**Введение**

**Тема проекта:** Проект предназначен для анализа замеров артериального давления, сохранения и визуализаций вносимых данных, комментариями к ним.

**Цель:** У пожилого возраста есть ежедневные системные потребности в замерах артериального давления и их записями куда придется, без возможности их анализа. Цель расставить приоритеты поведения образа жизни в рамках дневных циклов.

**Задачи:**

1. В рамках проекта будет разработан веб-сервис на C# с применением шаблона паттерна MVC, для управления задачами с использованием фреймворка ASP.Net Core. Проект будет предоставлять API для создания, редактирования, удаления и получения информации о вносимых данных и их визуализацию в таблицах и виде графиков.
2. Создание и подключение к базе данных Mysql, с помощью Entity Framework Core, которое представляет ORM-решение.
3. Реализовать систему двухфакторной аутентификации.
4. Разработка дизайна и структуры сайта.
5. Клиентская часть будет реализовывать код в виде javascript.js с сочетанием vue .js библиотеки.

**Инструментами буду пользоваться следующими:**

* Программирования в среде Microsoft Visual Studio и VSC
* SDK для .NET / Net Core / Javascript / Vue.js / jquery / html / css bootstrap.
* Серверная часть приложение Net Core на языке c#.
* Клиентская Javascript / Vue.js / jquery / html / css / bootstrap.

**Что такое веб-приложение ASP.NET Core и его особенности.**

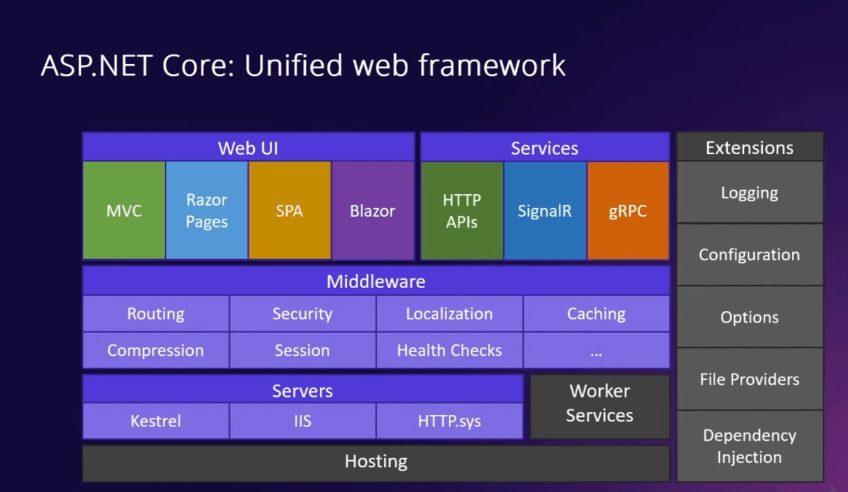
ASP.NET Core представляет технологию для создания веб-приложений на платформе .NET, развиваемую компанией Microsoft. В качестве языков программирования для разработки приложений на ASP.NET Core используются C# и F#.

История ASP.NET фактически началась с выходом первой версии .NET в начале 2002 года и с тех пор ASP.NET и .NET развивались параллельно: выход новой версии .NET знаменовал выход новой версии ASP.NET, поскольку ASP.NET работает поверх .NET. В то же время изначально ASP.NET была нацелена на работу исключительно в Windows на веб-сервере IIS (хотя благодаря проекту Mono приложения на ASP.NET можно было запускать и на Linux).

Однако 2014 год ознаменовал большие перемены, фактически революцию в развитии платформы: компания Microsoft взяла курс на развитии ASP.NET как кроссплатформенной технологии, которая развивается как opensource-проект. Данное развитие платформы в дальнейшем получило название **ASP.NET Core**, собственно, как ее официально именует Microsoft до сих пор. Первый релиз обновленной платформы увидел свет в июне 2016 года. Теперь она стала работать не только на Windows, но и на MacOS и Linux. Она стала более легковесной, модульной, ее стало проще конфигурировать, в общем, она стала больше отвечать требованиям текущего времени.

Текущая версия ASP.NET Core, которая собственно и будет охвачена в текущем руководстве, вышла вместе с релизом **.NET 8** в ноябре 2023 года.

ASP.NET Core теперь полностью является opensource-фреймворком. Все исходные файлы фреймворка доступны на github в репозитории <https://github.com/dotnet/aspnetcore/>.



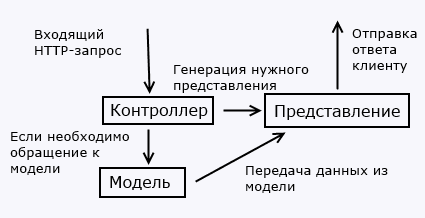
На самом верхнем уровне располагаются различные модели взаимодействия с пользователем. Это технологии построения пользовательского интерфейса и обработки ввода пользователя, как MVC, Razor Pages, SPA (Single Page Application - одностраничные приложения с использованием Angular, React, Vue) и Balzor. Кроме того, это сервисы в виде встроенных HTTP API, библиотеки SignalR или сервисов GRPC.

Все эти технологии базируются и/или взаимодействуют с чистым ASP.NET Core, который представлен прежде всего различными встроенными компонентами middleware - компонентами, которые применяются для обработки запроса. Кроме того, технологии высшего уровня также взаимодействуют с различными расширениями, которые не являются непосредственной частью ASP.NET Core, как расширения для логирования, конфигурации и т.д.

И на самом нижнем уровне приложение ASP.NET Core работает в рамках некоторого веб-сервера, например, Kestrel, IIS, библиотеки HTTP.sys.

Это вкратце архитектура платформы, теперь рассмотрим моделей разработки приложения ASP.NET Core:

* Прежде всего это **базовый ASP.NET Core**, который поддерживает все основные моменты, необходимые для работы современного веб-приложения: маршрутизация, конфигурация, логирования, возможность работы с различными системами баз данных и т.д.. В ASP.NET Core 6 в фреймворк был добавлен так называемый **Minimal API** - минимизированная упрощенная модель, который еще упростила процесс разработки и написания кода приложения. Все остальные модели разработки работаю поверх базового функционала ASP.NET Core
* **ASP.NET Core MVC** представляет в общем виде построения приложения вокруг трех основных компонентов - Model (модели), View (представления) и Controller (контроллеры), где модели отвечают за работу с данными, контроллеры представляют логику обработки запросов, а представления определяют визуальную составляющую.



### **Особенности платформы**

* ASP.NET Core работает поверх платформы .NET и, таким образом, позволяет задействовать весь ее функционал.
* В качестве языков разработки применяются языки программирования, поддерживаемые платформой .NET. Официально встроенная поддержка для проектов ASP.NET Core есть у языков C# и F#
* ASP.NET Core представляет кроссплатформенный фреймворк, приложения на котором могут быть развернуты на всех основных популярных операционных системах: Windows, Mac OS, Linux. И таким образом, с помощью ASP.NET Core мы можем как создавать кроссплатформенные приложения на Windows, на Linux и Mac OS, так и запускать на этих ОС.
* Благодаря модульности фреймворка все необходимые компоненты веб-приложения могут загружаться как отдельные модули через пакетный менеджер Nuget.
* Поддержка работы с большинством распространенных систем баз данных: MS SQL Server, MySQL, Postgres, MongoDB
* ASP.NET Core характеризуется расширяемостью. Фреймворк построен из набора относительно независимых компонентов. И мы можем либо использовать встроенную реализацию этих компонентов, либо расширить их с помощью механизма наследования, либо вовсе создать и применять свои компоненты со своим функционалом.
* Богатый инструментарий для разработки приложений. В качестве инструментария разработки мы можем использовать такую среду разработки с богатым функционалом, как **Visual Studio** от компании Microsoft.

Кроме того, имеющаяся оснастка .NET CLI позволяет создавать и запускать проекты ASP.NET в консоли. И таким образом, для написания кода можно использовать обычных текстовый редактор, например, **Visual Studio Code**.

### **Структура проекта ASP.NET Core**

Инфраструктура ASP.NET Core MVC следует паттерну под названием «модель-представление-контроллер» (model-view-controller MVC), который управляет формой веб-приложения и взаимодействует между содержащимися в нем компонентами.

Взаимодействие пользователя с приложением, которое придерживается паттерном MVC, следует естественному циклу: пользователь предпринимает действие, а приложение в ответ изменяет свою модель данных и доставляет обновленное представление пользователю. Затем цикл повторяется. Это удобно укладывается в схему веб-приложения, представляемых в веде последовательностей запросов и ответов HTTP.

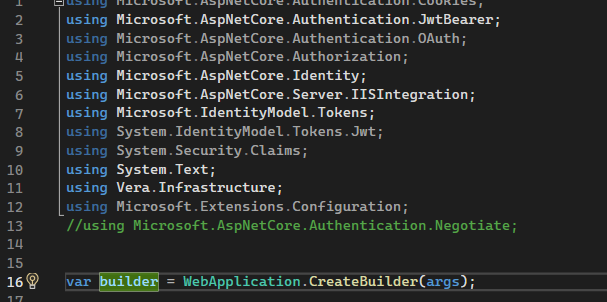
Веб-приложения, нуждаются в сочетаний нескольких технологиях(например, баз данных, HTML-разметки и исполняемого кода), обычно разделяются на набор слоев уровней. Полученные в результате таких сочетаний комбинаций естественным образом отображаются на концепций в паттерне MVC.

Проект начинается с помещения ASP.NET Core МVС в контекст разработки. Здесь объяснены преимущества и практическое влияние паттерна МVС, рассмотрен способ, которым инфраструктура ASP.NET Core МVС вписывается в современную разработку веб-приложений, а также описаны инструменты и средства языка С#, необходимые каждому программисту ASP.NET Core MVC.

В главе 2 мы углубимся в детали и создадим веб-приложение, чтобы получить представление о том, каковы основные компоненты и строительные блоки, и каким образом они сочетаются друг с другом. Однако большинство материала этой части посвящено разработке проекта под названием Vera, посредством которого демонстрируется реалистичный процесс разработки, начиная с постановки задачи и заканчивая развертыванием, с привлечением основных функциональных возможностей ASP.NET Core МVС.

В центре приложения ASP.NET находится класс WebApplication.

Однако для создания этого объекта необходим другой объект -WebApplicationBuilder, который в данном коде представлен переменной builder.



Кроме создания объекта WebApplication класс **WebApplicationBuilder** выполняет еще ряд задач, среди которых можно выделить следующие:

* Установка конфигурации приложения
* Добавление сервисов
* Настройка логирования в приложении
* Установка окружения приложения

Класс WebApplication применяется для управления обработкой запроса, установки маршрутов, получения сервисов и т.д.

Класс WebApplication применяет три интерфейса:

* **IHost**: применяется для запуска и остановки хоста, который прослушивает входящие запросы
* **IApplicationBuilder**: применяется для установки компонентов, которые участвуют в обработке запроса
* **IEndpointRouteBuilder**: применяется для установки маршрутов, которые сопоставляются с запросами

Для получения доступа к функциональности приложения можно использовать свойства класса WebApplication:

* **Configuration**: представляет конфигурацию приложения в виде объекта **IConfiguration**
* **Environment**: представляет окружение приложения в виде **IWebHostEnvironment**
* **Lifetime**: позволяет получать уведомления о событиях жизненного цикла приложения
* **Logger**: представляет логгер приложения по умолчанию
* **Services**: представляет сервисы приложения
* **Urls**: представляет набор адресов, которые использует сервер

**Зависимостей Dependency Injection.**

Dependency injection (DI) или внедрение зависимостей представляет механизм, который позволяет сделать взаимодействующие в приложении объекты слабосвязанными. Такие объекты связаны между собой через абстракции, например, через интерфейсы, что делает всю систему более гибкой, более адаптируемой и расширяемой.

Тем не менее остается проблема управления подобными зависимостями, особенно если это касается больших приложений. Нередко для установки зависимостей в подобных системах используются специальные контейнеры - IoC-контейнеры (Inversion of Control). Такие контейнеры служат своего рода фабриками, которые устанавливают зависимости между абстракциями и конкретными объектами и, как правило, управляют созданием этих объектов.

Преимуществом ASP.NET Core в этом отношении является то, что фреймворк уже по умолчанию имеет встроенный контейнер внедрения зависимостей, который представлен интерфейсом **IServiceProvider**. А сами зависимости еще называются сервисами, собственно поэтому контейнер можно назвать **провайдером сервисов**. Этот контейнер отвечает за сопоставление зависимостей с конкретными типами и за внедрение зависимостей в различные объекты.

### **Сервисы MVC**

Функциональность MVC и ее работа в приложении зависит от добавляемых сервисов. В практической части ниже мы будем использовать метод AddControllersWithViews() для добавления сервисов MVC, благодаря чему система маршрутизации сможет связать запрос с контроллером. Однако в данном случае у нас есть ряд опций по встраиванию сервисов, которые мы можем при необходимости использовать:

* **AddMvc()**: добавляет все сервисы фреймворка MVC (в том числе сервисы для работы с аутентификацией и авторизацией, валидацией и т.д.)
* **AddMvcCore()**: добавляет только основные сервисы фреймворка MVC, а всю дополнительную функциональность, типа аутентификацией и авторизацией, валидацией и т.д., необходимо добавлять самостоятельно
* **AddControllersWithViews()**: добавляет только те сервисы фреймворка MVC, которые позволяют использовать контроллеры и представления и связанную функциональность. При создании проекта по типу **ASP.NET Core Web App (Model-View-Controller)** используется именно этот метод
* **AddControllers()**: позволяет использовать контроллеры, но без представлений.

И в зависимости от того, насколько широко нам надо использовать возможности фреймворка MVC, выбирается соответствующий метод.

**Аутентификация и авторизация**

Важное место в приложении занимает аутентификация и авторизация. Аутентификация представляет процесс определения пользователя. Авторизация представляет процесс определения, имеет ли пользователь право доступа к некоторому ресурсу. То есть, если аутентификация отвечает на вопрос "Кем является пользователь?", то авторизация отвечает на вопрос "Какие права пользователь имеет в системе?" ASP.NET Core имеет встроенную поддержку аутентификации и авторизации.

### **Аутентификация**

Для выполнения аутентификации в конвейере обработки запроса отвечает специальный компонент middleware - AuthenticationMiddleware. Для встраивания этого middleware в конвейер применяется метод расширения UseAuthentication()

Следует отметить, что метод **UseAuthentication()** должен встраиваться в конвейер до любых компонентов middleware, которые используют аутентификацию пользователей.

Для выполнения аутентификации этот компонент использует сервисы аутентификации, в частности, сервис **IAuthenticationService**, которые регистрируются в приложении с помощью метода **AddAuthentication()**:

В качестве параметра вторая версия метода **AddAuthentication()** принимает схему аутентификации в виде строки. Третья версия метода AddAuthentication принимает делегат, который устанавливает опции аутентификации - объект **AuthenticationOptions**.

