="btn btn-primary" type="submit">
                OK
            </button>
        </div>

        @if (!String.IsNullOrEmpty(_error))
        {
            <div class="alert alert-danger" role="alert">@_error</div>
        }
    </EditForm>
</div>
```

$2.1.1 - 1 - LogInForm.razor

Το αρχείο LogInForm.razor περιέχει ένα *EditForm* component το οποίο αποτελεί την blazor εκδοχή του HTML στοιχείου *<form>.* Το component αυτό δέχεται μία παράμετρο Model που στην περίπτωση μας είναι είναι ένα αντικείμενο τύπου *SignInModel*.

```
private class SignInModel
{
        [Required(ErrorMessage = "Email is required")]
        public string Email { get; set; }
        [Required(ErrorMessage = "Password is required")]
        public string Password { get; set; }
        public string PasswordCrossCheck { get; set; }
}
```

Επίσης δέχεται ως παράμετρο μία μέθοδο μέσω της οποίας πραγματοποιείται η υποβολή των στοιχείων της φόρμας, εφόσον ικανοποιούνται οι απαιτούμενες συνθήκες που ορίζονται μέσω των *Validation Attributes*. Εάν δεν πληρείται κάποια από τις συνθήκες αυτές, η μέθοδος *Submit* που έχει ανατεθεί στην *OnValidSubmit* παράμετρο της φόρμας δεν θα κληθεί, κατά την υποβολή της με το πάτημα του OK-button που φέρει το χαρακτηριστικό type=”submit”. Το μοντέλο δηλαδή της φόρμας θα είναι invalid και τότε θα εμφανιστούν τα αντίστοιχα μηνύματα στα πεδία που δεν έχουν συμπληρωθεί από τον χρήστη. Για να αποτρέψουμε την υποβολή της φόρμας μέσω του submit-button στην περίπτωση που το μοντέλο είναι invalid και να προβληθούν τότε τα αντίστοιχα μηνύματα λάθους, πρέπει να έχουμε συμπεριλάβει στο εσωτερικό του EditForm το component *<DataAnnotationsValidator />*.



Σημείωση: Μπορούμε να παρακάμψουμε αυτή την συμπεριφορά χρησιμοποιώντας την *OnSubmit* παράμετρο του EditForm component και να χειριστούμε εμείς το validation του μοντέλου.

Στην παρακάτω σελίδα βλέπουμε την μέθοδο Submit της φόρμας. Η λογική που έχει ακολουθηθεί για το register των χρηστών της εφαρμογής είναι πολύ απλή, βασίζεται στον ρόλο ‘admin’ του συστήματος και είναι η εξής: προκειμένου να γίνει ένας χρήστης register στην εφαρμογή, πρέπει ένας χρήστης με ρόλο admin να έχει πρώτα δημιουργήσει μία εγγραφή στον πίνακα AspNetUsers της βάσης με το επιθυμητό email. Αυτό συμβαίνει διότι όταν κληθεί η *Submit* θα αναζητήσει μέσω του *UserManager* έναν χρήστη με το email που συμπληρώνεται στην φόρμα και αν δεν τον βρει θα προβληθεί το μήνυμα “Invalid Email”. Εάν βρεθεί κάποιος χρήστης με το συγκεκριμένο email και η φόρμα δεν είναι σε register mode, θα γίνει έλεγχος του κωδικού, ο οποίος εάν είναι σωστός θα πραγματοποιηθεί κλήση στο API για το login στην εφαρμογή, μέσω της εντολής:

```
NavMan.NavigateTo($"api/users/login?queryStringParam={pdata}", forceLoad:true);
```

Η παράμετρος που περνάμε στο query-string του Url περιέχει τα εξής τμήματα:

- Το Id του χρήστη

- Ένα token που δημιουργείται αφού γίνει επαλήθευση του κωδικού
- Το url της σελίδας όπου θα επιστρέψουμε αφότου ολοκληρωθεί το login. Αυτή η πληροφορία θα περιλαμβάνεται στο query-string εάν προηγουμένως έχουμε κάνει log-out από το σύστημα, οπότε οδηγούμαστε από τη σελίδα που βρισκόμασταν, πίσω στο login της εφαρμογής, μέσω του *NotAuthorized* component (Εικόνα $2.1 – NotAuthorized.razor).

```
private async Task Submit()
{
        var email = _userModel.Email.Trim();
        var user = await UserManager.FindByEmailAsync(email)
                ?? await UserManager.FindByNameAsync(email);
        if (user != null)
        {
            if (!_register && await UserManager.CheckPasswordAsync(
                                        user,
                                        _userModel.Password))
            {
                var token = await UserManager.GenerateUserTokenAsync(
                                    user,
                                    TokenOptions.DefaultProvider,
                                    "SignIn");
                var data = $"{user.Id}|{token}";

                var parsedQuery = System.Web.HttpUtility.ParseQueryString(
                                        new Uri(NavMan.Uri).Query);
                var returnUrl = parsedQuery["returnUrl"];

                if (!string.IsNullOrWhiteSpace(returnUrl))
                {
                    data += $"|{returnUrl}";
                }

                var protector = ProtectionProvider.CreateProtector("SignIn");
                var pdata = protector.Protect(data);

                NavMan.NavigateTo(
                     $"api/users/login?queryStringParam={pdata}",
                     forceLoad:true);
            }
            else if (_register)
            {
                . . .
            }
            else
            {
                _error = "Invalid password";
            }
        }
        else
        {
            _error = "Invalid Email";
        }
}
```

$2.1.1 - 2 - Submit method of LogInForm's EditForm component

Να επισημάνουμε σε αυτό το σημείο ότι οι κλάσεις *UserManager, ProtectionProvider* και *NavMan* συγκαταλέγονται στα application-services, που παρέχονται αυτόματα στην κλάση *LogInForm* κατά τη δημιουργία της, επειδή έχουν οριστεί ως μέλη της που έχουν "στολιστεί" με το ***[Inject] attribute***.

```
[Inject] UserManager<User> UserManager { get; set; }
[Inject] NavigationManager NavMan { get; set; }
[Inject] IDataProtectionProvider ProtectionProvider { get; set; }
```

$2.1.1 - 3 - Inject Attribute

```
[HttpGet("login")]
public async Task<IActionResult> Login(string queryStringParam)
{
	var data = dataProtector.Unprotect(queryStringParam);

	var parts = data.Split('|');

	var identityUser = await userManager.FindByIdAsync(parts[0]);

	var isTokenValid = await userManager.VerifyUserTokenAsync(
				identityUser,
				TokenOptions.DefaultProvider,
				"SignIn",
				parts[1]);

	if (isTokenValid)
	{
		await signInManager.SignInAsync(identityUser, true);
		if (parts.Length == 3 && Url.IsLocalUrl(parts[2]))
		{
			return Redirect(parts[2]);
		}
		var url = await RedirectTo(identityUser);
		if (!string.IsNullOrEmpty(url))
		{
			return Redirect(url);
		}
		return Redirect("/");
	}
	else
	{
		return Unauthorized("STOP!");
	}
}

private async Task<string> RedirectTo(User user)
{
	var userWithRole = await dbContext.Users
		.Include(u => u.UserRole).ThenInclude(ur => ur.Role)
		.SingleOrDefaultAsync(u => u.Id == user.Id);

	if (new string[] { Role.StaffRoleKey, Role.DoctorRoleKey}
			.Contains(userWithRole.UserRole?.Role.Name) == true)
	{
		return "/diagnostic-centers";
	}
	return "";
}
```

$2.1.1 - 4 - Login μέθοδος του αρχείου UsersController.cs

Στην παραπάνω εικόνα έχουμε τονίσει την μέθοδο *RedirectTo* μέσω της οποίας μπορούμε να ανακατευθύνουμε τον συνδεδεμένο χρήστη στην επιθυμητή σελίδα, ανάλογα με το ρόλο του. Για να το πετύχουμε αυτό συμπεριλαμβάνουμε στο LINQ query, που χρησιμοποιούμε για να ανακτήσουμε τον χρήστη από τη βάση, το τμήμα:

```
.Include(u=>u.UserRole).ThenInclude(ur => ur.Role) ή εναλλακτικά
.Include(u=>u.UserRole.Role)
```

‘Ετσι το αποτέλεσμα της εκτέλεσης του query που είναι τύπου User, θα περιλαμβάνει - αν υπάρχει - και την συσχετιζόμενη συνδετική οντότητα *UserRole* με την αντίστοιχη *Role* οντότητα. Καθώς υπάρχει το ενδεχόμενο να μην έχει ανατεθεί κάποιος ρόλος σε έναν χρήστη και συνεπώς να μην υπάρχει κάποια σχετιζόμενη *UserRole* οντότητα, στη συνθήκη που ακολουθεί χρησιμοποιούμε τον ***null-conditional-operator*** της C# “**?.**” που επιστρέφει *null* εάν η έκφραση που προηγείται ειναι *null*. Σε αυτή την περίπτωση η έκφραση

```
new string[] { Role.StaffRoleKey, Role.DoctorRoleKey}.Contains(null) == true
```

είναι *false.*

Όταν έχει γίνει επιτυχώς ο έλεγχος των στοιχείων του χρήστη εκτελείται η παραπάνω μέθοδος *Login* της *UsersController* κλάσης, η οποία φέρει το attribute *[Route(“api/users”)].* Η μέθοδος αυτή λοιπόν καλείται, όταν γίνεται κλήση της *NavigateTo* μεθόδου της *NavigationManager* κλάσης, με παράμετρο το route που είναι ορισμένο στον controller ακολουθούμενο από το route που έχει οριστεί μέσω του *Http Verb Attribute – [HttpGet]* της μεθόδου. Η συγκεκριμένη μέθοδος δέχεται επιπλέον ως παράμετρο μία μεταβλητή τύπου string που αντιστοιχίζεται με το query-string που περιγράψαμε προηγουμένως. Η λειτουργία αυτή που βασίζεται στην μεταβίβαση της εκτέλεσης του κώδικα σε μία controller-κλάση επιτυγχάνεται μέσω της *Configure* μεθόδου της *Startup* κλάσης, όπου ορίζεται μία σειρά από ελέγχους για την εξυπηρέτηση ενός request. Ακολουθεί ένα απόσπασμα από την μέθοδο αυτή:

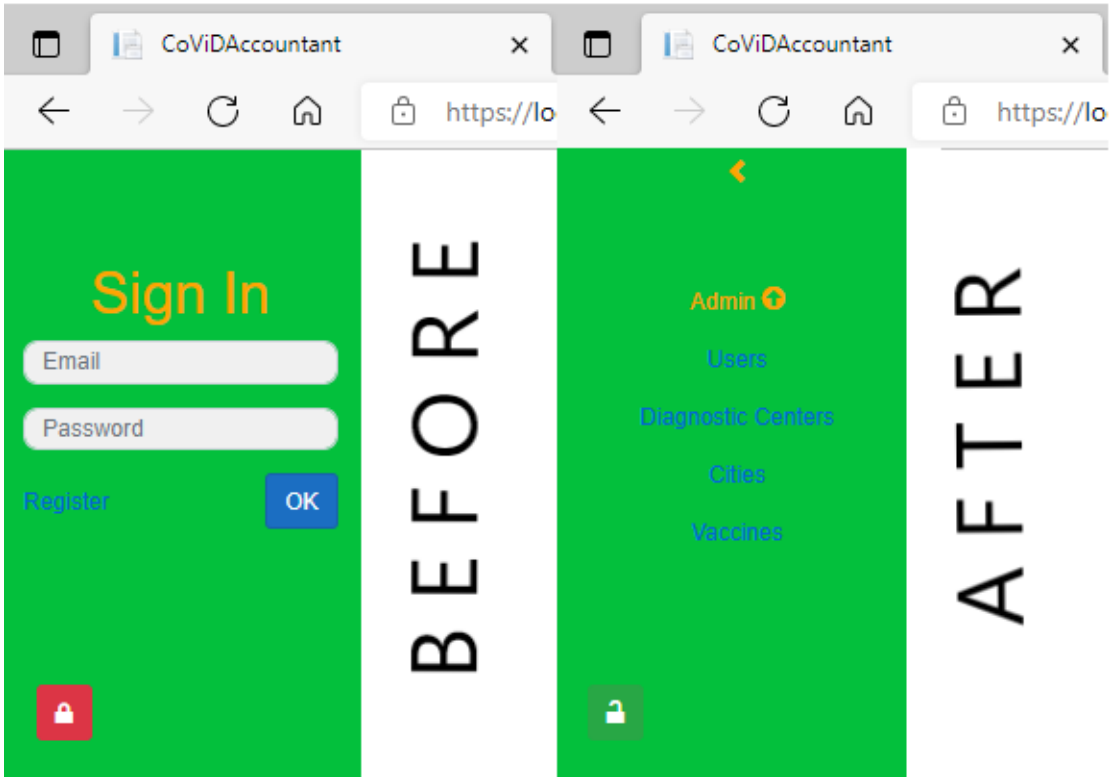
```
app.UseEndpoints(endpoints =>
{
        endpoints.MapControllers();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/_Host");
});
```

$2.1.1 - Snippet from Startup’s Configure method

Μέσω της μεθόδου *MapControllers* προσθέτουμε στην διαδικασία του routing (που βασίζεται στα paths των *@page-directives* των σελίδων) τα paths από τις controller-κλάσεις.

Προτού προχωρήσουμε στην ανάλυση της υποδομής των σελίδων της εφαρμογής, θα δούμε στην επόμενη υποενότητα την συμπεριφορά του μενού που εμφανίζεται στη θέση της login-φόρμας, αφότου γίνει επιτυχώς η ταυτοποίηση κάποιου χρήστη.

2.1.2 Navigation Menu



$2.1.2 - 1 - Before / After Login

Σε αυτή την υποενότητα θα εξηγήσουμε την συμπεριφορά του μενού που εμφανίζεται στην αριστερή πλευρά της οθόνης. Από τον κώδικα του NavMenu component που ακολουθεί παρακάτω πρέπει να επικεντρωθούμε αρχικά στην μεταβλητή *_sidebarExpanded*, ανάλογα με την οποία αναθέτουμε στα HTML στοιχεία την CSS-κλάση *expanded*. Ανατρέχοντας στο αρχείο *app.css* στον *wwwroot* φάκελο, βλέπουμε τους CSS κανόνες για την κλάση *top-row* και για την εμφωλευμένη σε αυτή κλάση *sidebar-menu*. Παρατηρούμε ότι όταν αυτές οι κλάσεις συνδυαστούν με την κλάση *expanded*, γίνεται μία τροποποίηση της απόστασης των αντίστοιχων HTML στοιχείων από την αριστερή πλευρά του παραθύρου του browser.

```css
.top-row {
    padding: 10px;
    position: fixed;
    z-index: 500;
    height: 60px;
    background-color: #03c03c;
    color: orange;
    width: 250px;
    height: 36px;
    left: -214px;
    text-align: right;
}

.top-row.expanded {
    left: 0;
    text-align: center;
}
```

```css
.top-row .sidebar-menu {
    position: absolute;
    z-index: 900;
    top: 0;
    margin-top: 36px;
    width: 250px;
    height: calc(100vh - 36px);
    overflow-y: auto;
    background-color: #03c03c;
    left: -250px;
    /*border-right: 1px solid #0f162b;
}

.top-row .sidebar-menu.expanded {
    left: 0;
}
```

$2.1.2 - 2 - CSS-rules in app.css file

Στην περίπτωση που δεν παρουσιάζεται η κλάση *expanded* η ορισμένη απόσταση από την αριστερή πλευρά είναι αρνητική και όταν είναι ίση (κατά απόλυτη τιμή) με το πλάτος του στοιχείου HTML το στοιχείο αυτό χάνεται από το παράθυρο. Αυτό συμβαίνει στην περίπτωση του στοιχείου *div* με την κλάση *sidebar-menu*, που βρίσκεται εντός του *div* με κλάση *top-row*. Για το *div*-στοιχείο με κλάση *top-row* δεν συμβαίνει αυτό, δεν εξαφανίζεται δηλαδή από την οθόνη αν δεν φέρει και την κλάση *expanded*, ώστε να έχουμε την δυνατότητα να επαναφέρουμε το μενού κάνοντας κλικ στην περιοχή που παραμένει εντός της οθόνης.

```
<div class="top-row d-flex align-items-center
            @(_sidebarExpanded ? "expanded" : "")" id="navMenu">
    @if(_loggedIn)
    {
        <div class="menu-icon w-100" @onclick="@((args) => ToggleMenu())">
            <span class="oi @_menuIconCssClass"></span>
        </div>
    }
    <div class="sidebar-menu @(_sidebarExpanded ? "expanded" : "")">

        <LogInForm @ref="_logIn" Hidden="_loggedIn" />

        @if (_loggedIn && _sidebarExpanded)
        {
            <ul class="nav flex-column" style="margin-top: 50px">
                <AuthorizeView Roles="@Role.AdminRoleKey">
                    <NavMenuItem Text="Admin">
                        <AdminSubMenu OnSelect="ToggleMenu" />
                    </NavMenuItem>
                </AuthorizeView>
                <AuthorizeView Roles="@string.Join(',', new string[] {
                                    Role.DoctorRoleKey, Role.StaffRoleKey })">
                    <NavMenuItem Text="Medical" Path="/medical/records"
                                 OnSelect="_ => ToggleMenu()" />
                </AuthorizeView>
            </ul>
        }

        <button class="btn btn-@_loggedInIconColorCssClass"
                style="position:absolute; bottom:25px;left:25px"
                @onclick="@LoggingOut">
            <span class="oi oi-lock-@_loggedInIconCssClass"></span>
        </button>
    </div>
</div>
```

$2.1.2 - 3 - NavMenu.razor

Μέσω του κουμπιού που βρίσκεται στην κάτω αριστερή γωνία του μενού μπορουμε να κάνουμε log-out από την εφαρμογή καλόντας και πάλι το API της εφαρμογής.

```
private void LoggingOut()
{
    if (_loggedIn)
    {
        _navigationManager.NavigateTo("api/users/logout", forceLoad: true);
    }
}
```

```
[Authorize]
[HttpGet("logout")]
public async Task<IActionResult> SignOut()
{
       await signInManager.SignOutAsync();
       return Redirect("/account/login");
}
```

$2.1.2 - 4 - Log-Out από την εφαρμογή

Σημείωση: Βλέπουμε ότι το *Authorize attribute* μπορούμε να το χρησιμοποιήσουμε και σε μεθόδους προκειμένου να αποκλείσουμε την εκτέλεση μίας λειτουργίας από μη εξουσιοδοτημένους χρήστες.

‘Ετσι οδηγούμαστε στην αρχική σελίδα της εφαρμογής, στην κατάσταση “before” της εικόνας 2.1.2-1, όπου ζητείται από τον χρήστη να κάνει log-in.

2.2 Η Σελίδα DiagnosticCenters.razor

Εξετάζοντας τη σελίδα των διαγνωστικών κέντρων θα δούμε δύο βασικά components που χρησιμοποιούνται εκτεταμένα στην εφαρμογή:

- **DataTable.razor**
- **ModalDialog.razor**

2.2.1 *DataTable* component

Το *DataTable* είναι ένα component που λειτουργεί ως template για την προβολή δεδομένων σε ένα HTML στοιχείο πίνακα. Εξυπηρετεί τη λειτουργία της σελιδοποίησης (paging), ώστε να μην χρειάζεται να φορτώσουμε στην μνήμη όλο τον όγκο των δεδομένων που αφορούν μία σελίδα, από την βάση δεδομένων, και την λειτουργία της αναζήτησης σύμφωνα με συγκεκριμένα κριτήρια. Το *ModalDialog* εξυπηρετεί στην εμφάνιση ενός διαλόγου μέσω του οποίου μπορούμε να προσθέσουμε μία νέα εγγραφή στην βάση ή να τροποποιήσουμε μία ήδη υπάρχουσα. Επίσης χρησιμοποιείται στην περίπτωση που θέλουμε να διαγράψουμε μία εγγραφή, όπου εμφανίζεται ένας διάλογος που ζητά να γίνει επιβεβαίωση της ενέργειας, προκειμένου να αποτραπεί μία εσφαλμένη διαγραφή δεδομένων.

Βλέπουμε στην παρακάτω εικόνα την δομή του DataTable component της σελίδας των διαγνωστικών κέντρων της εφαρμογής. Αρχικά παρατηρούμε ότι δέχεται τρεις παραμέτρους:

- **State** (τύπου *PageState*) Περιέχει τα εξής δεδομένα:
  - Την χωρητικότητα της σελίδας
  - Τον συνολικό αριθμό σελίδων που προκύπτει από τον συνολικό αριθμό των εγγραφών ενός πίνακα της βάσης και από το τρέχων μέγεθος της σελίδας
  - Την τρέχουσα σελίδα
  - Το κριτήριο αναζήτησης
- **OnStateChange** (τύπου *EventCallback*) Καλεί την μέθοδο που πραγματοποιεί refresh των δεδομένων όποτε συμβαίνει κάποια αλλαγή στην παράμετρο *State*
- **HasPaging** (τύπου *bool*) Αποτελεί μία συνθήκη για το αν το component υποστηρίζει την λειτουργία της σελιδοποίησης. Επειδή σε αυτή την σελίδα έχουν πρόσβαση όλοι οι ρόλοι, καθώς υπάρχει διασύνδεση των χρηστών με ρόλο “doctor” ή “staff” με διάφορα διαγνωστικά κέντρα μέσω της οντότητας *DiagnosticCenterUser*, επιθυμούμε η λειτουργία της σελιδοποίησης να υποστηρίζεται μόνο στην περίπτωση που ο συνδεδεμένος χρήστης έχει ρόλο “admin”, όπου δεν υπάρχει κάποιος περιορισμός στα δεδομένα που του προβάλλονται. Σε άλλη περίπτωση ο όγκος των δεδομένων θα μπορεί να υποστηριχτεί και από μία μόνο σελίδα.

Επίσης περιέχει και τρεις ακόμα παραμέτρους που είναι τύπου *RenderFragment* και αφορούν τα μέρη από τα οποία αποτελείται το component:

- **Actions**

- **TableHeader**
- **TableBody**