Какую бы мы версию метода не использовали, для аутентификации необходима установить схему аутентификации. Две наиболее распространенные схемы аутентификации:

* **"Cookies"**: аутентификация на основе куки. Хранится в константе CookieAuthenticationDefaults.AuthenticationScheme
* **"Bearer"**: аутентификация на основе jwt-токенов. Хранится в константе JwtBearerDefaults.AuthenticationScheme

**Схема аутентификации** позволяет выбирать определенный обработчик аутентификации. **Обработчик аутентификации**, собственно, и выполняет непосредственную аутентификацию пользователей на основе данных запросов и исходя из схемы аутентификации.

Например, для аутентификации с помощью куки передается схема "Cookies". Соответственно для аутентификации пользователя будет выбираться встроенный обработчик аутентификации класс **Microsoft.AspNetCore.Authentication.Cookies.**

**CookieAuthenticationHandler**, который на основе полученных в запросе cookie выполняет аутентификацию.

А если используется схема "Bearer", то это значит, что для аутентификации будет использоваться jwt-токен, а в качестве обработчика аутентификации будет применяться класс Microsoft.AspNetCore.Authentication.JwtBearer.JwtBearerHandler. Стоит отметить, что для аутентификации с помощью jwt-токенов необходимо добавить в проект через Nuget пакет **Microsoft.AspNetCore.Authentication.JwtBearer**

При чем в ASP.NET Core мы не ограничены встроенными схемами аутентификации и можем создавать свои кастомные схемы и под них своих обработчиков аутентификации.

Кроме применения схемы аутентификации необходимо подключить аутентификацию определенного типа. Для этого можно использовать следующие методы:

* **AddCookie()**: подключает и конфигурирует аутентификацию с помощью куки.
* **AddJwtBearer()**: подключает и конфигурирует аутентификацию с помощью jwt-токенов (для этого метода необходим Nuget-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**)

### **Авторизация**

Авторизация представляет процесс определения прав пользователя в системе, к каким ресурсам приложения он имеет право доступа и при каких условиях.

Хотя авторизация представляет отдельный независимый процесс, тем не менее для нее также необходимо, чтобы приложение также применяло аутентификацию.

Для подключения авторизации необходимо встроить компонент **Microsoft.AspNetCore.Authorization.AuthorizationMiddleware**. Для этого применяется встроенный метод расширения **UseAuthorization()**

Кроме того, для применения авторизации необходимо зарегистрировать сервисы авторизации с помощью метода **AddAuthorization()**:

Вторая версия метода принимает делегат, который с помощью параметра **AuthorizationOptions** позволяет сконфигурировать авторизацию.

Ключевым элементом механизма авторизации в ASP.NET Core является атрибут **AuthorizeAttribute** из пространства имен Microsoft.AspNetCore.Authorization, который позволяет ограничить доступ к приложению.

Атрибут **Authorize** указывается перед обработчиком конечной точки.

Применение данного атрибута означает, что к конечной точке, имеют доступ только аутентифицированные пользователи.

**Аутентификация с помощью JWT-токенов**

Одним из подходов к авторизации и аутентификации в ASP.NET Core представляет механизм аутентификации и авторизации с помощью JWT-токенов. Что такое JWT-токен? JWT (или JSON Web Token) представляет собой веб-стандарт, который определяет способ передачи данных о пользователе в формате JSON в зашифрованном виде.

JWT-токен состоит из трех частей:

* **Header** - объект JSON, который содержит информацию о типе токена и алгоритме его шифрования
* **Payload** - объект JSON, который содержит данные, нужные для авторизации пользователя
* **Signature** - строка, которая создается с помощью секретного кода, Headera и Payload. Эта строка служит для верификации токена

Для использования JWT-токенов в проект ASP.NET Core необходимо добавить Nuget-пакет **Microsoft.AspNetCore.Authentication.JwtBearer**.

**Введение в Entity Framework Core**

Entity Framework Core (EF Core) представляет собой объектно-ориентированную, легковесную и расширяемую технологию от компании Microsoft для доступа к данным. EF Core является ORM-инструментом (object-relational mapping - отображения данных на реальные объекты). То есть EF Core позволяет работать базами данных, но представляет собой более высокий уровень абстракции: EF Core позволяет абстрагироваться от самой базы данных и ее таблиц и работать с данными независимо от типа хранилища. Если на физическом уровне мы оперируем таблицами, индексами, первичными и внешними ключами, но на концептуальном уровне, который нам предлагает Entity Framework, мы уже работаем с объектами. Entity Framework Core поддерживает множество различных систем баз данных. Таким образом, мы можем через EF Core работать с любой СУБД, если для нее имеется нужный провайдер.

По умолчанию на данный момент Microsoft предоставляет ряд встроенных провайдеров: для работы с MS SQL Server, для SQLite, для PostgreSQL. Также имеются провайдеры от сторонних поставщиков, например, для MySQL.

Центральной концепцией Entity Framework является понятие сущности или **entity**. Сущность определяет набор данных, которые связаны с определенным объектом. Поэтому данная технология предполагает работу не с таблицами, а с объектами и их коллекциями. При этом сущности могут быть связаны ассоциативной связью один-ко-многим, один-ко-одному и многие-ко-многим, подобно тому, как в реальной базе данных происходит связь через внешние ключи.

**ASP.NET Core Identity**

ASP.NET Identity представляет встроенную в ASP.NET систему аутентификации и авторизации. Данная система позволяет пользователям создавать учетные записи, аутентифицироваться, управлять учетными записями или использовать для входа на сайт.

### **Контекст данных IdentityDbContext**

Для работы с базой данных ASP NET Identity использует контекст данных, который наследуется от класса IdentityDbContext из пространства имен Microsoft.AspNetCore.Identity.EntityFrameworkCore

По умолчанию данный класс наследует весь функционал от IdentityDbContext. Так, мы можем получить содержимое таблиц из бд с помощью следующих свойств:

* **Users**: набор объектов IdentityUser, соответствует таблице пользователей
* **Roles**: набор объектов IdentityRole, соответствует таблице ролей
* **RoleClaims**: набор объектов IdentityRoleClaim, соответствует таблице связи ролей и объектов claims
* **UserLogins**: набор объектов IdentityUserLogin, соответствует таблице связи пользователей с их логинами их внешних сервисов
* **UserClaims**: набор объектов IdentityUserClaim, соответствует таблице связи пользователей и объектов claims
* **UserRoles**: набор объектов IdentityUserRole, соответствует таблице, которая сопоставляет пользователей и их роли
* **UserTokens**: набор объектов IdentityUserToken, соответствует таблице токенов пользователей

**IdentityUser**

В ASP.NET Core Identity пользователь представлен классом **IdentityUser** из пространства имен Microsoft.AspNetCore.Identity.EntityFrameworkCore. Этот класс предоставляет базовую информацию о пользователе с помощью следующих свойств:

* **Id**: уникальный идентификатор пользователя
* **UserName**: ник пользователя
* **Email**: электронный адрес пользователя
* **Logins**: коллекция логинов, которые использовались пользователем для входа через сторонние сервисы (Google, Facebook и т.д.)
* **Claims**: коллекция клаймов или дополнительных объектов, которые используются для авторизации пользователя
* **PasswordHash**: хеш пароля. В базе данных напрямую не хранится пароль, а только его хеш.
* **Roles**: набор ролей, к которым принадлежит пользователь
* **PhoneNumber**: номер телефона
* **SecurityStamp**: некоторое специальное значение, которое меняется при смене аутентификационных данных, например, пароля
* **AccessFailedCount**: количество неудачных входов пользователя в систему
* **EmailConfirmed**: подтвержден ли адрес электронной почты
* **PhoneNumberConfirmed**: подтвержден ли номер телефона

**Менеджер пользователей UserManager**

Как правило, для управления пользователями используется не контекст данных, а специальный класс - UserManager<T> из пространства имен Microsoft.AspNetCore.Identity. Основные из его методов и свойств:

* **ChangePasswordAsync(user, old, new):** изменяет пароль пользователя
* **CreateAsync(user):** создает нового пользователя
* **DeleteAsync(user):** удаляет пользователя
* **FindByIdAsync(id):** ищет пользователя по id
* **FindByEmailAsync(email):** ищет пользователя по email
* **FindByNameAsync(name):** ищет пользователя по нику
* **UpdateAsync(user):** обновляет пользователя
* **Users:** возвращает всех пользователей
* **AddToRoleAsync(user, role):** добавляет для пользователя user роль role
* **GetRolesAsync (user):** возвращает список ролей, к которым принадлежит пользователь user
* **IsInRoleAsync(user, name):** возвращает true, если пользователь user принадлежит роли name
* **RemoveFromRoleAsync(user, name):** удаляет роль name у пользователя user

В проекте будут использованы и те и другие подходы.

## **Авторизация пользователей в Identity**

Аутентификация и авторизация предоставляются системой ASP.NET Сoге Ideпtity. которая аккуратно интегрируется как в платформу ASP.NET Сoге, так и в приложения MVC. В последующих разделах мы создадим базовую настройку защиты, которая позволит одному пользователю проходить аутентификацию и получать доступ к административным функциям в приложении. Система ASP.NET Сoге Ideпtity предлагает множество других средств для аутентификации пользователей, а также авторизации доступа к функциям и данным приложения.

**CRUID – операций**

Большинство операций с данными так или иначе представляют собой CRUD операции (Create, Read, Update, Delete), то есть создание, получение, обновление и удаление. Entity Framework Core позволяет легко выполнять все эти действия.

**Модели и базы данных**

Модели (М в MVC) содержат данные, с которыми работают пользователи.

Существуют два обширных типа моделей: модели представлений, которые выражают сами данные, передаваемые из контроллера в представление, и модели предметной области, которые содержат данные в предметной области наряду с операциями, трансформациями и правилами для создания, хранения и манипулирования данными, вместе называемыми логикой моделей.

После определения моделей надо выбрать хранилище данных для этих моделей. Мы будем использовать MySql.EntityFrameworkCore. Преимущество фреймворка Entity Framework состоит в том, что он позволяет абстрагироваться от структуры конкретной базы данных и вести все операции с данными через модель.

Чтобы взаимодействовать с базой данных нам нужен контекст данных. Причем Entity Framework Core использует подход Code First, при котором нам надо сначала определить модели и контекст данных, а потом уже исходя и этих моделей и класса контекста будет создаваться бд и все ее таблицы.

**Контроллер и их действия**

Контроллеры являются "соединительной тканью" паттерна MVC, исполняя роль

каналов между моделью данных и представлениями. Контроллеры определяют действия, предоставляющие бизнес-логику, которая оперирует на модели данных и обеспечивает представления данными, подлежащими отображению для пользователя.

Центральным звеном в архитектуре ASP.NET Core MVC является контроллер. При получении запроса система маршрутизации выбирает для обработки запроса нужный контроллер и передает ему данные запроса. Контроллер обрабатывает эти данные и посылает обратно результат обработки.

В ASP.NET Core MVC контроллер представляет обычный класс на языке C#, который наследуется от абстрактного базового класса Microsoft.AspNetCore.Mvc.Controller. По умолчанию проект ASP.NET Core MVC содержит как минимум один контроллер – HomeController.

**Асинхронные методы и класс Task**

Асинхронные методы осуществляют возврат и выполняют работу в фоновом режиме с уведомлением о ее завершении, позволяя коду в это время заниматься другими действиями. Асинхронные методы - важный инструмент при устранении узких мест в коде, они позволяют приложениям извлекать преимущества от наличия нескольких процессоров и процессорных ядер. чтобы выполнять работу в параллельном режиме.

В инфраструктуре MVC асинхронные методы могут применяться для увеличения

общей производительности приложения, предоставляя серверу большую степень гибкости относительно того, как запросы планируются и выполняются. Для выполнения работы асинхронным образом используются два ключевых слова С# - async и await.

В эпоху многоядерных машин, которые позволяют параллельно выполнять сразу несколько процессов, стандартных средств работы с потоками в .NET уже оказалось недостаточно. Поэтому во фреймворк .NET была добавлена библиотека параллельных задач TPL (Task Parallel Library), основной функционал которой располагается в пространстве имен System.Threading.Tasks. Данная библиотека упрощает работу с многопроцессорными, многоядерными системами. Кроме того, она упрощает работу по созданию новых потоков. Поэтому обычно рекомендуется использовать именно TPL и ее классы для создания многопоточных приложений, хотя стандартные средства и класс Thread по-прежнему находят широкое применение.

В основе библиотеки TPL лежит концепция задач, каждая из которых описывает отдельную продолжительную операцию. В библиотеке классов .NET задача представлена специальным классом - классом **Task**, который находится в пространстве имен System.Threading.**Tasks**. Данный класс описывает отдельную задачу, которая запускается асинхронно в одном из потоков из пула потоков. Хотя ее также можно запускать синхронно в текущем потоке. Однако в любом случае следует отметить, что **задача - это не поток**.

**Результаты действий**

Представления содержат логику. которая требуется для отображения данных пользователю или для сбора данных от пользователя, так что они могут быть обработаны каким-то действием контроллера.

При обращении к веб-приложению, как правило, пользователь ожидает получить некоторый ответ, например, в виде веб-страницы, которая наполнена данными. На стороне сервера метод контроллера, получая параметры и данные запроса, обрабатывает их и формирует ответ в виде результата действия. Результат действия - это тот объект, который возвращается методом после обработки запроса.

**Запрос Post данных в контроллер через формы**

Кроме GET-запросов также широко применяются POST-запросы. Как правило, такие запросы отправляются с помощью форм на веб-странице. Но основные принципы передачи данных будут теми же, что и в GET-запросах.

При получении **сложных** объектов, данных форм действуют те же правила привязки, что и при получении параметров строки запроса. Поэтому с той же формы можно получить значения в виде сложных объектов, в которых названия свойств соответствуют названиям полей формы

В данном случае поля формы соответствуют по названию свойствам класса **BloodPressure**, поэтому вместо одиночных разрозненных значений мы можем получить отправленную форму в виде объекта **BloodPressure**.

**Контроллер Web API**

Web API представляет способ построения приложения ASP.NET, который специально заточен для работы в стиле REST (Representation State Transfer или "передача состояния представления"). REST-архитектура предполагает применение следующих методов или типов запросов HTTP для взаимодействия с сервером:

* GET
* POST
* PUT
* DELETE

Зачастую REST-стиль особенно удобен при создании всякого рода Single Page Application, которые нередко используют специальные javascript-фреймворки типа Angular, React или Vue.js. По сути Web API представляет собой веб-службу, к которой могут обращаться другие приложения. Причем эти приложения могут представлять любую технологию и платформу - это могут быть веб-приложения, мобильные или десктопные клиенты.

**Frontend(HTML, Js, Vue, Css)**

В работе над сайтом фронтенд-разработчик использует язык разметки HTML (от англ. HyperText Markup Language), каскадные таблицы стилей CSS и JavaScript — язык программирования, который добавляет веб-сайту интерактивности.