```
<DataTable State="_state" OnStateChange="Refresh" HasPaging="_loggedInAsStaffQuery == null">
    <Actions>
        @if(_loggedInAsStaffQuery == null)
        {
            <button class="btn btn-success btn-circle btn-md" @onclick="_ => OnModalShow()">
                <span class="oi oi-plus"></span>
            </button>
        }
    </Actions>
    <TableHeader>
        <tr>
            <th>Name</th>
            <th>District</th>
            <th>Address</th>
            @if (_loggedInAsStaffQuery == null)
            {
            <th></th>
            }
        </tr>
    </TableHeader>
    <TableBody>
        @foreach (var i in _items)
        {
            <tr>
                <td><a href="/diagnostic-centers/@i.Id/records" @i.Name</a></td>
                <td>@($"{i.Burg.Name} ({i.Burg.City.Name})")</td>
                <td>@i.Address.ToString()</td>
                @if(_loggedInAsStaffQuery == null)
                {
                <td class="actions actions-3x">
                    <button class="btn btn-light btn-circle" @onclick="_ => OnAssignUser(i)">
                        <span class="oi oi-script"></span>
                    </button>
                    <button class="btn btn-info btn-circle" @onclick="_ => OnModalShow(i)">
                        <span class="oi oi-pencil"></span>
                    </button>
                    <button class="btn btn-danger btn-circle" @onclick="_ => OnDelete(i)">
                            <span class="oi oi-minus"></span>
                    </button>
                </td>
                }
            </tr>
        }
    </TableBody>
</DataTable>
```

$2.2.1 - 1 - DataTable component in DiagnosticCenters.razor

Το *Actions* τμήμα αποτελεί το σημείο όπου θα τοποθετηθούν τα HTML στοιχεία μέσω των οποίων θα πραγματοποιούνται ενέργειες όπως είναι για παράδειγμα, η έναρξη της καταχώρησης μίας νέας εγγραφής ή το φιλτράρισμα των δεδομένων, μέσω της επιλογής από λίστα κάποιας συγκεκριμένης τιμής ενός σχετικού χαρακτηριστικού τους. Το *TableHeader* τμήμα αποτελεί την γραμμή κεφαλίδας του πίνακα, όπου υπάρχουν οι τίτλοι για τις πληροφορίες της κάθε στήλης του πίνακα, ενώ το *TableBody* τμήμα αποτελεί το μέρος όπου γίνεται η προβολή των δεδομένων που ανακτώνται από την βάση.

Σημείωση: Η μεταβλητή *_loggedInAsStaffQuery* θα είναι null μόνο στην περίπτωση που ο συνδεδεμένος χρήστης έχει ρόλο “admin”. Μόνο τότε θέλουμε να έχει τη δυνατότητα ο

χρήστης να πραγματοποιήσει ενέργειες όπως είναι η καταχώρηση, τροποποίηση ή διαγραφή ενός διαγνωστικού κέντρου, ή η σύνδεση ενός χρήστη (με ρόλο “doctor” ή “staff”) με ένα διαγνωστικό κέντρο, ώστε να μπορεί να έχει πρόσβαση στα δεδομένα που το αφορούν κατά την σύνδεση του στην εφαρμογή. Ο χρήστης μπορεί να μεταφερθεί στην σελίδα ενός διαγνωστικού κέντρου, κάνοντας κλικ στο όνομα του που αποτελεί σύνδεσμο.

```
<div class="row no-gutters">
        <div class="col-auto grid-actions">
            @Actions
        </div>
        <div class="col"></div>
        <div class="row align-items-center mx-auto">
            <input type="search" class="form-control col mr-2"
                   @bind="_searchTerm" @onkeyup="@OnSearchKeyUp"
                   placeholder="Search...">
            <button class="btn btn-primary btn-circle" type="button"
                    title="Search" @onclick="@(_ => Search(_searchTerm))">
                <span class="oi oi-magnifying-glass"></span>
            </button>
        </div>
</div>

<div class="table-responsive">
        <table class="table">
            <thead>
                @TableHeader
            </thead>
            <tbody>
                @if (State.TotalPages > 0)
                {
                    @TableBody
                }
                else
                {
                    <tr><td colspan="100" class="no-records">
                            No records found
                     </td></tr>
                }
            </tbody>
        </table>
</div>

@code {
    [Parameter] public RenderFragment TableHeader { get; set; }
    [Parameter] public RenderFragment TableBody { get; set; }
    [Parameter] public RenderFragment Actions { get; set; }
    [Parameter] public PageState State { get; set; }
    [Parameter] public EventCallback OnStateChange { get; set; }
    [Parameter] public bool HasPaging { get; set; } = true;

    private string _searchTerm;
    private async Task OnSearchKeyUp(KeyboardEventArgs args)
    {
        if (args.Key == "Enter")
            await Search(_searchTerm);
    }

    private async Task Search(string searchTerm)
    {
        State.Filter = searchTerm;
        await OnStateChange.InvokeAsync(null);
    }
    private async Task PageChanged(int page)
    {
        State.CurrentPage = page;
        await OnStateChange.InvokeAsync(null);
    }
}
```

$2.2.1 - 2 - DataTable.razor (Για λόγους απλότητας παραλείπεται το τμήμα που αφορά το paging)

Σημείωση: Ο τύπος *RenderFragment* χρησιμοποιείται για να μεταφέρουμε HTML περιεχόμενο σε ένα component. Πρέπει να προσδιορίσουμε σε ποια παράμετρο τύπου RenderFragment αντιστοιχεί ένα τμήμα HTML περικλειοντάς το σε "*ετικέτες*" με το όνομα της παραμέτρου. Για παράδειγμα, η HTML που αφορά την παράμετρο *Actions*, βρίσκεται μεταξύ των ετικετών **<Actions> </Actions>**. Παρακάτω θα δούμε μία εξαίρεση στον κανόνα.

2.2.2 *ModalDialog* component

```
@typeparam TModel
@if (_onDisplay)
{
    <div class="modal" tabindex="-1" style="display:block" role="dialog">
        <div class="modal-dialog">
            <div class="modal-content">
                <div class="modal-header" style="@CssStyle">
                    <h3 class="modal-title">@Header</h3>
                    <button type="button" class="close" onclick="Hide">
                        <span>X</span>
                    </button>
                </div>
                @if (!SystemicDialog)
                {
                    <EditForm EditContext="_editContext"
                              OnValidSubmit="OnSubmit">
                        <DataAnnotationsValidator />
                        @*<ValidationSummary />*@

                        <div class="modal-body">
                            @ChildContent

                            <button class="btn btn-danger" type="button"
                                    @onclick="Hide">
                                Cancel
                            </button>
@*CAUTION !!! ALL other buttons passed as content must be of type-button !!! *@
                            <button class="btn btn-primary" type="submit">
                                Save
                            </button>
                        </div>
                        <div class="modal-footer">
                            @if (!string.IsNullOrEmpty(Error))
                            {
                                <div class="alert-danger">
                                    @Error
                                </div>
                            }
                        </div>
                    </EditForm>
                }
                else
                {
                    <div class="modal-body">
                        @ChildContent
                    </div>
                }
            </div>
        </div>
    </div>
}
```

$2.2.2 - 1 - ModalDialog.razor

Η css-κλάση *modal* παρέχεται από το bootstrap.min.css αρχείο οπου αναζητώντας την βλέπουμε τους εξής κανόνες:

```
.modal{

            position:fixed; top:0; left:0;

            z-index:1050; display:none; width:100%; height:100%;

            overflow:hidden; outline:0

}
```

Το χαρακτηριστικό z-index είναι ο λόγος που η HTML, που περιέχεται εντός του στοιχείου *div*, εμφανίζεται μπροστά από την HTML μιας σελίδας, όταν το ModalDialog component είναι ορατό, δηλαδη η μεταβλητή *_onDisplay* έχει τιμή *true*.

```
public partial class ModalDialog<TModel> where TModel : class
{
        [Parameter] public RenderFragment ChildContent { get; set; }
        [Parameter] public EventCallback<TModel> Submit { get; set; }
        [Parameter] public string Header { get; set; }
        [Parameter] public string CssStyle { get; set; }
        [Parameter] public bool SystemicDialog { get; set; }

        private EditContext _editContext;
        private bool _onDisplay;

        public void Show(TModel model)
        {
            Error = string.Empty;
            if (model != null)
                _editContext = new EditContext(model);
            _onDisplay = true;
            StateHasChanged();
        }
        public void Hide()
        {
            _onDisplay = false;
        }
        public void ReValidate(FieldIdentifier  field)
        {
            _editContext.NotifyFieldChanged(field);
        }

        private async Task OnSubmit()
        {
            await Submit.InvokeAsync(_editContext.Model as TModel);
            Hide();
        }
}
```

$2.2.2 – 2 – ModalDialog.razor.cs

Σημείωση: Το HTML περιεχόμενο που μεταφέρουμε στο *ModalDialog* component μέσω της μεταβλητής *ChildContent* (τύπου *RenderFragment*) δεν χρειάζεται να περικλείεται μεταξύ **<ChildContent> </ ChildContent>** ετικετών**.**

Το component *ModalDialog* περιέχει ένα *EditForm* component το περιεχόμενο του οποίου ορίζεται από την HTML που αντιστοιχεί στην παράμετρο *ChildContent* και περικλείεται μεταξύ των ετικετών **<ModalDialog> </ModalDialog>**, τις οποίες εισάγουμε σε μία σελίδα ή σε ενα άλλο component για να δηλώσουμε ένα ModalDialog. Σε αυτό το σημείο κρίνεται χρήσιμη μία διευκρίνηση της ορολογίας που χρησιμοποιούμε. Με τον όρο λοιπόν σελίδα εννοούμε ένα component (αφού πρόκειται για .razor αρχείο) στο οποίο οδηγούμαστε μέσω του routing μηχανισμού του Blazor και επομένως περιέχει στην αρχή του ένα ή περισσότερα ***@page*** *razor-directives*. Δεν είναι λοιπόν όλα τα components σελίδες αλλά όλες οι σελίδες είναι components. Επίσης να διευκρινήσουμε στο σημείο αυτό την κατάληξη *.razor.cs* του αρχείου με τον C# κώδικα, που είδαμε στην παραπάνω εικόνα, ο οποίος συνδέεται με το ModalDialog.razor αρχείο. Γενικά μπορούμε να συμπεριλάβουμε τον κώδικα C# που αφορά κάποιο component στο ίδιο αρχείο (.razor) εντός ενός block, που ακολουθεί τον HTML κώδικα του component, το οποίο σημειώνουμε με το ***@code*** *razor-directive* (@code{ … }). Εναλλακτικά μπορούμε να δημιουργήσουμε ένα .cs αρχείο (αρχείο κλάσης) με όνομα <component-name>.razor το οποίο θα περιέχει τον C# κώδικα. Επιπλέον, επειδή όπως έχουμε επισημάνει, κάθε component αντιστοιχεί σε μία κλάση (δηλώνει δηλαδή έναν τύπο), πρέπει να συμπεριλάβουμε στην δήλωση της κλάσης του αρχείου που δημιουργήσαμε, η οποία έχει το ίδιο όνομα με την κλάση στην οποία αντιστοιχεί το component, το keyword *partial*.

Το ModalDialog δέχεται ως παράμετρο μια μέθοδο Submit, η οποία καλείται όταν η φόρμα γίνεται submit μέσω του κουμπιού που φέρει το χαρακτηριστικό type=”submit” και εφόσον το μοντέλο της φόρμας (τύπου *TModel*) πληρεί τους validation κανόνες. Υπενθυμίζουμε ξανά ότι αυτό συμβαίνει λόγω του ότι έχουμε χρησιμοποιήσει την *OnValidSubmit* παράμετρο του *EditForm* component. Να τονίσουμε ότι χρειάζεται όλα τα *button* στοιχεία που εισάγονται ως HTML μέσω της ChildContent παραμέτρου να φέρουν το χαρακτηριστικό *type=”button”* διότι η default τιμή του χαρακτηριστικού αυτού στην πλειονότητα των browsers είναι *submit*. Η κλάση ModalDialog ορίζει τις public μεθόδους *Show* και *Hide* με τις οποίες μπορούμε να εμφανίζουμε και να εξαφανίζουμε τον διάλογο. Η μέθοδος *Show* δέχεται μία παράμετρο μέσω της οποίας αρχικοποιείται η μεταβλητή *_editContext* που χρησιμοποιείται ως τιμή της παραμέτρου *EditContext* του *EditForm* component. Μέσω του *EditContext* τύπου μπορούμε να καλέσουμε κάποιες built-in μεθόδους, όπως η `NotifyFieldChanged` που καλείται μέσω της public μεθόδου *ReValidate*, ώστε να ανανεώσουμε τα validation μηνύματα που χαρακτηρίζουν ως invalid κάποια πεδία της φόρμας, όταν κάποιο από αυτά πάψει να ισχύει.

Στη συνέχεια βλέπουμε το περιεχόμενο του ModalDialog component της σελίδας DiagnosticCenters.razor. Παρατηρούμε ότι σε αυτό περιέχονται δύο components, το *CustomSelect* και το *AddressField*. Το *CustomSelect* είναι ένα HTML *select* στοιχείο, που

εξυπηρετεί στην επιλογή μίας τιμής από μία λίστα. Στην προκειμένη περίπτωση επιλέγουμε αρχικά μία πόλη για να εμφανιστεί στη συνέχεια ένα δεύτερο *select* από το οποίο επιλέγουμε την περιοχή με την οποία συνδέεται το διαγνωστικό κέντρο που δημιουργούμε ή τροποποιούμε. Το *AddressField* component περιλαμβάνει όλα τα στοιχεία της διεύθυνσης του κέντρου (οδός, αριθμός, ταχυδρομικός κώδικας), ώστε να μπορούμε εύκολα να συμπεριλάβουμε αυτή την πληροφορία όπου αλλού εμφανίζεται.

```
<ModalDialog @ref="_modal" TModel="DiagnosticCenterModel" Model="_model"
             Header="@(_model.Id == 0 ? "Add" : "Edit")"
             Submit="Submit">

    <div class="form-group row">
        <label class="col-sm-4 col-lg-3 col-form-label required">Name</label>
        <div class="col-sm-8 col-lg-9">
            <InputText class="form-control" @bind-Value="@_model.Name" />
        </div>
    </div>

    @if (_cities != null)
    {
        <div class="form-group row">
            <label class="col-sm-4 col-lg-3 col-form-label required">City</label>
            <div class="col-sm-8 col-lg-9">
                <CustomSelect IsRequired="_model.Id != 0" Prompt="Select City" NoValue="0" Value="_model.City?.Id ?? 0"
                              Items="@_cities"
                              ItemChanged="OnModelCityChanged"
                              ValueFunc="x => x.Id" LabelFunc="x => x.Name" />
                <ValidationMessage For="@(() => _model.City)" />
            </div>
        </div>
        @if (_model.City != null)
        {
            <div class="form-group row">
                <label class="col-sm-4 col-lg-3 col-form-label required">City</label>
                <div class="col-sm-8 col-lg-9">
                    <CustomSelect IsRequired="_model.Id != 0" Prompt="Select District" NoValue="0" Value="_model.Burg?.Id ?? 0"
                                  Items="@_model.City.Districts"
                                  ItemChanged="OnModelDistrictChanged"
                                  ValueFunc="x => x.Id" LabelFunc="x => x.Name" />
                    <ValidationMessage For="@(() => _model.Burg)" />
                </div>
            </div>
        }
    }

    <AddressField Address="@_model.Address" />

    <div class="form-group row">
        <label class="col-sm-4 col-lg-3 col-form-label required">Phone</label>
        <div class="col-sm-8 col-lg-9">
            <InputText class="form-control" @bind-Value="@_model.Phone" />
            <ValidationMessage For="@(() => _model.Phone)" />
        </div>
    </div>

    <div class="form-group row">
        <label class="col-sm-4 col-lg-3 col-form-label required">Email</label>
        <div class="col-sm-8 col-lg-9">
            <InputText class="form-control" @bind-Value="@_model.Email" />
            <ValidationMessage For="@(() => _model.Email)" />
        </div>
    </div>
</ModalDialog>
```

$2.2.2 – 3 – DiagnosticCenters.razor – ModalDialog component

Αν παρατηρήσουμε προσεκτικά την εικόνα, στην πρώτη γραμμή της δήλωσης του *ModalDialog* βλέπουμε το *attribute* **@ref**. Με αυτό τον τρόπο αποκτάμε μία αναφορά στο component, που περνιέται στην μεταβλητή *_modal* μέσω της οποίας έχουμε πρόσβαση στις public μεθόδους και μεταβλητές του. Για να εμφανίσουμε λοιπόν στον χρήστη τον διάλογο έχουμε μία μέθοδο στην οποία καλούμε την Show ως εξής:

```
private void OnModalShow(DiagnosticCenter selected = null)
{
        _model = Mappings.Mapper.Map(selected);
        if(selected != null)
                _model.City = _cities.SingleOrDefault(x =>
                        x.Id == selected.Burg.City.Id);
        _modal.Show(_model);
}
```

2.2.3 Ανάλυση της λειτουργικότητας της σελίδας

Όλα τα blazor-components είναι απόγονοι της κλάσης ***ComponentBase*** η οποία περιέχει διάφορες μεθόδους (*lifecycle methods*) τις οποίες μπορούμε να κάνουμε *override*. Για παράδειγμα την στιγμή που οδηγούμαστε σε μία σελίδα, αυτή δημιουργείται για να προβληθεί στον χρήστη και τότε εκτελείται η *OnInitialized* (ή *OnInitializedAsync*) μεθοδος, στην οποία αρχικοποιούμε όλες τις απαραίτητες -για την ορθή λειτουργία της- μεταβλητές.