Большая часть работы специалиста заключается в том, чтобы сайт или приложение были простым в навигации и понятным для пользователя.

Фронтенд (англ. frontend) — это разработка пользовательских функций и интерфейса. К ним относится всё, что пользователи видят на сайте или в приложении, и с чем можно взаимодействовать: картинки, выпадающие списки, меню, анимация, карточки товаров, кнопки, чекбоксы, интерактивные элементы.

Если говорить простым языком, то взаимодействие между backend и frontend происходит по кругу:

1. фронтенд отправляет пользовательскую информацию в бэкенд.

2. Там информация обрабатывается.

3. Информация возвращается обратно, принимая понятную для пользователя форму.

Варианты взаимодействия backend и frontend:

• HTTP-запрос. Напрямую отправляется на сервер, сервер ищет данные, встраивает их в шаблон, а потом возвращает в виде HTML-страницы.

**Элементы и атрибуты HTML5**

Прежде чем переходить непосредственно к созданию своих веб-страниц на HTML5, рассмотрим основные строительные блоки, кирпичики, из которых состоит веб-страница.

Документ HTML5, как и любой документ HTML, состоит из элементов, а элементы состоят из тегов. Как правило, элементы имеют открывающий и закрывающий тег, которые заключаются в угловые скобки.

Для расширения возможностей HTML тегов, используются атрибуты, с помощью которых мы можем дополнительно сообщить браузеру дополнительную информацию о том, как отображать тот или иной HTML элемент.

HTML атрибут состоит из имени и значения, между которыми ставится знак равно (=). В HTML коде это выглядит так:

<tag имя атрибута=”значение”>Текст </tag>.

Атрибуты HTML - это модификатор типа элемента HTML. Атрибут либо изменяет функциональность типа элемента по умолчанию, либо предоставляет функциональность определенным типам элементов, без которых они не могли бы корректно функционировать. В синтаксисе HTML атрибут добавляется к начальному тегу HTML.

**Основы css**

В теге link в атрибуте href расположен файл с стилями страницы.

Любой html-документ, сколько бы он элементов не содержал, будет, по сути, "мертвым" без использования стилей. Стили или лучше сказать каскадные таблицы стилей (**Cascading Style Sheets**) или попросту CSS определяют представление документа, его внешний вид. Рассмотрим вкратце применение стилей в контексте HTML5.

Стиль в CSS представляет правило, которое указывает веб-браузеру, как надо форматировать элемент. Форматирование может включать установку цвета фона элемента, установку цвета и типа шрифта и так далее.

Определение стиля состоит из двух частей: **селектор**, который указывает на элемент, и **блок объявления стиля** - набор команд, которые устанавливают правила форматирования.

**CSS фреймворк Bootstrap**

Преимущество использования CSS фреймворков заключается в том, что верстальщику не нужно думать о многих нюансах верстки, которые за него уже продумали создатели фреймворков. К таким нюансам относятся кроссбраузерность, поддержка различных разрешений экранов и многое другое. Верстальщик лишь указывает, что, как и когда нужно показать, остальное фреймворк делает сам. Данный подход может сильно ускорить вёрстку сайта. К преимуществам Bootstrap относится и его популярность. Это означает, что другому верстальщику будет проще поддерживать ваш код.

Здесь нам встречается первый компонент Bootstrap – колонки. Родительскому элементу колонок задается класс «row», а классы колонок начинаются с префикса «col-», затем идет размер экрана (xs, sm, md, lg), а заканчиваются относительной шириной колонки.

Колонке можно задавать одновременно различные классы со значениями для экранов, например class=«col-xs-12 col-md-8». Эти классы просто задают ширину колонке в процентах для определенного размера экрана. Если колонке не задан класс определенного экрана, то применится класс для минимально определенного экрана, а если и он не указан – то никакая ширина не применится и блок займет максимально возможную ширину. Подключаемые файлы:

<link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />.

<script src="/lib/bootstrap/dist/js/bootstrap.js"></script>

**Что такое jQuery**

Современное веб-программирование и создание веб-сайтов уже невозможно представить без использования языка JavaScript. Однако в настоящее время, все чаще используется не "голый" код javascript, а javascript-фреймворки и библиотеки. Одной из таких библиотек, причем наверное самой популярной на сегодняшний день, является jQuery. По некоторым оценкам не менее половины крупнейших сайтов в интернете используют эту библиотеку.

Хотя мы и можем назвать jQuery библиотекой, он на самом деле понятие "jQuery" объединяет целую экосистему библиотек, построенный вокруг базовой библиотеки: это и библиотека jquery.ui, предназначенная для создания визуальных интерфейсов, это и jqyery.mobile, используемая при разработке мобильных сайтов и др.

Какие преимущества несет нам использование jQuery?

* **Упрощение работы с кодом**. jQuery предлагает простой элегантный синтаксис для манипулирования элементами на веб-странице.
* **Расширяемость**. Весь код jQuery открыт для просмотра и изменения, и в случае, если что-то в библиотеке не устраивает, ее можно модифицировать. А также можно создавать плагины jQuery.
* **Кроссбраузерность**. jQuery имеет поддержку большинства известных браузеров, в том числе таких. как IE 7,8. (Хотя в силу того, что браузеры IE 6-8 постепенно становятся достоянием истории, а также чтобы уменьшить размер библиотеки в последней версии была прекращена поддержка IE 6-8).

**Подключение библиотеки jQuery**

Чтобы начать работать с данной библиотекой, нам первым делом надо ее загрузить. Ее найти можно на официальном сайте разработчика <https://jquery.com/download/>. На странице загрузок в самом можно найти несколько версий jQuery.

Библиотека представлена в двух вариантах - Compressed или Monified (минимизированная) и Uncompressed (обычный). Минимизированные версии предоставляют ту же функциональность, что и обычные, но отличаются тем, что не содержат всяких необязательных символов, наподобие пробелов, комментариев и т.д., и поэтому в своем названии имею суффикс min, например, *jquery-1.10.1.min.js*. Поскольку они производительнее за счет меньшего объема, их рекомендуется использовать в реальном производстве. В то же время, если вам захочется понять логику кода jQuery, то в этом случае можно обращаться к обычной версии библиотеки.

Библиотека jquery подключается также, как и другие файлы javascript.

<script src="/lib/jquery/dist/jquery.js"></script>

**Vue.js**

Vue предоставляет [официальный CLI (opens new window)](https://github.com/vuejs/vue-cli)для быстрого создания каркаса одностраничных приложений (SPA). Предлагаемые шаблоны содержат всё необходимое для организации современной фронтенд-разработки. За несколько минут можно получить работающую конфигурацию с горячей перезагрузкой модулей, листингом кода при сохранении и настроенной конфигурацией production-сборки.

<script src="/js/vue.min.js"></script>

Что такое Vue?​

Vue (произносится /vjuː/, примерно как view) — JavaScript фреймворк для создания пользовательских интерфейсов. Он создан на стандартах HTML, CSS и JavaScript и предоставляет декларативную и компонентную модель программирования, которая помогает эффективно разрабатывать пользовательские интерфейсы любой сложности.

**Метод входа в систему LoginIn()**

Метод **LoginIn(),** представляет из себя адаптивную размер формы авторизаций и регистраций в сервисе, который меняет свойства ширины и высоты, в зависимости от размера экрана.

JavaScript может отправлять сетевые запросы на сервер и подгружать новую информацию по мере необходимости.

Для сетевых запросов из JavaScript есть широко известный термин «AJAX» (аббревиатура от Asynchronous JavaScript And XML). XML мы использовать не обязаны, просто термин старый, поэтому в нём есть это слово. Возможно, вы его уже где-то слышали. Есть несколько способов делать сетевые запросы и получать информацию с сервера.

Метод **fetch()** — современный и очень мощный, поэтому начнём с него. Он не поддерживается старыми , но поддерживается всеми современными браузерами.

Базовый синтаксис: **let promise = fetch(url, [options])**

* url – URL для отправки запроса.
* options – дополнительные параметры: метод, заголовки и так далее.

Без options это простой GET-запрос, скачивающий содержимое по адресу url.

В основе метода LoginIn(), имеется метод fetch() , который по адресу /account/login, запрашивает форму логина. Далее происходит взаимодействие с тегами, подгруздка ssc – стилей, обновление данных.

Валидация формы login предполагается, что пользователь вводит неверные данные или не вводит их вообще.

На форме предполагается следующий интерфейс:

* Кнопка оправки формы логина на сервер, для авторизаций.
* Кнопка отмены оправки формы.
* Ввод данных в виде логина, пароля, адрес почты.
* Подгруздка капчи(код в виде искаженных цифр), которая в свою очередь, предполагает что данные вводит пользователь, а не робот.

**Интерфейс пользователя**

Интерфейс пользователя предполагается в виде:

1. Заголовка описания модели.
2. Добавление замеров АД
3. Основная таблица, в которой находятся модель по замеру АД
4. Удаление, обновление в таблице значений с помощью библиотеки Vue.js
5. Визуализация данных с помощью библиотеки ChartJS.

**ChartJS — JavaScript-библиотека визуализации данных**

JavaScript-библиотека Chart.js позволяет генерировать на стороне клиента привлекательные графики и диаграммы с использованием средств HTML5 (canvas). Библиотека поддерживает создание линейных графиков, столбцовых, круговых и радиальных диаграмм. Chart.js поддерживает использование анимированных эффектов.

Большим достоинством Chart.js является ее небольшой размер и отсутствие каких-либо зависимостей. К тому же имеется возможность ещё уменьшить размер библиотеки путем включения в нее только тех модулей, которые необходимы в конкретном случае. К примеру, если нужно создать только диаграмму кругового типа (pie chart), то можно подключить ядро библиотеки Chart.js и модуль, с помощью которого создаются диаграммы подобного типа. Таким образом, будет уменьшен общий размер библиотеки Chart.js и увеличена скорость загрузки страницы в целом. Другим достоинством библиотеки является адаптивность диаграммы, позволяющая изменять свой размер при изменении размеров окна браузера.

**Объект SmtpClient**

Как правило, большинство крупных сервисов при регистрации требуют ее подтверждения по email.

В ASP.NET MVC есть несколько моделей авторизации и аутентификации. В данном случае используем механизм авторизации AspNet Identity, так как он более сложен в плане понимания. При использовании других методов аутентификации/авторизации общая схема подтверждения регистрации по email будет аналогична.

Для отправки почты в среде интернет используется протокол SMTP (Simple Mail Transfer Protocol). Данный протокол указывает, как почтовые сервера взаимодействуют при передаче электронной почты.

**Глава 2 Практическая часть**

**Установка .NET Core SDK**

Установка Visual Studio содержит все средства. необходимые для разработки приложений ASP.NET Core MVC. но не включает комплект .NET Core SDK. который должен быть загружен и установлен отдельно.

Перейдем по ссылке https: / / www. microsoft.com/ net / core. загрузим программу установки .NET Соге SDK для Windows и запустим ее. После завершения работы

программы установки откроем окно командной строки или окно PowerShell и введем следующую команду для отображения установленной версии платформы NET:

dotnet --version

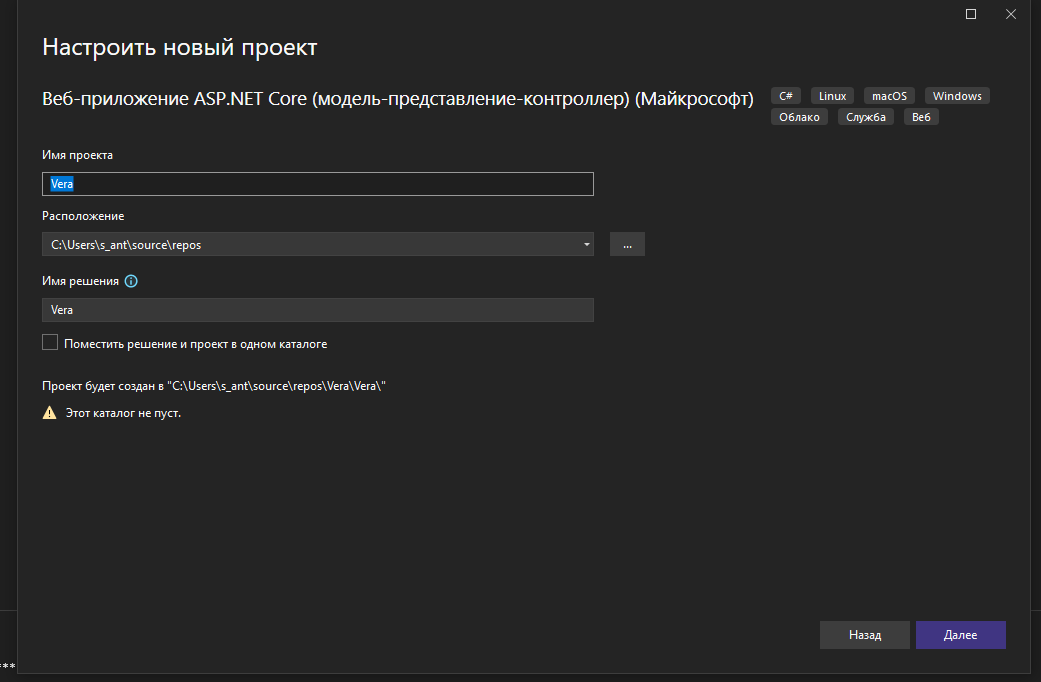
Если установка прошла успешно. тогда результирующим выводом команды будет

версия dotnet.

**Создание проекта**

ASP.NET Core MVC имеет встроенную поддержку создания веб-API. Объединение двух платформ упрощает создание приложений, включающих как пользовательский интерфейс (HTML), так и API, так как в данном случае у них будет общий код и конвейер.

Для создания проекта на ASP.NET Core MVC мы можем выбрать любой тип проекта на ASP.NET Core и в нем уже добавлять необходимые компоненты. Однако для упрощения Visual Studio уже по умолчанию предоставляет для этого шаблон веб приложение ASP.NET Core (Model-View-Controller):



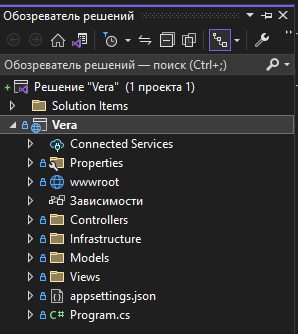
Выберем данный шаблон для создания проекта. Дальше нам откроется окно для установки имени проекта. Проект будет называться Vera.

Далее нам надо будет настроить стандартные настройки для ASP.NET Core.

Оставим все настройки по умолчанию и нажмем на ОК. И Visual Studio создаст новый проект MVC.