```
protected override async Task OnInitializedAsync()
{
_state = new PageState();
await ManagementService.DoWork(async dbContext =>
{
       _cities = await dbContext.Cities
              .Select(x => new City
              {
                     Id = x.Id,
                     Name = x.Name,
                     Districts = x.Districts.Select(y => new District
                     {
                            Id = y.Id,
                            Name = y.Name,
                            City = new City
                            {
                                Id = x.Id,
                                Name = x.Name
                            }
                      }).ToList()
                }).ToListAsync();

       var loggedInUser = (await _authStateTask).User;
       if(loggedInUser.IsInRole(Role.DoctorRoleKey) ||
          loggedInUser.IsInRole(Role.StaffRoleKey))
       {
              var user = await dbContext.Users.SingleOrDefaultAsync(x =>
                                  x.Email == loggedInUser.Identity.Name ||
                                  x.UserName == loggedInUser.Identity.Name);
              _loggedInAsStaffQuery = dbContext =>
                        dbContext.DiagnosticCenterUsers
                                 .Where(x => x.User.Id == user.Id)
                                 .Select(x => x.DiagnosticCenter);
       }
});
await Refresh();
}
```

$2.2.3 – 1 – OnInitializedAsync lifecycle-method of DiagnosticCenters.razor

Κατά τη δημιουργία λοιπόν της σελίδας εκτελείται η παραπάνω μέθοδος όπου αρχικοποιείται η μεταβλητή *_cities* η οποία είναι μία λίστα με τις πολείς που υπάρχουν στη βάση, είναι δηλαδή τύπου *List<City>*. Η κλάση *City* περιέχει μία λίστα με τις περιοχές της κάθε πόλης, διαθέτει δηλαδή ένα μέλος 'Districts' με τύπο *List<District>*. Από την άλλη η κλάση *District* διαθέτει ένα μέλος 'City' με τύπο *City*. Αυτή η σχεδίαση μας βοηθά στο να ανακτήσουμε εύκολα τις περιοχές μίας πόλης μέσω του μέλους της 'Districts'

πραγματοποιώντας μόνο μία κλήση στη βάση. Πιο συγκεκριμένα, εάν δεν είχαμε συμπεριλάβει στην κλάση *City* το μέλος 'Districts' και θέλαμε να ανακτήσουμε μία πόλη με ενα συγκεκριμένο *Id* μαζί με τις περιοχές της, θα έπρεπε:

A. Να κάνουμε μία κλήση στη βάση για να ανακτήσουμε την πόλη βάσει του Id:
```
var city = dbContext.Cities.SingleOrDefault(x => x.Id == someId);
```

B. Να κάνουμε άλλη μία κλήση για να ανακτήσουμε τις περιοχές με τις οποίες αυτη η πόλη συνδέεται:
```
var cityDistricts = dbContext.Districts.Where(x => x.City.Id == city.Id).ToList();
```

Συμπεριλαμβάνοντας ως μέλος της κλάσης *City* μία λίστα για τις περιοχές της επιτυγχάνουμε το σκοπό μας ως εξής:

```
var cityWithDistricts = dbContext.City.Include(x => x.Districts)
                                      .SingleOrDefault(x => x.Id == someId);
```

Η σχέση μεταξύ City και District είναι ένα παράδειγμα "ένα προς πολλά" σχέσης.

Στη συνέχεια ορίζουμε την μεταβλητή τύπου *ClaimsPrincipal*, *loggedInUser* την οποία αρχικοποιούμε μέσω της μεταβλητής *_authState* η οποία ορίζεται ως εξής: `[CascadingParameter] Task<AuthenticationState> _authStateTask { get; set; }` και είναι διαθέσιμη μέσω του *CascadingAuthenticationState* του App.razor component (εικόνα $2.1 - 1)

Ας γράψουμε πιο αναλυτικά τα βήματα για την αρχικοποίηση του *loggedInUser*

```
AuthenticationState authState = await _authStateTask;

ClaimsPrincipal loggedInUser = authState.User;
```

Σημείωση Με το keyword *await* αναμένουμε το αποτέλεσμα της ασύγχρονης εκτέλεσης ενος *Task*. Με τον όρο "ασύγχρονη" εννοούμε ότι ο κώδικας εκτελείται σε ένα άλλο ***thread*** προκειμένου το user-interface μίας εφαρμογής να είναι responsive και να μην δίνεται η αίσθηση στον χρήστη ότι η ιστοσελίδα έχει κολλήσει μέχρι να ολοκληρωθεί μία διαδικασία. Για να χρησιμοποιήσουμε το *await* θα πρέπει η υπογραφή της μεθόδου όπου θα εμφανίζεται να φέρει το keyword *async.* Όταν χρησιμοποιούμε το *await* για την ασύγχρονη εκτέλεση ενός *Task<T>*, το αποτέλεσμα που θα λάβουμε θα είναι τύπου *T*.

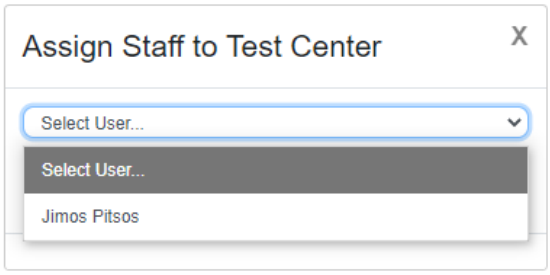
Στη συνέχεια μέσω του *ClaimsPrincipal* ελέγχουμε τον ρόλο του συνδεδεμένου χρήστη και εάν είναι 'doctor' ή 'staff' αρχικοποιούμε την μεταβλητή *_loggedInAsStaffQuery* μέσω της οποίας η ανάκτηση των διαγνωστικών κέντρων που προβάλλονται στον χρήστη γίνεται από τον πίνακα *DiagnosticCenterUsers* από όπου φιλτράρουμε τις εγγραφές βάσει του *Id* του συνδεδεμένου χρήστη και ύστερα επιλέγουμε από αυτές τα διαγνωστικά κέντρα. Η σχέση μεταξύ των χρηστών και των διαγνωστικών κέντρων χαρακτηρίζεται ως "πολλά προς πολλά".

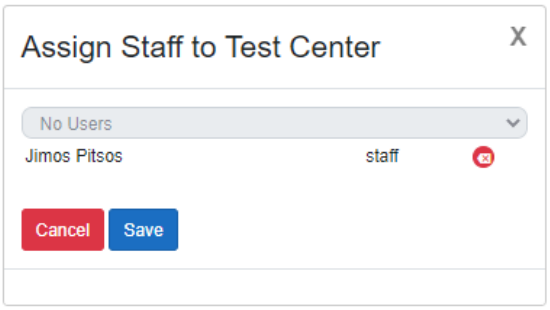
Για να συνδέσουμε κάποιον χρήστη (με ρόλο ‘doctor’ ή ‘staff’) με κάποιο διαγνωστικό κέντρο πρέπει πρώτα να συνδεθούμε ως ‘admin’ ώστε να έχουμε διαθέσιμη την λειτουργικότητα αυτή. Τότε κάνουμε κλικ στο κουμπί με το εικονίδιο του παπύρου στη γραμμή του διαγνωστικού κέντρου με το οποίο θέλουμε να γίνει η σύνδεση, για να εμφανιστεί ο διάλογος με τη λίστα όλων των χρηστών που δεν συνδέονται με το επιλεγμένο διαγνωστικό κέντρο. Για να μπορεί να εξυπηρετηθεί καλύτερα αυτή η διαδικασία θα πρέπει να προσθέσουμε στον χρήστη την πληροφορία μίας πόλης ώστε να ανακτούμε τους χρήστες όχι μόνο βάσει της απουσίας σύνδεσης τους με ένα διαγνωστικό κέντρο αλλά και βάσει της συμφωνίας της πόλης ενός χρήστη με την πόλη όπου βρίσκεται η περιοχή του διαγνωστικού κέντρου. Ας παρακάμψουμε όμως αυτήν την τεχνική λεπτομέρεια για να δούμε πως ανακτούμε τους χρήστες που δεν συνδέονται με κάποιο διαγνωστικό κέντρο. Στο κλικ λοιπόν αυτού του κουμπιού καλείται η παρακάτω μέθοδος:

```
private async Task OnAssignUser(DiagnosticCenter selected)
{
       await ManagementService.DoWork(async dbContext =>
       {
              var availableUsers = await dbContext.Users
                     .Include(u => u.UserRole.Role)
                     .Where(u => u.UserRole != null)
                     .Where(u => u.UserRole.Role.Name == Role.StaffRoleKey ||
                                 u.UserRole.Role.Name == Role.DoctorRoleKey)
                     .Where(u => dbContext.DiagnosticCenterUsers
                                          .Where(x => x.DiagnosticCenter.Id ==
                                                                 selected.Id)
                                          .All(x => x.User.Id != u.Id))
                     .ToListAsync();
              _assignedUsersModel = new DiagnosticCenterUsersAssignedModel
              {
                     Center = new DiagnosticCenter
                     {
                         Id = selected.Id,
                         Name = selected.Name
                     },
                     AvailableUsers = availableUsers
              };
       });
       _assignToCenterModal.Show(_assignedUsersModel);
}
```

Φιλτράρουμε λοιπόν τους χρήστες πρώτα βάσει του ρόλου τους και στη συνέχεια ζητάμε για κάθε χρήστη από τον πίνακα *AspNetUsers*, οι εγγραφές του πίνακα *DiagnosticCenterUsers* που σχετίζονται με το επιλεγμένο διαγνωστικό κέντρο, να μην περιλαμβάνουν το *Id* του χρήστη. Κατόπιν κατασκευάζουμε ένα μοντέλο τύπου *DiagnosticCenterUsersAssignedModel*, που περιέχει τη λίστα των χρηστών που δεν συνδέονται ήδη με το επιλεγμένο διαγνωστικό κέντρο, και που περνάμε στην μέθοδο εμφάνισης του ModalDialog.

```
private class DiagnosticCenterUsersAssignedModel
{
        public DiagnosticCenter Center { get; set; }
        public List<User> AvailableUsers { get; set; }
        public List<User> AssignedUsers { get; set; } = new List<User>();

        public List<User> RestUsers =>
              AvailableUsers.Except(AssignedUsers).ToList();
}
```

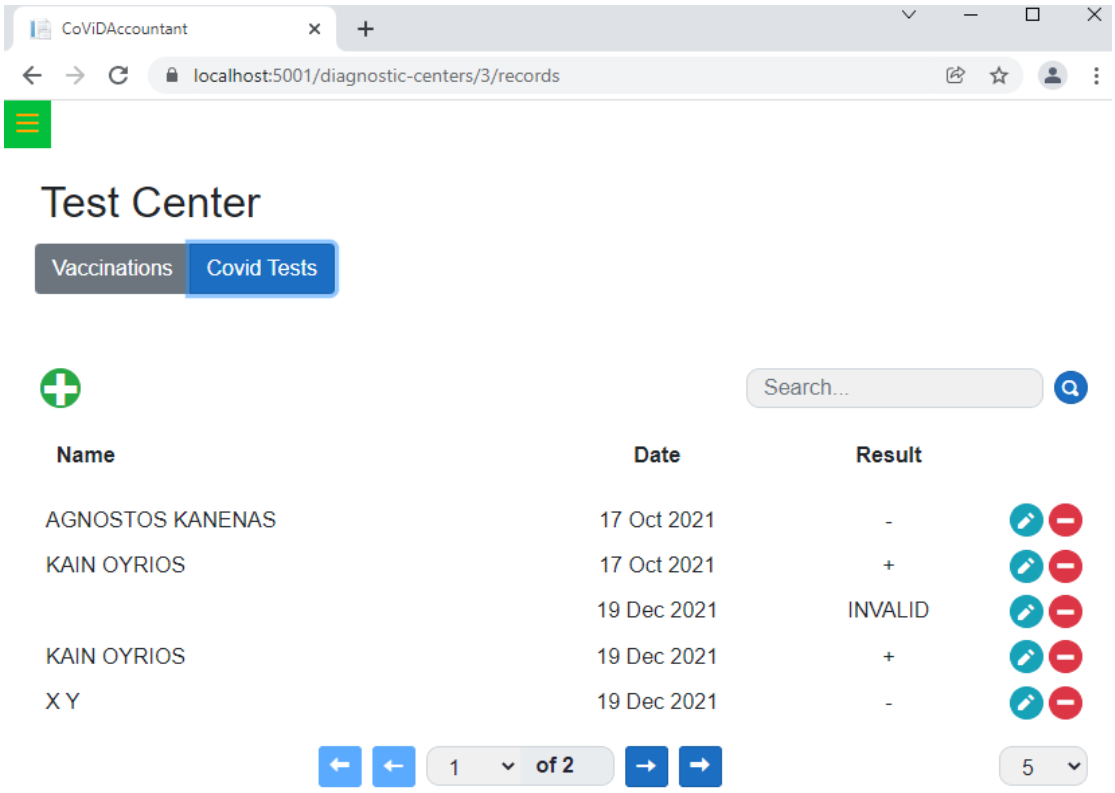




```
<ModalDialog @ref=_assignToCenterModal
             TModel="DiagnosticCenterUsersAssignedModel"
             Model="_assignedUsersModel"
             Header=@($"Assign Staff to {_assignedUsersModel.Center?.Name}")
             Submit="Assign">

    <CustomSelect Prompt="@(_assignedUsersModel.RestUsers.Any()
                                ? "Select User..." : "No Users")"
                  NoValue="0" Value="0"
                  Items="@_assignedUsersModel.RestUsers"
                  ItemChanged="OnAssignUser"
                  ValueFunc="x => x.Id" LabelFunc="x => x.Name"
                  Enabled="_assignedUsersModel.RestUsers.Any()" />
    <div class="form-group row mx-auto">
        <div class="table-responsive">
            <table class="table">
                @foreach (var u in _assignedUsersModel.AssignedUsers)
                {
                    <tr>
                        <td>@u.Name</td>
                        <td>@u.UserRole.Role.Name </td>
                        <td>
                            <button class="btn btn-danger btn-circle btn-sm"
                                    type="button"
                                    @onclick="() =>
                                    _assignedUsersModel.AssignedUsers.Remove(u)">
                                <span class="oi oi-delete"></span>
                            </button>
                        </td>
                    </tr>
                }
            </table>
        </div>
    </div>
</ModalDialog>
```

$2.2.3 - 2 - ModalDialog component used for assigning staff to a diagnostic center (DiagnosicCenters.razor)

2.3 Η Σελίδα *DiagnosticCenterRecords.razor*



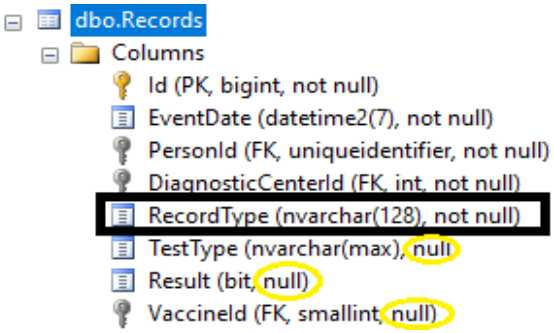
Μέσω της σελίδας *DiagnosticCenters.razor* έχουμε την δυνατότητα να πλοηγηθούμε στην σελίδα *DiagnosticCenterRecords.razor* όπου προβάλλονται πληροφορίες για τους εμβολιασμούς (κλάση *Vaccination*) αλλά και τα τεστ-ανίχνευσης (κλάση *CovidTest*) που έχουν πραγματοποιηθεί σε ένα συγκεκριμένο διαγνωστικό κέντρο. Η πληροφορία αυτή διατηρείται στον πίνακα *Records* της βάσης δεδομένων. Και οι δύο αναφερθείσες κλάσεις είναι τύπου Record:

```
namespace DbDesign.Entities
{
	public abstract class Record : Entity<long, Record>
	{
		public DateTime EventDate { get; set; }
		public Person Person { get; set; }
		public DiagnosticCenter DiagnosticCenter { get; set; }

		public abstract string GetRecordType();
	}
}

public class Vaccination : Record
{
	public Vaccine Vaccine { get; set; }

	public override string GetRecordType() => nameof(Vaccination);
}
```

```
public class CovidTest : Record
{
        public string TestType { get; set; }
        public bool? Result { get; set; }

        public override string GetRecordType() => nameof(CovidTest);
}
```

2.3.1 Table-Per-Hierarchy (TPH) mapping

Ο προκαθορισμένος τρόπος με τον οποίο το EF Core αντιστοιχίζει μία ιεραρχία τύπων με το μοντέλο της βάσης δεδομένων, είναι ένας πίνακας στον οποίο η κάθε εγγραφή απαρτίζεται από όλη συνολικά την πληροφορία των κλάσεων της ιεραρχίας. Επιπλέον δημιουργείται στον πίνακα αυτό, μία στήλη που καλείται “discriminator” μέσω της οποίας προσδιορίζεται ο τύπος στον οποίο αντιστοιχεί η κάθε εγγραφή. ‘Ετσι για την παραπάνω ιεραρχία δημιουργείται ο πίνακας *Records* με τις εξής στήλες:



Να τονίσουμε ότι καθώς ο πίνακας που δημιουργείται στη βάση μοιράζεται από διαφορετικούς τύπους που κληρονομούν από έναν κοινό (τον *Record* εν προκειμένω), δεν έχουμε την δυνατότητα να ορίσουμε ένα μέλος ή μία σχέση ως υποχρεωτικό/-ή. Δεν μπορούμε για παράδειγμα στον ορισμό του πίνακα *Records* να απαιτήσουμε η σχέση μεταξύ των *Vaccinations* (υποσύνολο του πίνακα *Records* όπου η discriminator-στήλη *RecordType* έχει τιμή “Vaccination”) και του πίνακα *Vaccines* να είναι υποχρεωτική, δηλαδή η στήλη *VaccineId* να μην επιτρέπει την τιμή *NULL*. Το ίδιο ισχύει και για τις στήλες *TestType* και *Result* που αντιστοιχούν στον τύπο *CovidTest*. Αντίθετα μπορούμε να χαρακτηρίσουμε ως υποχρεωτικά τα μέλη του abstract τύπου *Record* από τον οποίο κληρονομούν οι τύποι *Vaccination* και *CovidTest*.

2.3.2 *TabContainer.razor* component

```
@page "/diagnostic-centers/{DiagnosticCenterId:int}/records"
@*@implements ITabs*@
@using DbDesign.Entities
@using CoViDAccountant.Components

<h2 class="page-title">@_diagnosticCenter?.Name</h2>
<CascadingValue Value="this">
    <TabContainer @ref="_tabContainer">
        <Tab Sign="@VACCINATIONS_TAB">
            @if (_vaccinations != null)
            {
                <DataTable>...</DataTable>
            }
        </Tab>
        <Tab Sign="@COVIDTESTS_TAB">
            @if (_covidTests != null)
            {
                <DataTable>...</DataTable>
            }
        </Tab>
    </TabContainer>
</CascadingValue>
```

$2.3.2 – 1 – *DiagnosticCenterRecords.razor*

Στη σελίδα αυτή χρησιμοποιούμε το component *TabContainer.razor* το οποίο μας εξυπηρετεί στο να εναλλάσουμε την προβαλλόμενη πληροφορία ανάλογα με τον ποιος τύπος *Record* είναι επιλεγμένος. Βλέπουμε παρακάτω τον κώδικα του *TabContainer* component και τον τρόπο με τον οποίο χρησιμοποιείται στη σελίδα *DiagnosticCenterRecords.razor*. Στο *TabContainer.razor* υπάρχει ένα μέλος *Parent* που συνοδεύεται από το attribute ***[CascadingParameter]*** και που είναι τύπου *ITabs*, επομένως πρέπει να υλοποιεί το παρακάτω *interface*.

```
public interface ITabs
{
        string ActiveTabSign { get; set; }
        Task LoadData(string selectedTabSign);
}
```

$2.3.2 – 2 – *ITabs.cs*

```
public partial class DiagnosticCenterRecords : ITabs
{
        // ITabs
        public string ActiveTabSign { get; set; }
        public async Task LoadData(string selectedTab) { ... }
        ...
}
```

$2.3.2 – 3 – *DiagnosticCenterRecords.razor.cs*

```
<div class="btn-group mb-3" role="group">
@foreach (var tab in Tabs)
{
        <button type="button" class="btn @GetButtonClass(tab)"
                @onclick=@(() => ActivatePage(tab))>
            @tab.Sign
         </button>
}
</div>

<CascadingValue Value="this">
    @ChildContent <!-- This is the markup that is between the starting and
                        closing tags of TabContainer component -->
</CascadingValue>