**Создание структуры проекта**

Рассмотрим базовую структуру простейшего стандартного проекта ASP.NET Core:



* **Dependencies**: все добавленные в проект пакеты и библиотеки
* **wwwroot**: этот узел (на жестком диске ему соответствует одноименная папка) предназначен для хранения статических файлов - изображений, скриптов javascript, файлов css и т.д., которые используются приложением.
* **Controllers**: папка для хранения контроллеров, используемых приложением. По умолчанию здесь уже есть один контроллер – Homecontroller
* **Infrasructure**: в папке находится класс контекста данных EF, сервис по отправке данных на почту через SMTPClient и т.д.
* **Models**: каталог для хранения моделей. По умолчанию здесь создается модель ErrorviewModel
* **Views**: каталог для хранения представлений. Здесь также по умолчанию добавляются ряд файлов - представлений
* **appsettings.json**: хранит конфигурацию приложения
* **Program.cs**: файл, который определяет входную точку в приложение ASP.NET Core

Фактически эта та же структура, что и у проекта по типу Empty за тем исключением, что здесь также добавлены по умолчанию папки для ключевых компонентов фреймворка MVC: контроллеров и представлений. А также есть дополнительные узлы и файлы для управления зависимостями клиентской части приложения.

**Предполагается что структура разработки проекта следующая**:

Vera.**Models** {

User {

int id;

string Firstname;

string lastname

DateTime birthdate

string isapproved

string role

string email

string emailconfirmed

}

RegisterModel {

string Firstname;

string Lastname;

string City;

string Email;

int Year;

string Password;

string PasswordConfirm;

string Captcha;

}

LoginModel {

string Email;

string Password;

string Captcha;

}

BloodPressure {

int id;

DateTime datetimebloodpressure;

int sys;

int dia;

int pulse;

string comment;

}

Vera.**Controllers**{

**AccountController** {

[HttpGet]

IActionResult **StartPage()** – стартовая страница, где имеется

интерфейс для регистраций пользователя и входа в сервис с учетными данными.

[HttpGet]

IActionResult **Register()** – форма регистраций пользователя.

[HttpPost]

Task **<**IActionResult**> Register(**RegisterModelmodel**) –** метод Post регистраций.

Task **GenerateToken(**ApplicationUseruser**)** – генерация токена.

ActionResult **Captcha() –** генерация рандомного числа.

IActionResult **Login() –** метод для входа в сервис.

Task <IActionResult> **Logout() –** метод для выхода из сервиса.

}

**HomeControlle**{

IActionResult **Index() –** основной интерфейспользователя.

Task <IActionResult> **AddSave**(BloodPressure model) – добавление сохранение замеров давления.

Task<IActionResult> **Edit**(BloodPressure model) – редактирование сохраненных данных замера давления.

Task<IActionResult> **DeleteData(int id) –** удаление данных замера давления.

}

[Route("api/[controller]")]

[ApiController]

PressureController{

[HttpGet]

IActionResult **Get() –** возвращает список замера давления, для пользователя.

[HttpGet("{date1}/{date2}")]

IActionResult **GetDate**(string date1, string date2) - возвращает список замера давления, для пользователя отсортированным по датам.

[HttpPut("{id}")]

IActionResult **PutUpdata**(int id)

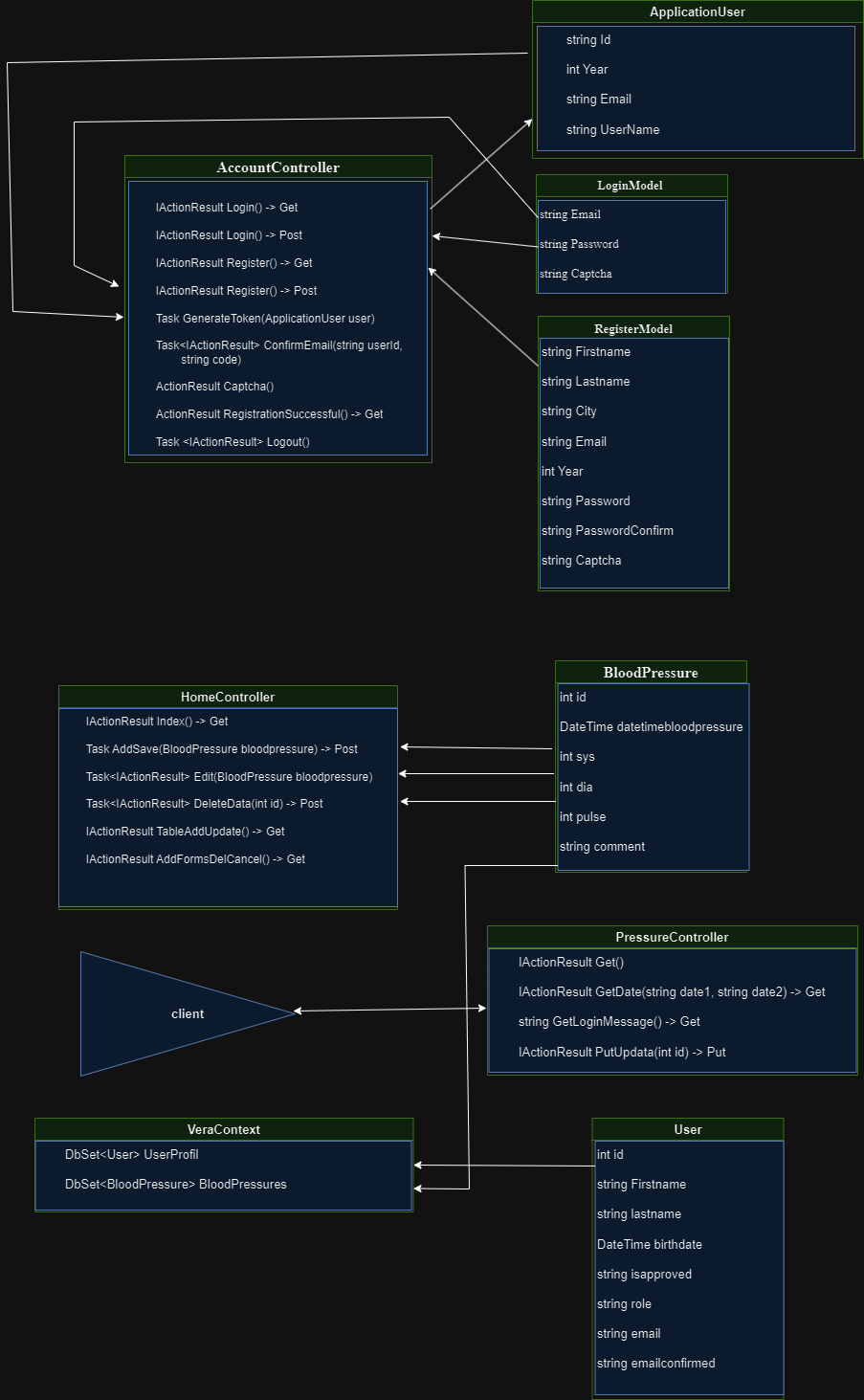
}

}

Атрибут:

* HttpGet – указывает на Get запросы.
* HttpPost - указывает на Post запросы.
* HttpPut – создаёт новый ресурс или заменяет представление целевого ресурса данными, представленными в теле запроса
* ApiController, Route("api/[controller]") - прежде всего к контроллеру применяется атрибут [ApiController], который позволяет использовать ряд дополнительных возможностей, в частности, в плане привязки модели и ряд других. Также к контроллеру применяется атрибут маршрутизации, который указывает, как контроллер будет сопоставляться с запросами.

**Структура проекта:**



**Внедрение зависимостей Dependency Injection.**

**Внедрение зависимостей —** это шаблон, согласно которому компонент (класс) передаёт заботу о создании необходимых для его работы других компонентов вовне.

В ASP.NET Core есть следующие способы передачи зависимостей компоненту:

1. Через конструктор класса (для любых классов).
2. Через параметр метода класса (для контроллеров MVC).
3. Через свойство класса (для представлений MVC).

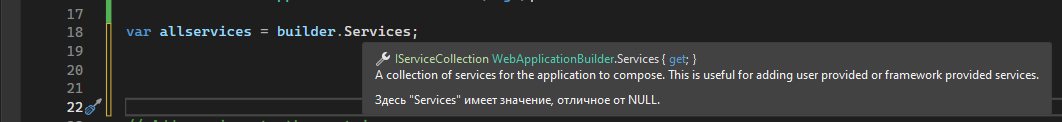
Если внедряемые компоненты также имеют зависимости, механизм, отвечающий за внедрение зависимостей, должен создать всю необходимую цепочку компонентов.

**Установка, регистрация, добавление встроенных сервисов**

За управление сервисами в приложении в классе **WebApplicationBuilder** определено свойство **Services**, которое представляет объект **IServiceCollection** - коллекцию сервисов:

**WebApplicationBuilder** builder **= WebApplication.CreateBuilder();**

**IServiceCollection** allServices **=** builder**.**Services**;** // коллекция сервисов



И даже если мы не добавляем в эту коллекцию никаких сервисов, IServiceCollection уже содержит ряд сервисов по умолчанию

Класс WebApplication применяется для управления обработкой запроса, установки маршрутов, получения сервисов и т.д.

Таким образом, после вызова метод Run/Start/RunAsync/StartAsync приложение будет запущено, и мы сможем к нему обращаться:

WebApplicationBuilder builder = WebApplication.CreateBuilder();

WebApplication app = builder.Build ();

app.Run();

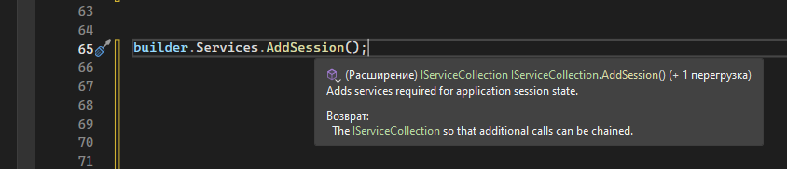
### **Регистрация встроенных сервисов ASP.NET Core**

Кроме ряда подключаемых по умолчанию сервисов ASP.NET Core имеет еще ряд встроенных сервисов, которые мы можем подключать в приложение при необходимости. Все сервисы и компоненты middleware, которые предоставляются ASP.NET по умолчанию, регистрируются в приложение с помощью методов расширений IServiceCollection, имеющих общую форму **Add**[название\_сервиса].

var builder = WebApplication.**CreateBuilder**();

builder.Services.**AddSession** ();

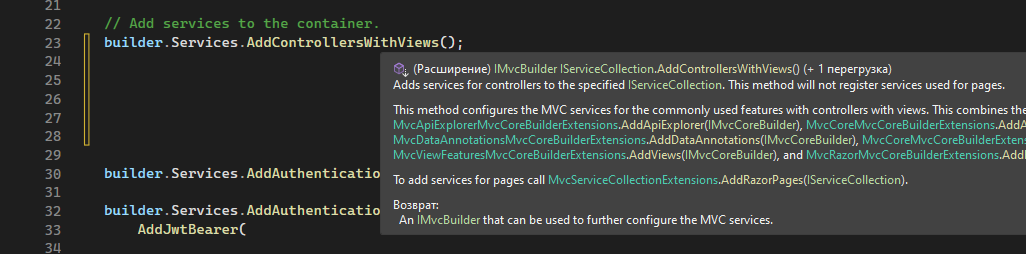
Например:



Для объекта IServiceCollection определено ряд методов расширений, которые начинаются на Add, как, например, AddSession(). Эти методы добавляют в объект IServiceCollection соответствующие сервисы. Например, AddSession() добавляет в приложение сервисы , добавляем сервис работы с сессиями благодаря чему мы сможем их использовать в приложении.

**Добавляем сервисы MVC**

Теперь задействуем функциональность фреймворка MVC в нашем приложении и изменим файл **Program.cs** следующим образом:

****

Прежде всего надо отметить, что функциональность MVC, в частности, поддержка контроллеров и представлений, подключается в приложение в виде сервиса - в данном случае с помощью вызова services.AddControllersWithViews(). После этого можем использовать функциональность фреймворка MVC.

Кроме того, чтобы связать приходящие от пользователей запросы с контроллерами применяется метод MapControllerRoute().

app.MapControllerRoute(

name: "default",

pattern: "{controller=account}/{action=startpage}/{id?}");

Через первый параметр - name в метод передается название маршрута - в данном случае "default". Через второй параметр - параметр pattern передается шаблон маршрута, которому должен соответствовать запрос. В качестве шаблона маршрута применяется шаблон "{controller=Home}/{action=Index}/{id?}", который представляет трехсегментный запрос. В нем первый сегмент представляет контроллер, второй сегмент - метод контроллера, а третий - необязательный параметр. При этом если в запросе не указаны сегменты (например, обращение идет к корню веб-приложения), то в качестве контроллера по умолчанию применяется HomeController, а в качестве его метода - метод Index.

**Добавление ASP.NET Core Identity**

Но чтобы все эти станицы могли выполнять свою роль, в классе Program.cs добавляется конфигурация Identity. В приложение добавляются сервисы Identity:

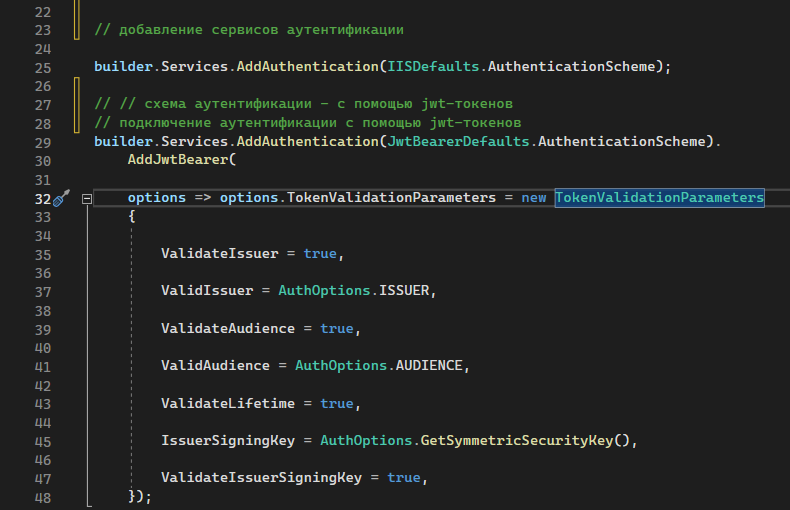
builder.Services.AddIdentity<ApplicationUser, IdentityRole>().AddEntityFrameworkStores<VeraContext>()

.AddDefaultTokenProviders();

Функционал аутентификации и авторизации на основе системы Identity становится доступным через вызовы методов **UseAuthentication** и **app.UseAuthorization()**, как и в любой системе аутентификации и авторизации в ASP.NET Core.

**Аутентификация и авторизация в практике**

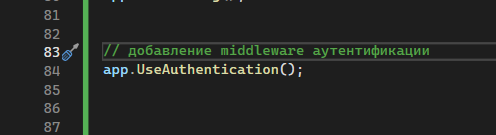
Добавление сервисов аутентификаций реализованы как методы расширения для типа AuthenticationBuilder, который возвращается методом AddAuthentication():



С помощью метода AddJwtBearer() в приложение добавляется конфигурация токена. Для конфигурации токена применяется объект JwtBearerOptions, который позволяет с помощью свойств настроить работу с токеном. Данный объект имеет множество свойств.

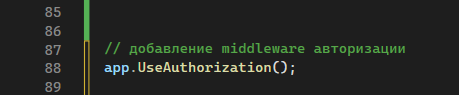
Метод **UseAuthentication()** должен встраиваться в конвейер до любых компонентов middleware, которые используют аутентификацию пользователей.

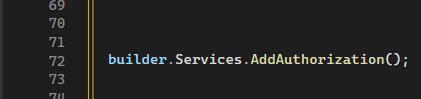
Для выполнения аутентификации этот компонент использует сервисы аутентификации, в частности, сервис **IAuthenticationService**, которые регистрируются в приложении с помощью метода **AddAuthentication()**:

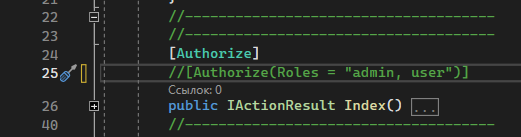


Для подключения **авторизации** необходимо встроить компонент **Microsoft.AspNetCore.Authorization.AuthorizationMiddleware**. Для этого применяется встроенный метод расширения **UseAuthorization()**

Кроме того, для применения авторизации необходимо зарегистрировать сервисы авторизации с помощью метода **AddAuthorization()**:



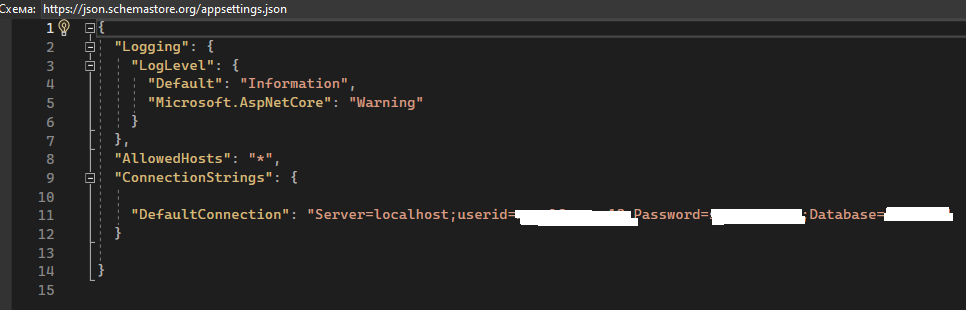




Применение данного атрибута **Authorize** означает, что имеют доступ только аутентифицированные пользователи. В данном случае атрибут применяется в контроллере HomeController, главной страницы пользователя.

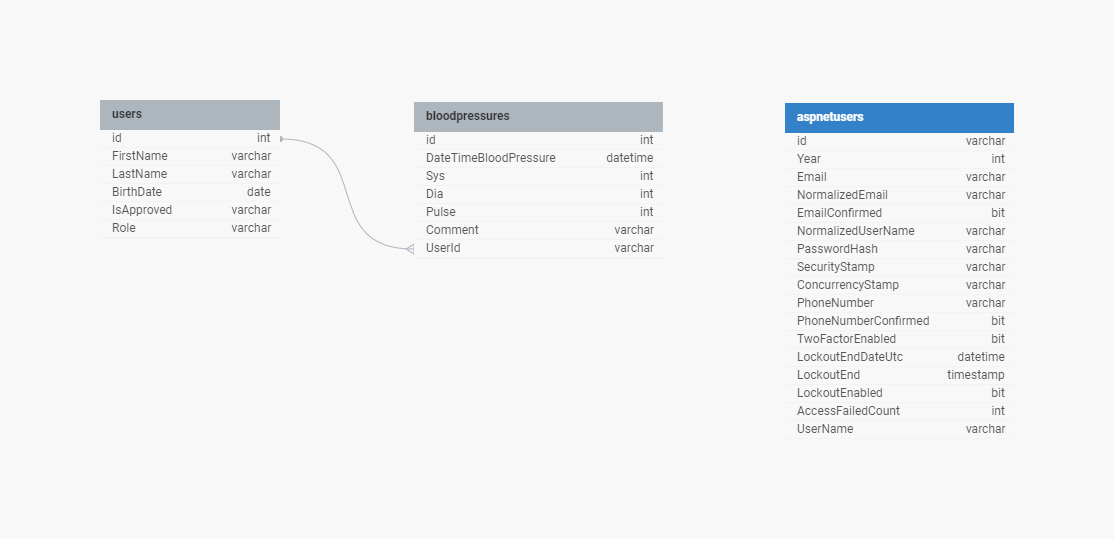
**Подключение к существующей базе данных**

Чтобы подключаться к базе данных, нам надо задать параметры подключения. Для этого изменим файл **appsettings.json**. По умолчанию он содержит только настройки логирования:



Добавленная разметка определяет строку подключения по имени ConnectionStrings, в которой указывается база данных и аутентификационные данные.

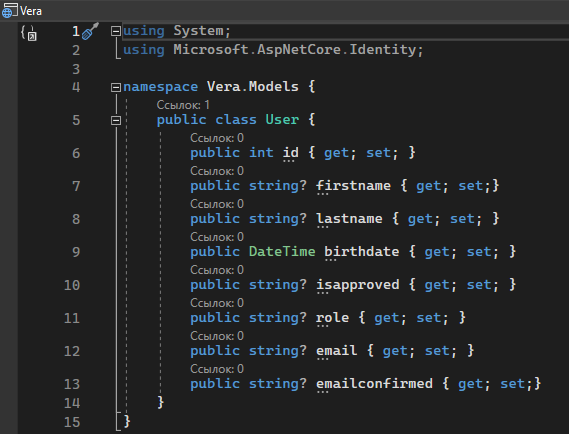
**Структура базы данных**

****

**Добавление модели и контекста данных IdentityDbContext**

Для работы с базами данных через Entity Framework Core необходим пакет MySql.EntityFrameworkCore.

Далее добавим в проект класс, которые будет представлять данные. Будет называться **User** и будет иметь следующий код:



Этот класс представляет сущность, которые будут храниться в базе данных.

Для взаимодействия с базой данных через Entity Framework Core.

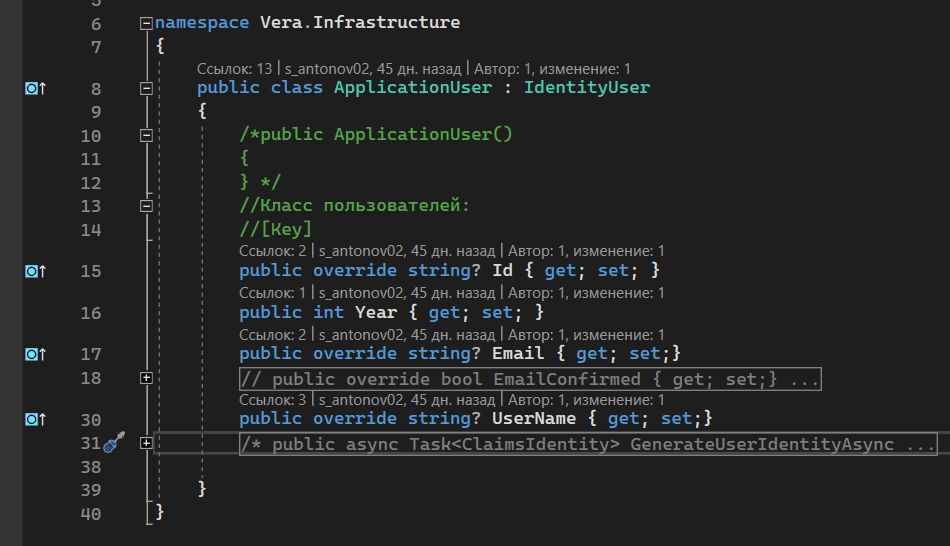
Добавим в папку Infrastructure новый класс, который назовем **VeraContext.**

Для взаимодействия с базой данных в пространстве имен Microsoft.AspNet.Identity.EntityFramework определен класс **IdentityDbContext**. Это обычный контекст данных, производный от **DbContext**, который уже содержит свойства, необходимые для управления пользователями и ролями: свойства Users и Roles. Хотя в реальном приложении мы будем иметь дело с классами, производными от IdentityDbContext.

Функциональность пользователей в AspNet Identity сосредоточена в классе **IdentityUser**, который определен в пространстве имен Microsoft.AspNet.Identity.EntityFramework. **IdentityUser** реализует интерфейс **IUser** и определяет следующие свойства:

* **Claims**: возвращает коллекцию claims - специальных атрибутов, которыми обладает пользователь и которые хранят о пользователе определенную информацию
* **Email**: email пользователя
* **Id**: уникальный идентификатор пользователя
* **Logins**: возвращает коллекцию логинов пользователя
* **PasswordHash**: возвращает хэш пароля
* **Roles**: возвращает коллекцию ролей пользователя
* **PhoneNumber**: возвращает номер телефона
* **SecurityStamp**: возвращает некоторое значение, которое меняется при каждой смене настроек аутентификации для данного пользователя
* **UserName**: возвращает ник пользователя
* **AccessFailedCount**: число попыток неудачного входа в систему
* **EmailConfirmed**: возвращает true, если email был подтвержден
* **PhoneNumberConfirmed**: возвращает true, если телефонный номер был подтвержден
* **TwoFactorEnabled**: если равен true, то для данного пользователя включена двухфакторная авторизация

Как правило, для управления пользователями определяют класс, производный от IdentityUser:



Чтобы создать контекст, нам надо унаследовать новый класс от класса **IdentityDbContext**, с обобщенным классом **ApplicationUser**. Для работы с базой данных ASP NET Identity использует контекст данных, который наследуется от класса **IdentityDbContext** из пространства имен Microsoft.AspNetCore.Identity.EntityFrameworkCore.

По умолчанию данный класс наследует весь функционал от IdentityDbContext. Так, мы можем получить содержимое таблиц из бд с помощью следующих свойств:

* **Users**: набор объектов IdentityUser, соответствует таблице пользователей
* **Roles**: набор объектов IdentityRole, соответствует таблице ролей
* **RoleClaims**: набор объектов IdentityRoleClaim, соответствует таблице связи ролей и объектов claims
* **UserLogins**: набор объектов IdentityUserLogin, соответствует таблице связи пользователей с их логинами их внешних сервисов
* **UserClaims**: набор объектов IdentityUserClaim, соответствует таблице связи пользователей и объектов claims
* **UserRoles**: набор объектов IdentityUserRole, соответствует таблице, которая сопоставляет пользователей и их роли
* **UserTokens**: набор объектов IdentityUserToken, соответствует таблице токенов пользователей

По умолчанию таблица, которая соответствует определенному набору, называется по имени набор плюс префикс AspNet. Доступ к таблице пользователей в контексте кода сущности User, находиться по ссылке DbSet<T>  UserProfiles.

Если нам надо хранить в базе данных объекты каких-то других классов, то в классе контекста можно определить для них свойство по типу DbSet<T>.

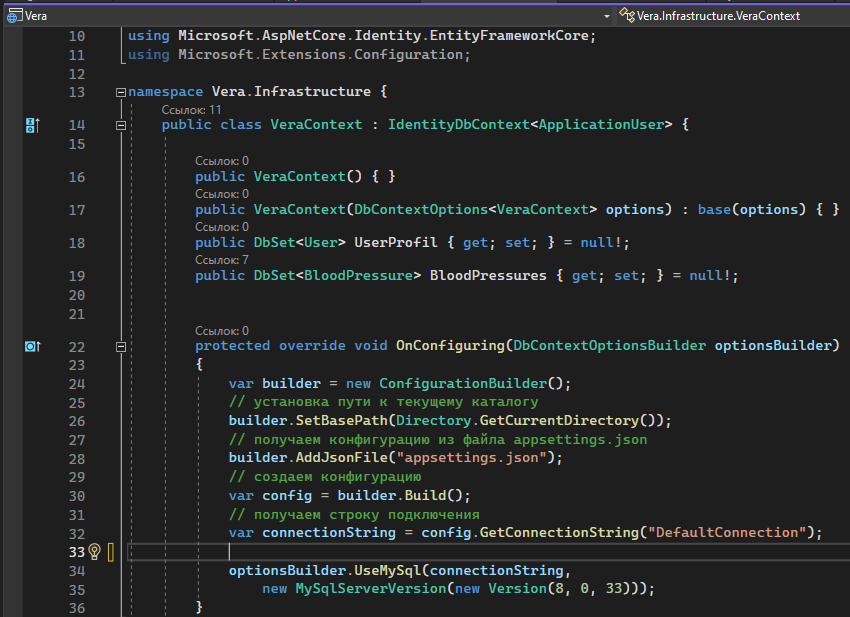
Свойства наподобие public DbSet< BloodPressure > BloodPressures { get; set; } помогают получать из БД набор данных определенного типа (например, набор объектов BloodPressure). Фактически каждое свойство DbSet будет соотноситься с отдельной таблицей в базе данных, что и фактически в коде реализовано.

В методе OnConfiguring () в приложение добавляются сервисы Identity:

Во-первых, здесь добавляются сервисы Entity Framework, которые используются для работы с базой данных, хранящей учетные записи. Все настройки подключения заданы в файле appsettings.json в узле GetConnectionString.

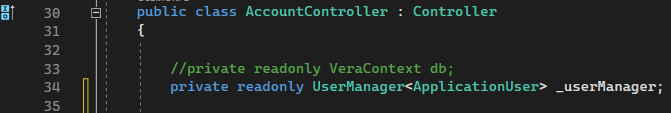
В перегруженном методе **OnConfiguring()**, следующие настройки:

* Установка пути к текущему каталогу.
* Получение конфигураций из файла appsettings.json.
* Получение строки подключения.
* Подключения Mysql сервера баз данных.



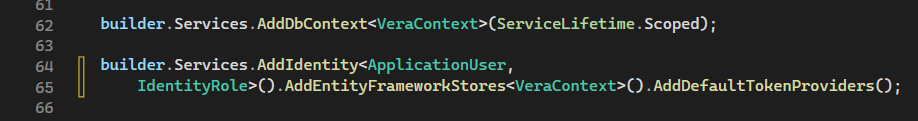
**Добавление менеджера пользователей UserManager**

После добавления сервисов Identity в классе AccountController мы можем получить объект UserManager через механизм внедрения зависимостей, например, через конструктор:



В классе Program добавляются сервисы для Entity Framework Core:

Далее добавляются специфичные для Identity сервисы:

****

Метод AddIdentity() позволяет установить некоторую начальную конфигурацию. Здесь мы указываем тип пользователя и тип роли, которые будут использоваться системой Identity. В качестве типа пользователя выступает созданный нами выше класс ApplicationUser, а в качестве типа роли взят стандартный класс IdentityRole, который находится в пространстве имен Microsoft.AspNetCore.Identity.EntityFrameworkCore.

Метод AddEntityFrameworkStores() устанавливает тип хранилища, которое будет применяться в Identity для хранения данных. В качестве типа хранилища здесь указывается класс контекста данных.