@code
{
        [CascadingParameter] private ITabs Parent { get; set; }
        [Parameter] public RenderFragment ChildContent { get; set; }

        public Tab ActiveTab { get; set; }
        public List<Tab> Tabs = new List<Tab>();

        internal void AddTab(Tab tab)
        {
                Tabs.Add(tab);
                // first tab to be added will be active
                if (Tabs.Count == 1)
                {
                        ActiveTab = tab;
                        Parent.ActiveTabSign = tab.Sign;
                }
        }

        private string GetButtonClass(Tab tab)
        {
                return tab == ActiveTab ? "btn-primary" : "btn-secondary";
        }

        private async Task ActivatePage(Tab tab)
        {
                if (Parent.ActiveTabSign == tab.Sign)
                {
                        return;
                }
                ActiveTab = tab;
                Parent.ActiveTabSign = tab.Sign;
                await Parent.LoadData(tab.Sign);
        }
}
```

$2.3.2 – 4 – *TabContainer.razor*

Το attribute *[CascadingParameter]* δίνει την δυνατότητα σε ένα component να λάβει μία τιμή η οποία παρέχεται από ένα “υψηλότερου επιπέδου” component. Αυτό επιτυγχάνεται

μέσω του *CascadingValue* component (built-in), το οποίο μεταδίδει προς όλα τα (χαμηλότερου επιπέδου) components που περικλείει, μία τιμή που περνιέται στην παράμετρό του *Value*. Έτσι τα χαμηλότερου επιπέδου components μπορούν να λάβουν αυτή την τιμή εφόσον περιέχουν ένα μέλος του ίδιου τύπου, που συνοδεύεται από το attribute *[CascadingParameter]*. 'Ετσι, η τιμή *this* στην εικόνα "$2.3.2 – 1", είναι μία αναφορά στην ίδια την σελίδα *DiagnosticCenterRecords.razor* (που είναι τύπου *DiagnosticCenterRecords*), που υλοποιεί το *interface ITabs* (Εικόνα $2.3.2 - 3). Το *TabContainer* component λοιπόν, λαμβάνοντας μία αναφορά τύπου *DiagnosticCenterRecords* έχει πρόσβαση σε όλα τα *public* μέλη του τύπου αυτού. Οπότε έχει τη δυνατότητα να καλέσει την μέθοδο *LoadData* του *ITabs* ενώ επίσης μπορεί να ανακτήσει ή να τροποποιήσει την τιμή της *ActiveTabSign* μεταβλητής.

Σημείωση: Τα μέλη μίας κλάσης που φέρουν το attribute *[CascadingParameter]* μπορεί να είναι *private* σε αντίθεση με τα μέλη που φέρουν το attribute *[Parameter]* που πρέπει να είναι *public*.

Στην εικόνα "$2.3.2 – 4" όπου φαίνεται ο κώδικας του *TabContainer* component βλέπουμε ότι χρησιμοποιείται και εκεί το *CascadingValue* component, το οποίο παρέχει στα components που περιέχει, μία αναφορά στο *TabContainer* component. Έτσι, τα δύο *Tab.razor* components από τα οποία αποτελείται το component *TabContainer,* λαμβάνουν μία αναφορά σε αυτό μέσω του μέλους τους *Container* όπως βλέπουμε και στην παρακάτω εικόνα.

```
@if (Container.ActiveTab.Sign == Sign)
{
        @ChildContent
}

@code
{
    [CascadingParameter] private TabContainer Container { get; set; }

    [Parameter] public RenderFragment ChildContent { get; set; }

    [Parameter] public string Sign { get; set; }

    protected override void OnInitialized()
    {
        if (Container == null)
            throw new System.ArgumentNullException(nameof(Container),
                                "Tab component must exist within a tab control");
        base.OnInitialized();
        Container.AddTab(this);
    }
}
```

$2.3.2 – 5 – *Tab.razor*

Σημείωση: Η λίστα *Tabs* του *TabContainer* διαμορφώνεται μέσω της *AddTab* μεθόδου του, που καλείται από τα *Tab* components που περιέχει, τη στιγμή της δημιουργίας τους, δηλαδή στην *OnInitialized* μέθοδό τους.

2.3.3 Μερικές πολυπλοκότερες λειτουργικότητες

‘Οταν πραγματοποιούμε μία ενέργεια για να διαγράψουμε μία εγγραφή από τη βάση, εμφανίζεται ένας διάλογος ο οποίος ζητά να επιβεβαιώσουμε την πρόθεσή μας. Αυτό το επιτυγχάνουμε μέσω του ConfirmDialog.razor component το οποίο βρίσκεται στο App.razor component.

```
@inject Services.UIService UI

<CascadingAuthenticationState>

    <Router AppAssembly="@typeof(Program).Assembly">
        @*PreferExactMatches="@true">*@
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
                <NotAuthorized>
                    <CoViDAccountant.Components.NotAuthorized />
                </NotAuthorized>
                <Authorizing>
                    <h1>Authenticating...</h1>
                </Authorizing>
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <LayoutView Layout="@typeof(MainLayout)">
                <p>Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>

</CascadingAuthenticationState>

<CoViDAccountant.Components.Dialogs.ConfirmDialog @ref="_confirmDlg" />
```

@2.3.3 – 1 – App.razor

‘Οπως βλέπουμε στην εικόνα $2.3.3-2, το component ConfirmDialog αποτελείται από το component ModalDialog του οποίου η παράμετρος SystemicDialog έχει την τιμή true. ‘Ετσι, αν ανατρέξουμε στην εικόνα $2.2.2–1 θα δούμε ότι, κατά αυτό τον τρόπο παραλείπουμε το EditForm component του ModalDialog και συνεπώς καθίσταται περιττή η παράμετρος OnSubmit, που είναι κάποια μέθοδος που καλείται με την υποβολή της φόρμας. Το ConfirmDialog component δέχεται μία παράμετρο η οποία είναι τύπου Func<Task>, δηλαδή αποτελεί μία μέθοδο χωρίς παραμέτρους που επιστρέφει έναν τύπο Task. Μπορούμε δηλαδή να ταυτίσουμε αυτόν τον τύπο με μία μέθοδο όπως η ακόλουθη:

```
private async Task Submit()
{
       await ... // code
}
```

Εν συντομία την μέθοδο αυτή μπορούμε να την διατυπώσουμε και ως εξής:

```
async () =>
{
       await ... // code

}
```

Σε αυτή την περίπτωση αποτελεί μία ανώνυμη μέθοδο (anonymous method). Τα keywords async/await είναι προαιρετικά, για την περίπτωση όπου θέλουμε να περιμένουμε να ολοκληρωθεί η εκτέλεση κάποιου ασύγχρονου κώδικα. Στην περίπτωση που ο κώδικας που εκτελούμε δεν είναι ασύγχρονος, επειδή πρέπει να επιστρέφεται ένας τύπος *Task*, στο τέλος της μεθόδου πρέπει να υπάρχει η δήλωση *return Task.CompletedTask;*

```
<ModalDialog @ref=_modal TModel="object"
             Header="Confirm Action"
             SystemicDialog="true"
             CssStyle="background-color:yellow">
    <p>@(new MarkupString(Message))</p>

    <br />
    <div class="text-right">
        <button class="btn btn-danger" type="button" @onclick="_modal.Hide">
            No
        </button>
        <button class="btn btn-success" type="button" @onclick="Confirm">
            Yes
        </button>
    </div>
</ModalDialog>

@code {
    [Parameter] public string Message { get; set; }
    [Parameter] public Func<Task> OnConfirm { get; set; }

    private ModalDialog<object> _modal;

    private async Task Confirm()
    {
        try
        {
            if (OnConfirm != null)
                await OnConfirm();
        }
        finally
        {
            _modal.Hide();
        }
    }

    public void Show(string message, Func<Task> onConfirm)
    {
        Message = message;
        OnConfirm = onConfirm;
        _modal.Show(null);
    }
}
```

$2.3.3 – 2 – *ConfirmDialog.razor*

Η χρήση του *ConfirmDialog* component γίνεται μέσω της κλάσης *UIService*, την οποία έχουμε καταχωρήσει ως application-service στην Startup κλάση της εφαρμογής, και παρέχει τις δύο παρακάτω *public* μεθόδους:

```
public class UIService
{
        private ConfirmDialog _confirmDialogRef;

        public void SetConfirmDialog(ConfirmDialog confirmDialogRef)
        {
            _confirmDialogRef = confirmDialogRef;
        }

        public void ShowConfirmDialog(string message, Func<Task> onConfirm)
        {
            _confirmDialogRef?.Show(message, onConfirm);
        }
}
```

$2.3.3 – 3 – CoViDAccountant\Services\UIService.cs

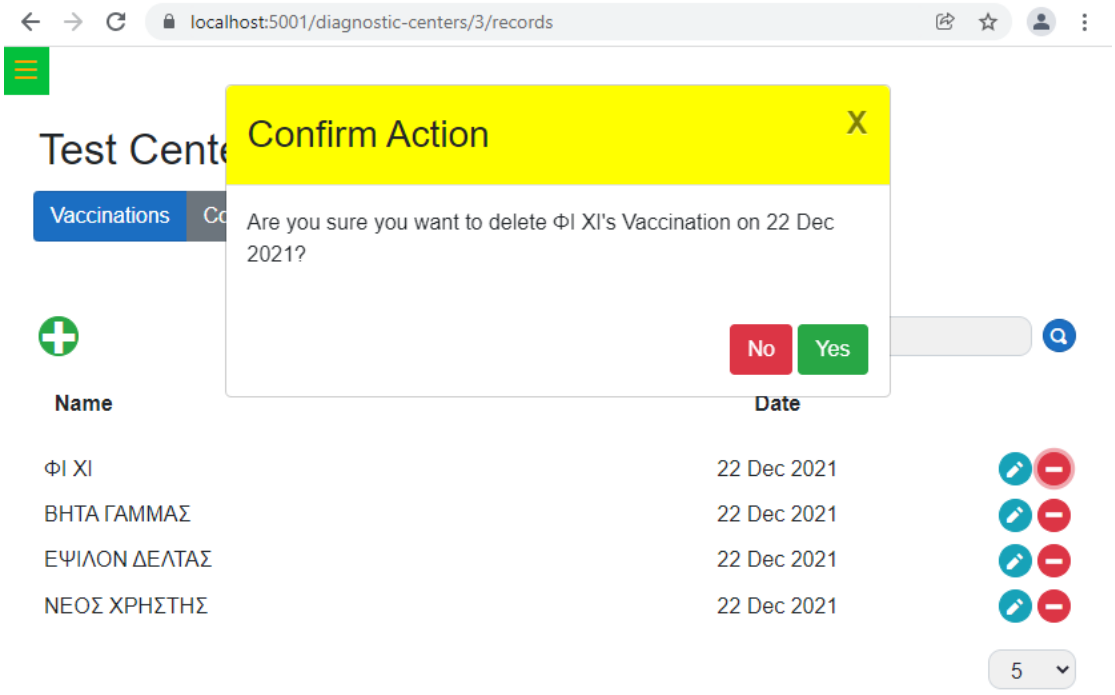
Μέσω της μεθόδου *ShowConfirmDialog* εμφανίζουμε τον διάλογο επιβεβαίωσης, την στιγμή του on-click event του κουμπιού της διαγραφής, μέσω του μέλους "UI" της σελίδας που είναι τύπου *UIService*. Το μέλος αυτό συνοδεύεται από το attribute *[Inject]* ώστε να αρχικοποιείται αυτόματα, μέσω του dependency-injection μηχανισμού, όποτε δημιουργείται μία σελίδα *DiagnosticCenterRecords*. Η μέθοδος *SetConfirmDialog* καλείται στην *OnAfterRender* – *lifecycle* – μέθοδο του *App* component, η οποία κληρονομείται από την κλάση *ComponentBase* και καλείται μετά την δημιουργία του (μετά την *OnInitialized* μέθοδο) για να διαμορφώσει την αναφορά του *_confirmDialogRef*.