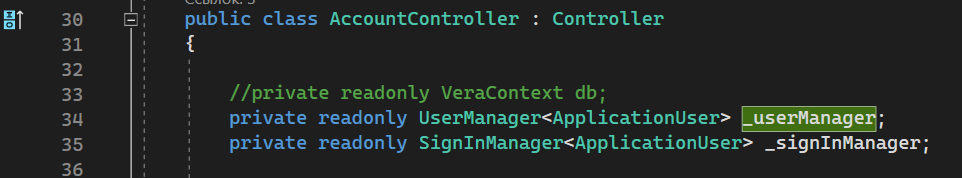
Затем, чтобы использовать Identity, в классе Program устанавливается компонент middeware - UseAuthentication.

Теперь система Identity подключена в проект, и мы можем с ней работать.

**Добавление, регистрация и создание пользователей в Identity**

Продолжим работу с проектом и добавим в него функционал регистрации пользователей.

Для работы с учетными записями пользователей добавим в контроллер AccountController и определим в нем метод для регистрации пользователей:

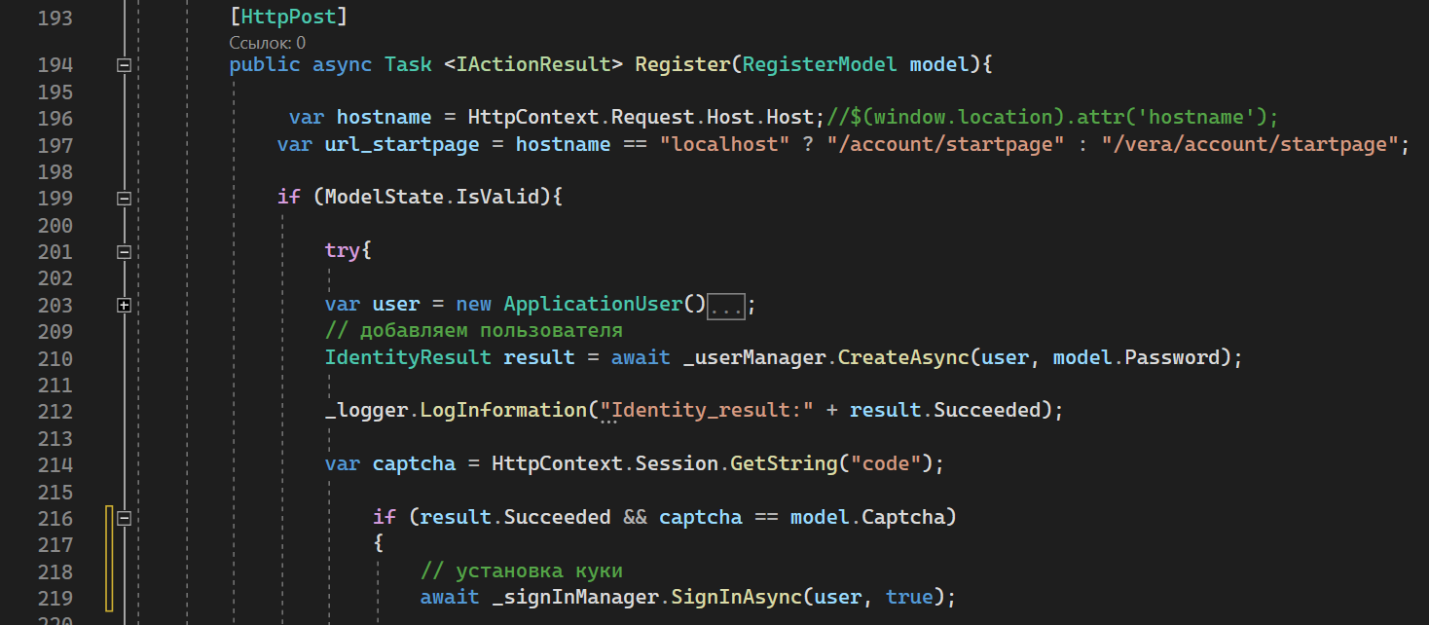


**Регистрация и создание пользователей в Identity**

Поскольку в классе Program были добавлены сервисы Identity, то здесь в контроллере через конструктор мы можем их получить. В данном случае мы получаем сервис по управлению пользователями - UserManager и сервис SignInManager, который позволяет аутентифицировать пользователя и устанавливать или удалять его куки.

С помощью метода \_userManager.CreateAsync пользователь добавляется в базу данных. В качестве параметра передается сам пользователь и его пароль.

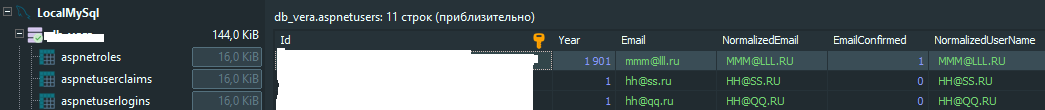
Данный метод возвращает объект IdentityResult, с помощью которого можно узнать успешность выполненной операции. Вполне возможно, что переданные значения не удовлетворяют требованиям, и тогда пользователь не будет добавлен в базу данных. В случае удачного добавления с помощью метода \_signInManager.SignInAsync() устанавливаем аутентификационные куки для добавленного пользователя. В этот метод передается объект пользователя, который аутентифицируется, и логическое значение, указывающее, надо ли сохранять куки в течение продолжительного времени. И далее выполняем переадресацию на главную страницу приложения.



Если добавление прошло неудачно, то добавляем к состоянию модели с помощью метода ModelState все возникшие при добавлении ошибки, и отправленная модель возвращается в представление.

После удачной регистрации нас переадресует на главную страницу.

А в самой базе данных в таблице **dbo.AspNetUsers** мы сможем увидеть добавленного пользователя.

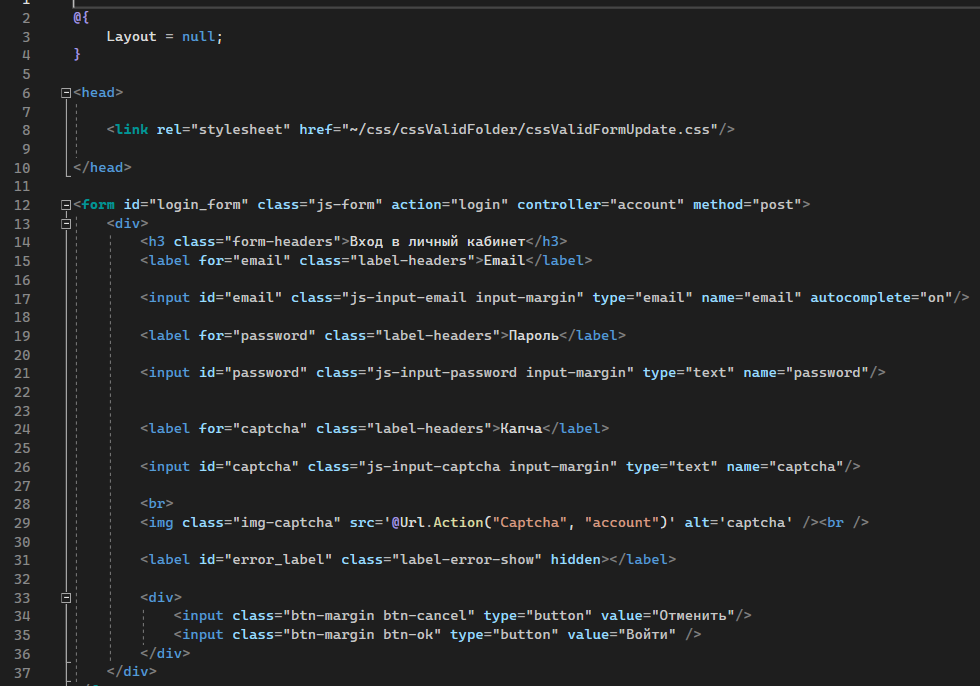


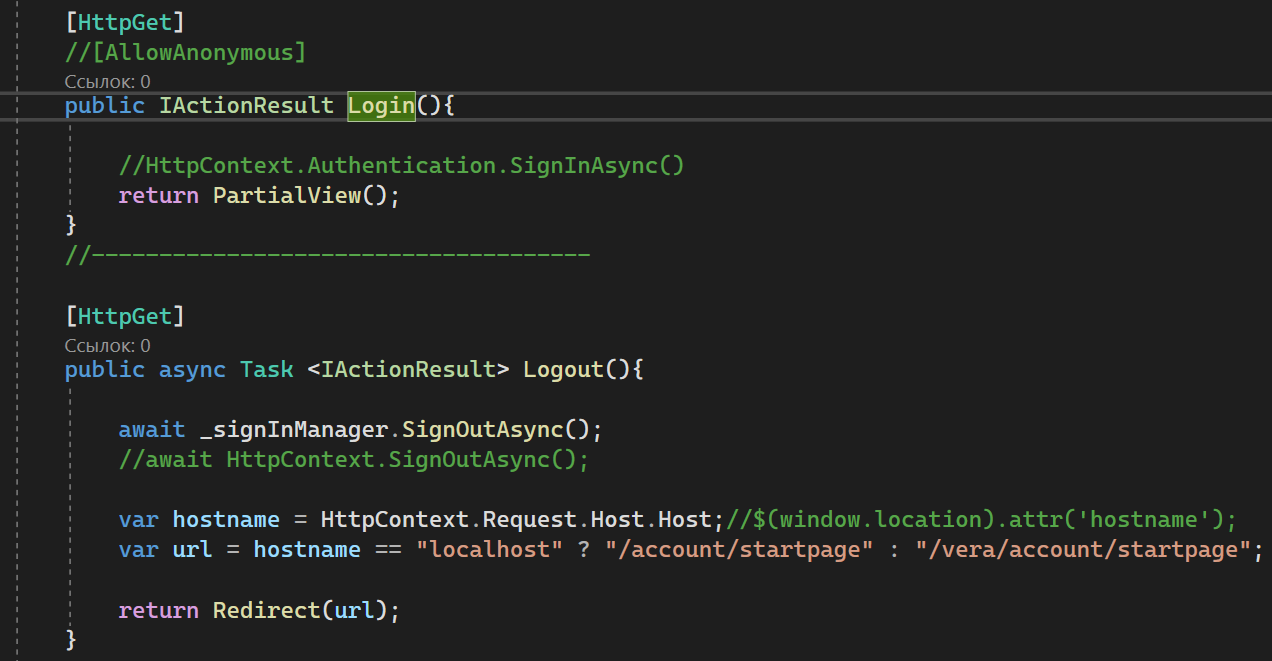
**Добавление авторизация пользователей в Identity**

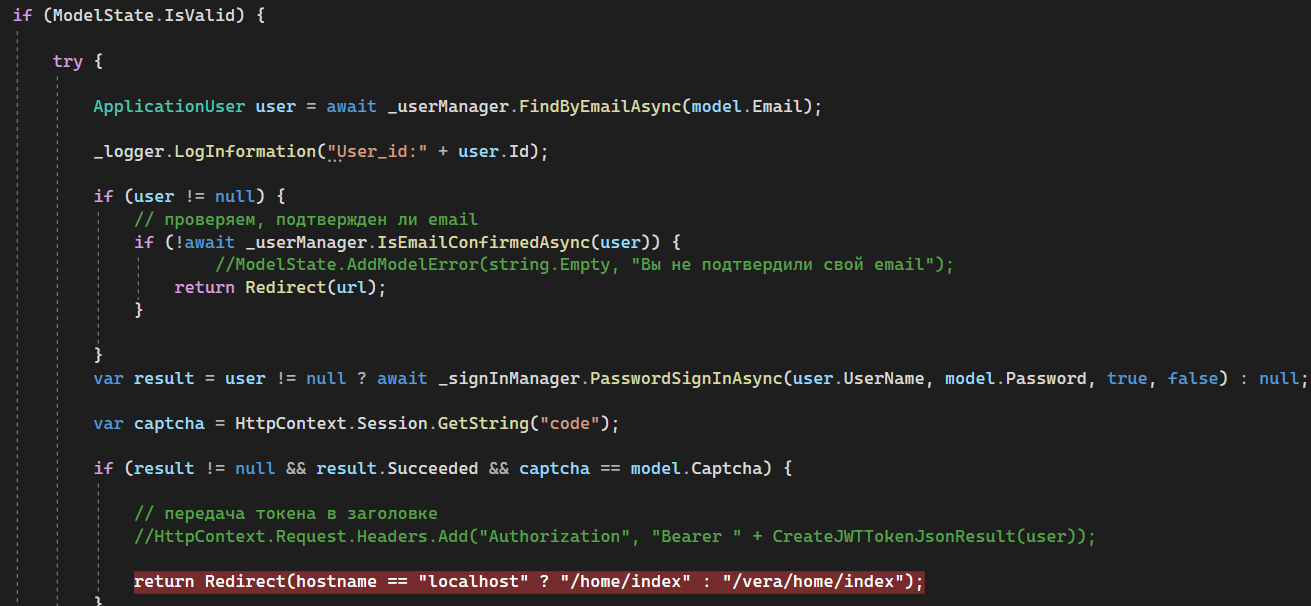
Теперь добавим в AccountController следующие три метода:

Login (Get), Logout (Get).

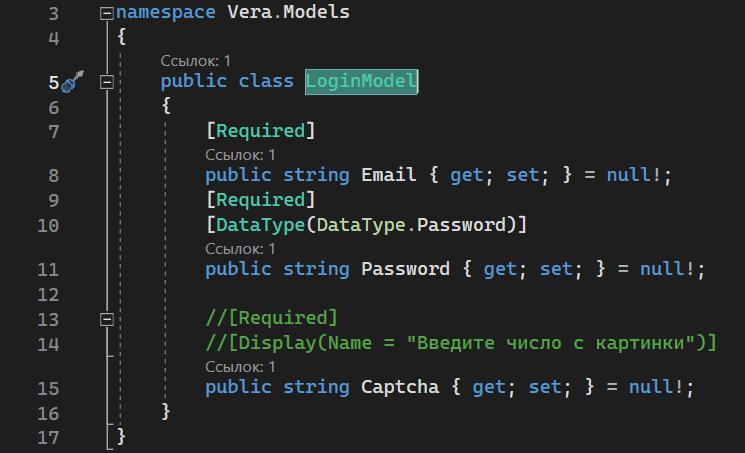
В Get-версии метода Login мы получаем форму авторизаций.







Метод FindByEmailAsync() содержащий результаты асинхронной операции по поиску пользователя, если таковой имеется по адресу электронной почты. Проверяем, подтвержден ли email при регистраций пользователя в базе данных с помощью метода IsEmailConfirmedAsync(), а если нет , то происходит перенаправление запроса к исходному состоянию.



В Post-версии метода Login получаем данные из представления в виде модели LoginModel. Всю работу по аутентификации пользователя выполняет метод signInManager.PasswordSignInAsync(). Этот метод принимает логин и пароль пользователя. Третий параметр метода указывает, надо ли сохранять устанавливаемые куки на долгое время.

Данный метод также возвращает IdentityResult, с помощью которого можно узнать, завершилась ли аутентификация успешно.

Следующий код из текущей сессий присваивает сгенерированную капчу в переменную captcha:

var captcha = HttpContext.Session.**GetString**("code");

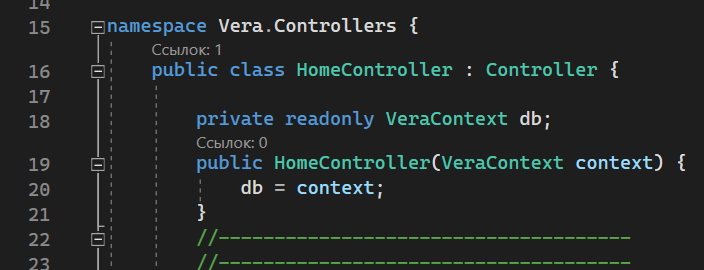
При выполнений условий проверки пользователя и введенную капчу пользователем, наш запрос переадресовывается на главную страницу пользователя, а иначе переадресовывает на стартовую страницу входа.

Третий метод - метод **Logout()** выполняет выход пользователя из приложения. За выход отвечает метод \_signInManager.**SignOutAsync(),** который очищает аутентификационные куки.

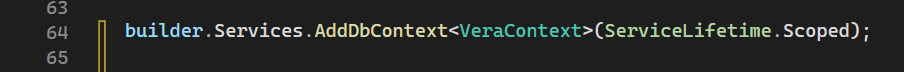
**CRUID – операций. Создание контроллера основного интерфейса пользователя**

Итак, начальные данные определены, и теперь мы хотим их выводить на веб-страницу, чтобы пользователи добавляли записи и смогли бы их видеть в виде таблиц и графики.

Для этого перейдем к папке Controllers, где создадим контроллер HomeController. Изменим его содержимое следующим образом:

****

Добавлен конструктор, в котором получаем контекст данных. Здесь применяется встроенный механизм внедрения зависимостей. Так как, в классе Program.cs контекст данных был добавлен в качестве сервиса:

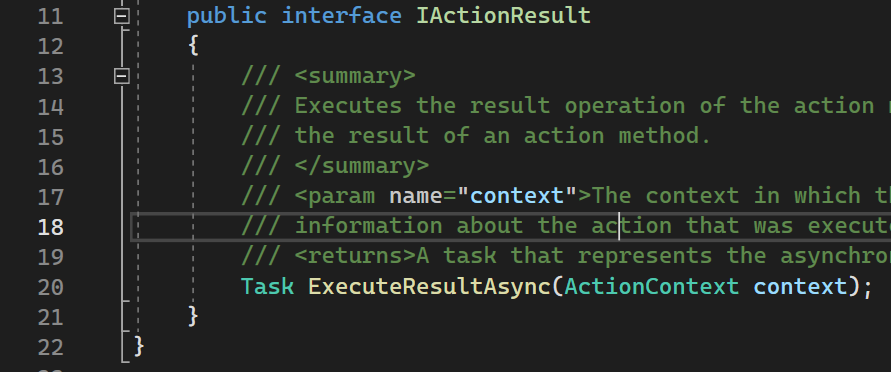


То через конструктор мы можем получить эту зависимость и можем использовать ее.

**Описание результатов действий**

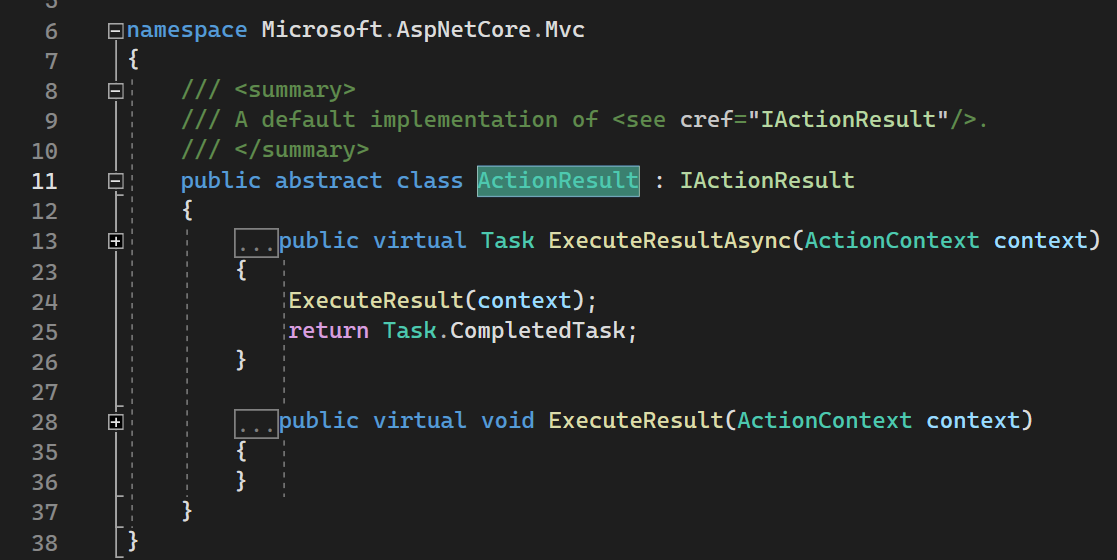
При обращении к веб-приложению, как правило, пользователь ожидает получить некоторый ответ, например, в виде веб-страницы, которая наполнена данными. На стороне сервера метод контроллера, получая параметры и данные запроса, обрабатывает их и формирует ответ в виде результата действия. Результат действия - это тот объект, который возвращается методом после обработки запроса.

Объекты типа **IActionResult**, которые непосредственно предназначены для генерации результата действия. Интерфейс IActionResult находится в пространстве имен Microsoft.AspNetCore.Mvc и определяет один метод.



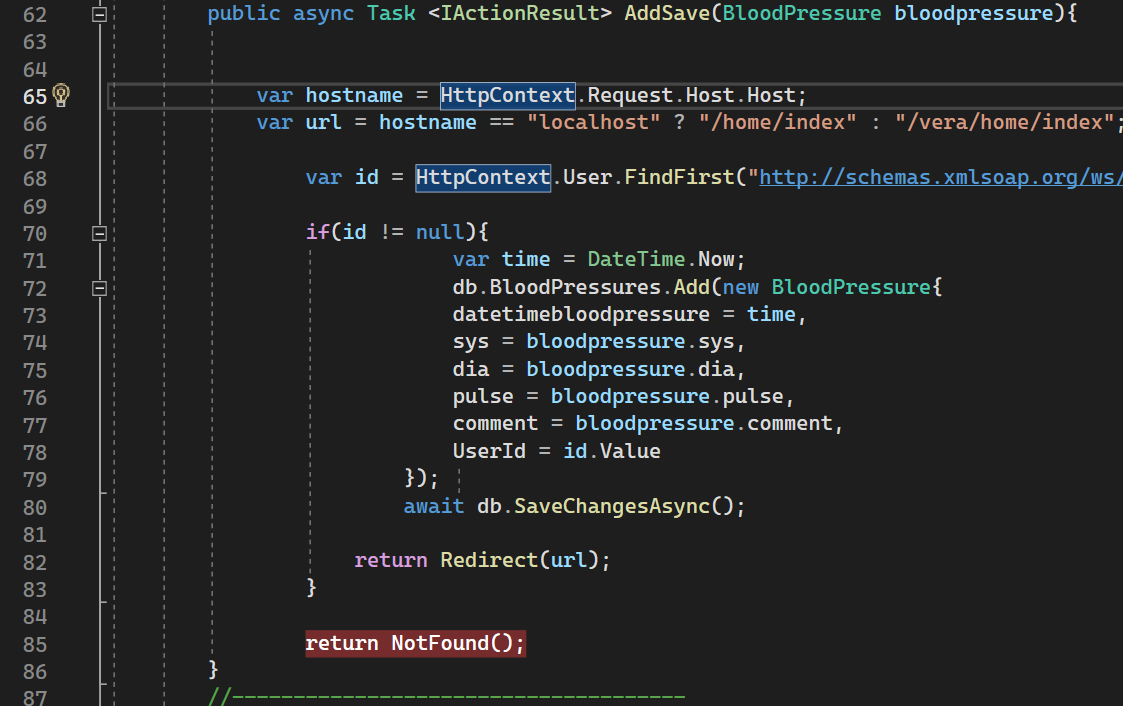
Метод ExecuteResultAsync() принимает контекст действия и выполняет генерацию результата.

Этот интерфейс затем реализуется абстрактным базовым классом **ActionResult**:



ActionResult тип метода, который выполняется асинхронно. И если мы вдруг захотим создать свой класс результата действий, то как раз можем либо унаследовать его от ActionResult, либо реализовать интерфейс IActionResult.

**Добавление, обновление, удаление пользователя в базе данных**



Добавляем асинхронный метод AddSave(), добавления и сохранение новых значений в базу данных , который представляет собой асинхронную операцию и реализует интерфейс IActionResult, выполняет операцию с результатом метода action асинхронно. Этот метод вызывается MVC для обработки результата метода action.

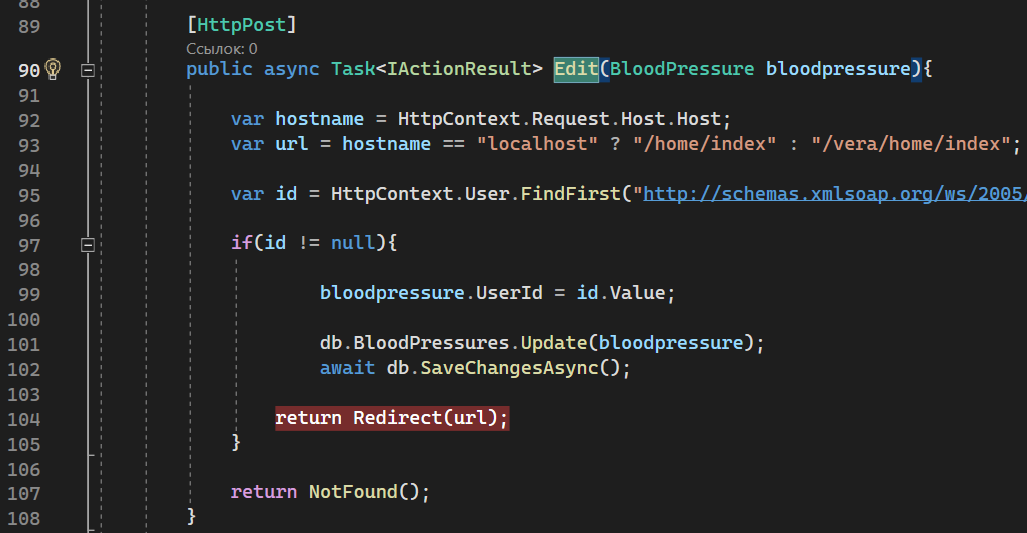
Далее в сущность BloodPressures, добавляем новый объект BloodPressure с новыми значениями.

Метод Add устанавливает значение Added в качестве состояния нового объекта. Поэтому метод db.SaveChanges() сгенерирует выражение INSERT для вставки модели в таблицу.

Затем вызывается метод Redirect(), который перенаправляет запрос.

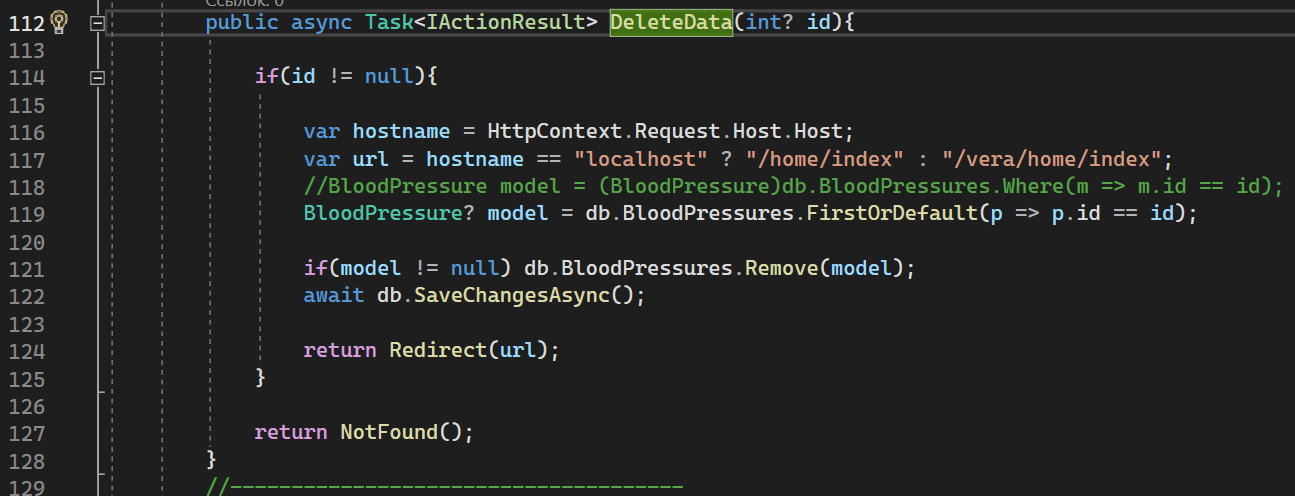
**Редактирование**

При изменении объекта Entity Framework сам отслеживает все изменения, и когда вызывается метод **SaveChanges()**, будет сформировано SQL-выражение UPDATE для данного объекта, которое обновит объект в базе данных.



**Удаление**

Удаление производится с помощью метода **Remove**:



Данный метод установит статус объекта в Deleted, благодаря чему Entity Framework при выполнении метода db.SaveChanges() сгенерирует SQL-выражение DELETE.

**Передача данных в контроллер через формы**

При обращении к методу пользователь увидит в браузере форму ввода. При нажатии ссылки на метод **Edit** введенные данные будут отправляться на сервер. Поскольку у элемента <form> не задан атрибут action, который устанавливает адрес, то введенные данные отправляются на тот же адрес (то есть, по сути, методу с тем же именем - методу Edit). Но поскольку у формы установлен атрибут method='post', то данные будут отправлять в запросе типа POST. А запросы данного типа обрабатывает с атрибутом **HttpPost**.

Чтобы система могла связать параметры метода и данные формы, необходимо, чтобы **атрибуты** у полей формы соответствовали названиям параметров

**Создание контроллера Web API**

Далее добавим в проект новую папку **Controllers**, а в ней создадим новый api-контроллер. Для этого при добавлении нового элемента в проект можно использовать шаблон API Controller Class.

Прежде всего к контроллеру применяется атрибут **[**ApiController**]**, который позволяет использовать ряд дополнительных возможностей, в частности, в плане привязки модели и ряд других. Также к контроллеру применяется атрибут маршрутизации, который указывает, как контроллер будет сопоставляться с запросами.