```
public partial class App
{
        [Inject] private UIService UI { get; set; }

        private Components.Dialogs.ConfirmDialog _confirmDlg;

        protected override void OnAfterRender(bool firstRender)
        {
            base.OnAfterRender(firstRender);
            if (firstRender)
            {
                UI.SetConfirmDialog(_confirmDlg);
            }
        }
}
```

$2.3.3 – 4 – *App.razor.cs*

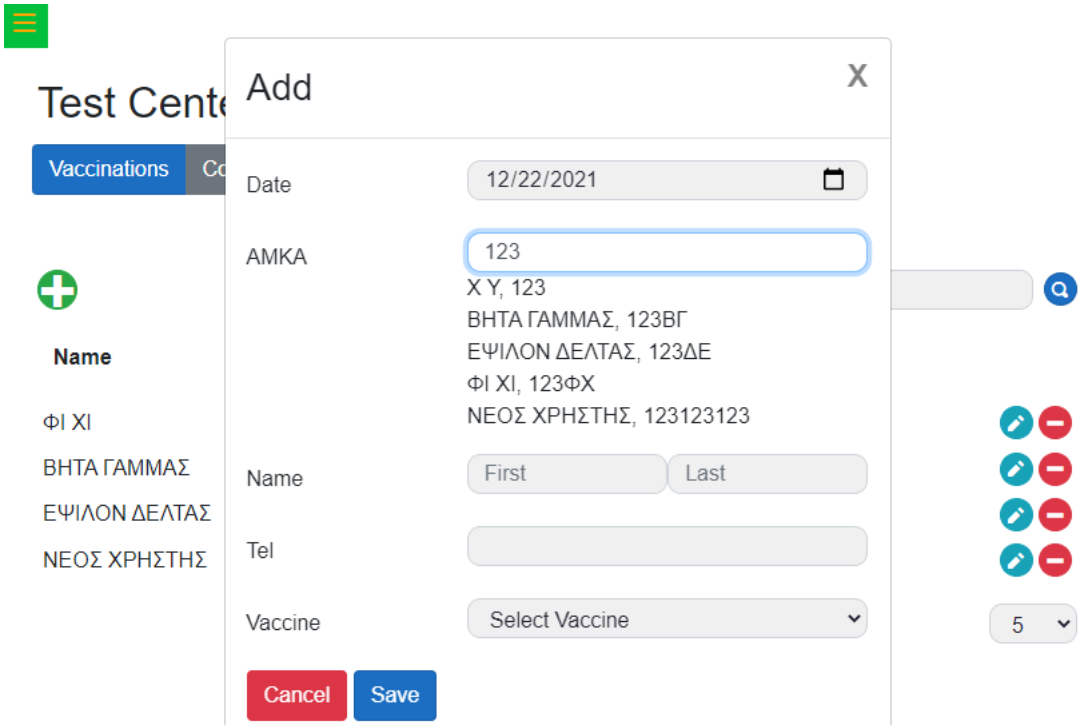
Σημείωση: Όταν έχει ολοκληρωθεί η δημιουργία του *App.razor* component και άρα και του *ConfirmDialog* component που περιέχει και που φέρει το *@ref directive-attribute*, καλείται η *OnAfterRender* μέθοδος. Μέσω αυτής, περνιέται στην *UIService* κλάση η αναφορά

_*confirmDialog* που διαμορφώνεται μέσω του *@ref* όποτε ολοκληρώνεται η δημιουργία του *App.razor* component.



§2.3.3 – 5 – Επιβεβαίωση διαγραφής εγγραφής από τον πίνακα Records

Στη συνέχεια ακολουθεί μία γενική περιγραφή του *TypeAhead* component μέσω του οποίου μπορούμε και προβάλλουμε αποτελέσματα που περιέχουν ως substring το input του χρήστη. Συγκεκριμένα, κατά την εισαγωγή του ΑΜΚΑ της ιδιότητας “Person” του τύπου *Record*, εμφανίζουμε μία λίστα από εγγραφές τύπου *Person*, που η τιμή της ιδιότητας τους “ΑΜΚΑ” ξεκινάει με το input του χρήστη.



$2.3.3 – 6 – Αποτελέσματα αναζήτησης εγγραφών από τον πίνακα Persons

```
protected override void OnInitialized()
{
	_debounceTimer = new Timer
	{
		Interval = DebounceInterval,
		AutoReset = false
	};
	_debounceTimer.Elapsed += Search;

	Initialize();

	base.OnInitialized();
}

private void Initialize()
{
	SearchText = "";
	_isShowingSuggestions = false;
	_maskOn = Value != null;
}
```

$2.3.3 – 7 – *TypeAhead.razor.cs*

Στην *OnInitialized* μέθοδο του *TypeAhead.razor* component, που καλείται σε κάθε εμφάνιση του *ModalDialog* component, δημιουργείται ένα νέο αντικείμενο τύπου *Timer* το οποίο ανατίθεται στην μεταβλητή *_debounceTimer*. Για την δημιουργία του τύπου *Timer* ορίζουμε την ιδιότητά του *Interval*, ένα χρονικό δηλαδή διάστημα στο οποίο αναθέτουμε την τιμή που λαμβάνεται από την παράμετρο “DebounceInterval” που εκφράζει τον χρόνο σε

milliseconds (τύπου *int*). Όταν παρέλθει αυτό το διάστημα (από την στιγμή της έναρξης της μέτρησης, μέσω της μεθόδου *Start* της κλάσης *Timer*), ενεργοποιείται το *Elapsed* event στο οποίο καλείται η μέθοδος *Search*:

```
private async void Search(object sender, ElapsedEventArgs args)
{
	if (_searchText.Length < MinimumLength)
		return;

	_suggestions = (await SearchMethod.Invoke(_searchText,
					MaximumSuggestions)).ToArray();

	_isShowingSuggestions = true;
	await InvokeAsync(StateHasChanged);
}
```

Εφόσον το μήκος του input του χρήστη είναι επαρκές για την πραγματοποίηση αναζήτησης καλείται η μέθοδος *SearchMethod* η οποία ορίζεται παραμετρικά:

```
[Parameter]
public Func<string, int, Task<IEnumerable<TItem>>> SearchMethod { get; set; }
```

Δηλαδή αποτελεί μία μέθοδο με δύο παραμέτρους που επιστρέφει *Task<IEnumerable<TItem>>* που αναμένουμε να εκτελεστεί για να αναθέσουμε τα αποτελέσματα της αναζήτησης, που είναι τύπου *TItem* (τύπου *Person* στην περίπτωσή μας), στο μέλος *_suggestions* του *TypeAhead* component. Βλέπουμε παρακάτω την μέθοδο της σελίδας *DiagnosticCenterRecords.razor* που καλείται την ώρα του event.

```
private Task<IEnumerable<Person>> SearchPersons(string amka, int maxResults)
{
	return ManagementService.Execute(async dbContext =>
	{
		amka = amka.Trim();
		var results = await dbContext.Persons
			.Where(x => x.AMKA.StartsWith(amka))
			.Take(maxResults)
			.ToListAsync();
		if (results.Count == 0)
		{
			_model.Person = new Person { AMKA = amka };
			await InvokeAsync(StateHasChanged);
		}
		return results.AsEnumerable();
	 });
}
```

Αφού αναλύσαμε τον τρόπο που λειτουργεί η type-ahead αναζήτηση, ας δούμε πως ενεργοποιείται η αντίστροφη μέτρηση μέσω του πεδίου _debounceTimer. Το *TypeAhead* component περιέχει το παρακάτω HTML-input στοιχείο

```
<input @bind-value="SearchText"
       @bind-value:event="oninput"
       class="form-control @CssClass"
       autocomplete="off"
       type="text" />
```

Το στοιχείο αυτό φέρει τα attributes *@bind-value* και *@bind-value:event*, μέσω των οποίων η τιμή της ιδιότητας *SearchText* του component διαμορφώνεται κάθε φορά που ενεργοποιείται το on-input event του *<input />* στοιχείου, δηλαδή κάθε φορά που ο χρήστης πληκτρολογεί έναν χαρακτήρα εντός του στοιχείου.

```
private string SearchText
{
       get => _searchText;
       set
       {
              _searchText = value;
              if (value.Length == 0)
              {
                     _debounceTimer.Stop();
              }
              else
              {
                    _debounceTimer.Stop();
                    _debounceTimer.Start();
              }
       }
}
```

$2.3.3 – 8 – *SearchText* property with *getter-setter* methods

Η ιδιότητα *SearchText* συνδέεται με το πεδίο *_searchText* (backing-field) του component. Όταν δηλαδή ανακτούμε την τιμή της ιδιότητας *SearchText* λαμβάνουμε την τιμή του πεδίου *_searchText*, ενώ όποτε γίνεται κάποια τροποποίηση που αφορά στην τιμή της ιδιότητας *SearchText*, η νέα τιμή ανατίθεται στο πεδίο *_searchText*. Έτσι μέσω του on-input event καλείται η setter μέθοδος της ιδιότητας *SearchText* που ορίζεται μέσω του keyword *set*. Όποτε λοιπόν ο χρήστης πληκτρολογεί εντός του HTML input στοιχείου, επανεκκινείται η αντίστροφη μέτρηση για το χρονικό διάστημα που ορίζεται παραμετρικά μέσω της *DebounceInterval* παραμέτρου του *TypeAhead* component (2,5 sec).

Βιβλιογραφία

Razor Syntax - *https://docs.microsoft.com*

Blazor - *https://docs.microsoft.com* - *https://blazor-university.com*

Michael Washington (2020), Blazor Succintly

Jon P. Smith (2018), Entity Framework Core In Action

Joe Mayo (2015), C# Succintly

Jon Skeet (2019), C# In Depth