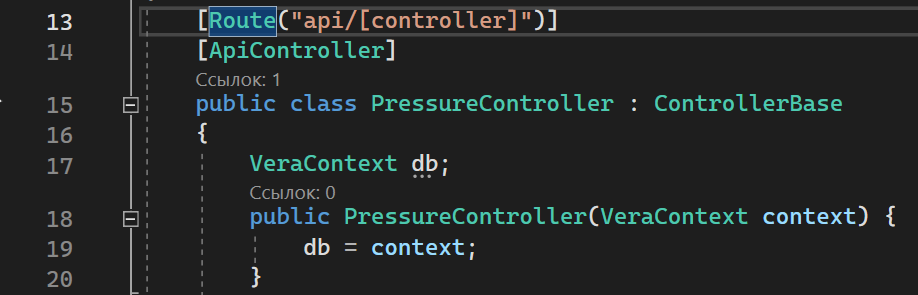
В конструкторе контроллера получаем контекст данных и используем его для операций с данными. Также в конструкторе контроллера добавляем ряд начальных данных.

Чтобы подключить маршрутизацию контроллеров на основе атрибутов, нужно у класса прописать атрибут. После этого мы сможем обращаться к контроллеру через запрос api/pressure, поскольку к контроллеру применяется атрибут маршрутизации [Route("api/[controller]")], где параметр "controller" указывает на название контроллера.

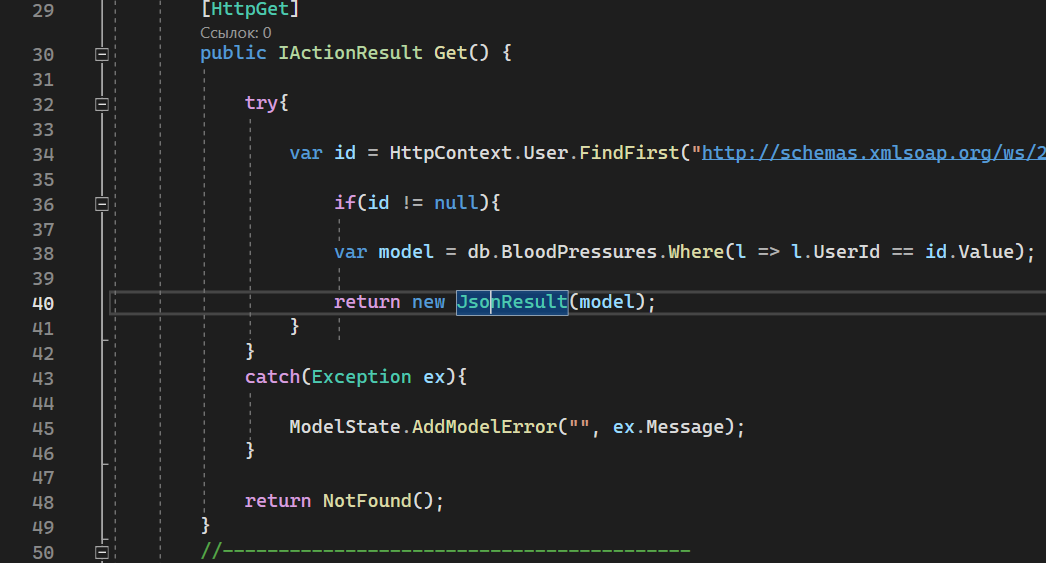
Контроллер представляет обычный класс, который наследуется от базового класса **System.Web.Mvc.Controller**. В свою очередь класс Controller реализует абстрактный базовый класс ControllerBase, а через него и интерфейс **IController**.

Назовем новый элемент **PressureController**.

После его создания напишем код следующим образом:



Таким образом, формально, чтобы создать свой класс контроллера, достаточно создать класс, реализующий интерфейс IController и имеющий в имени суффикс Controller.



Метод Get реализует интерфейс IActionResult, который описывал выше и который имеет атрибут HttpGet, который в свою очередь делает Get-запрос по протоколу Http. Метод представляет из себя запрос возвращает первый объект claim, который соответствуют определенному типу или условию var id = HttpContext.User.FindFirst (), к базе данных и возвращает id пользователя. Далее проверяем на null. При обращении к методу Get() были переданы параметры id и если определенный параметр был передан, то к объекту IQueryable, представляющего выборку, добавляется выражение в виде лямбды:

var model = db.BloodPressures.**Where**(l => l.UserId == id.Value)

Который фильтрует и находит замеры пользователя. Метод **Get ()**, возвращает **JsonResult** который отправляет клиенту ответ в формате json, а если срабатывает блок catch () {ModelState.**AddModelError()**}, то в браузере на клиенте увидим ошибку. Объект **ModelState** сохраняет все значения, которые пользователь ввел для свойств модели, а также все ошибки, связанные с каждым свойством и с моделью в целом.

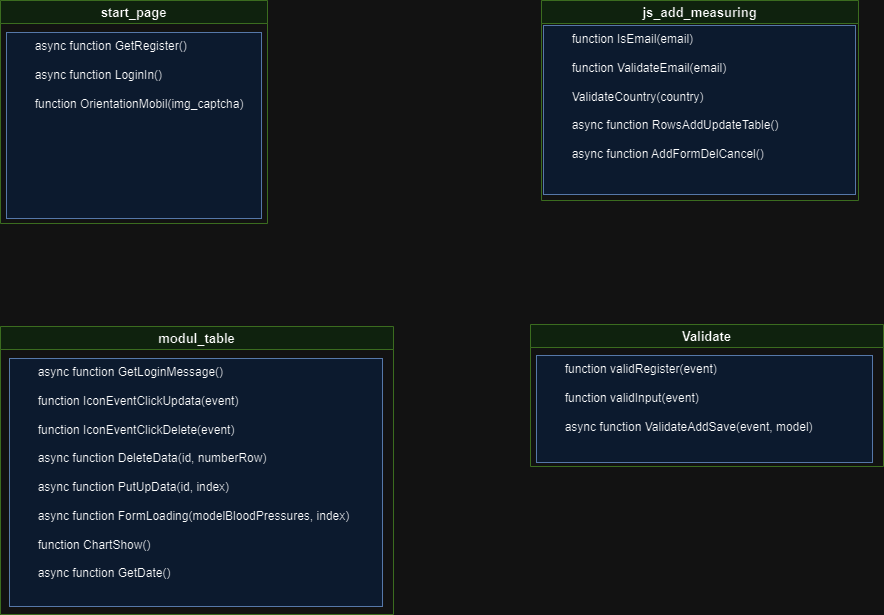
**NotFoundResult** и **NotFoundObjectResult** посылает код 404, уведомляя браузер о том, что ресурс не найден. Второй класс в дополнении к статусному коду позволяет отправить дополнительную информацию, которая потом отобразится в браузере.

Объекты обоих классов создаются методом **NotFound**. Для первого класса - это метод без параметров, для второго класса - метод, который в качестве параметра принимает отправляемую информацию.

Метод **GetDate()** в целом делает тоже, что и **Get()** метод, только в аргументах метода задаются две даты на оснований которых фильтруются сущность BloodPressures.

**Frontend. Элементы и атрибуты HTML5**

**Структура на клиенте:**

****



Внутри элемента html мы можем разместить два других элемента: **head** и **body**. Элемент head содержит метаданные веб-страницы - заголовок веб-страницы, тип кодировки и т.д., а также ссылки на внешние ресурсы - стили, скрипты, если они используются. Элемент body, собственно, определяет содержимое html-страницы.

В элементе head определены элементы:

* элемент meta определяет метаинформацию страницы. Для корректного отображения символов предпочтительно указывать кодировку. В данном случае с помощью атрибута charset="utf-8" указываем кодировку utf-8.

Элемент meta определяет метаданные документа.

Чтобы документ корректно отображал текст, необходимо задать кодировку с помощью атрибута charset. Рекомендуемой кодировкой является utf-8.

Мета тег <meta name="viewport"> отвечает за размер области просмотра и масштаб страницы на мобильных устройствах. Мета тег <meta name="viewport"> размещаем в HTML файле в секции <head></head>.

Чаще всего используется значение device-width, что означает «использовать ширину экрана текущего устройства». <meta name="viewport" content="width=device-width"> initial-scale — задает начальный масштаб страницы. Значение 1 означает, что страница будет отображена без масштабирования.

Основная часть документа html, фактически все, что мы увидим в своем браузере при загрузке веб-страницы, располагается между тегами **<body>** и **</body>**. Здесь размещаются большинство элементов html.

Стартовая страница start\_page.js в теге <head> имеет следующий вид:



**Создание Vue.js**

<div id="containerId" class="container body-content">, containerId указывает на объект Vue.js ссылкой vue\_start\_page. Который имеет два метода RegisterForm(), LoginEnter().

Следующий блок кода:

<div class="col-md-offset-10 text-align-center">

<**img** class="img-icon-login-startpage" src="~/images/user-login-icon.png"/><br><br>

<a v-on:click="LoginEnter()" class="login-in">Вход</a>

</div>

Специальные классы col-md(sm | lg) - offset-\* позволяют задать смещение относительно левого блока или начала строки в условных единицах с помощью библиотеки bootstrap.

В классе text-align-center, реализован свойство text-align: center,

определяет горизонтальное выравнивание текста в пределах элемента.

Следующим кодом подгружаем иконку в формате file.png:

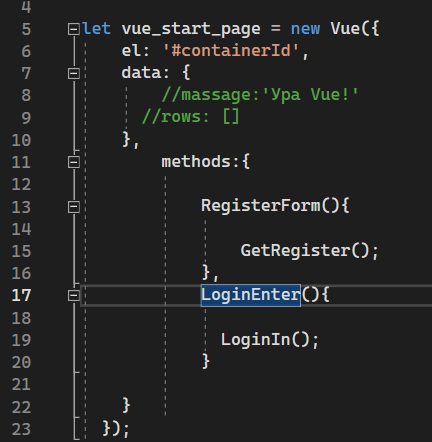
<img class="img-icon-login-startpage" src="~/images/user-login-icon.png"/> и с помощью ссылки на «вход» в сервис отрабатывает событие «click» Vue.js библиотеки, который запускает метод LoginEnter(), файла start\_page.js.

LoginIn() метод, подгружает форму в виде Get-запроса /account/login с помощью асинхронного метода :

let responseGet = await fetch( url, {

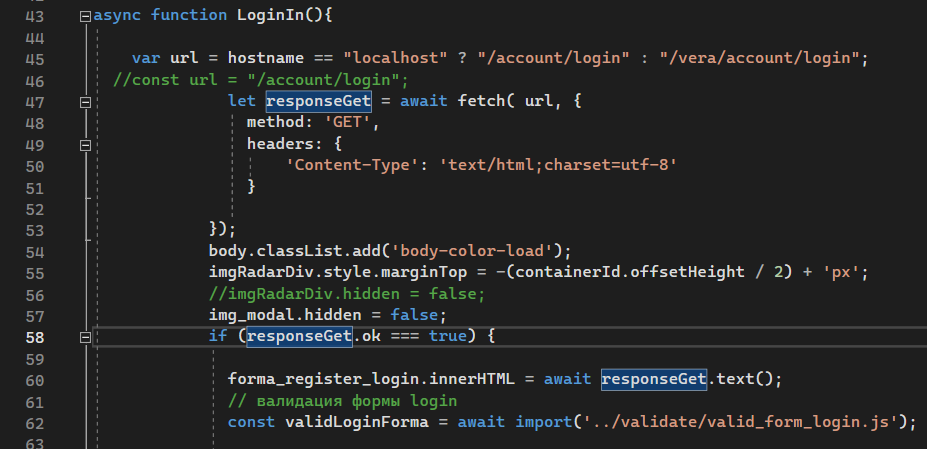
method: 'GET',

headers: { 'Content-Type': 'text/html;charset=utf-8' } });

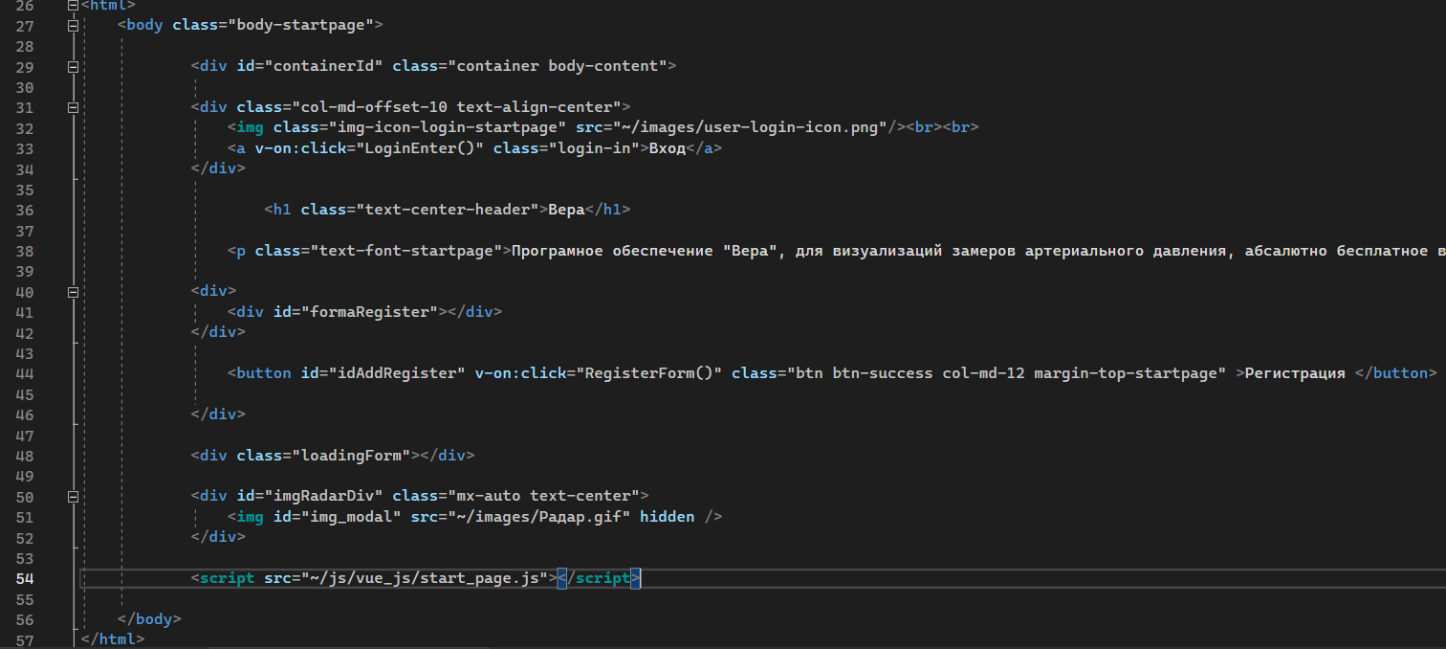


**Создание метода LoginIn(), входа в систему**

В методе делаем Get запрос помощью объекта fetch(), на адрес /account/login, который возвращает форму авторизаций.



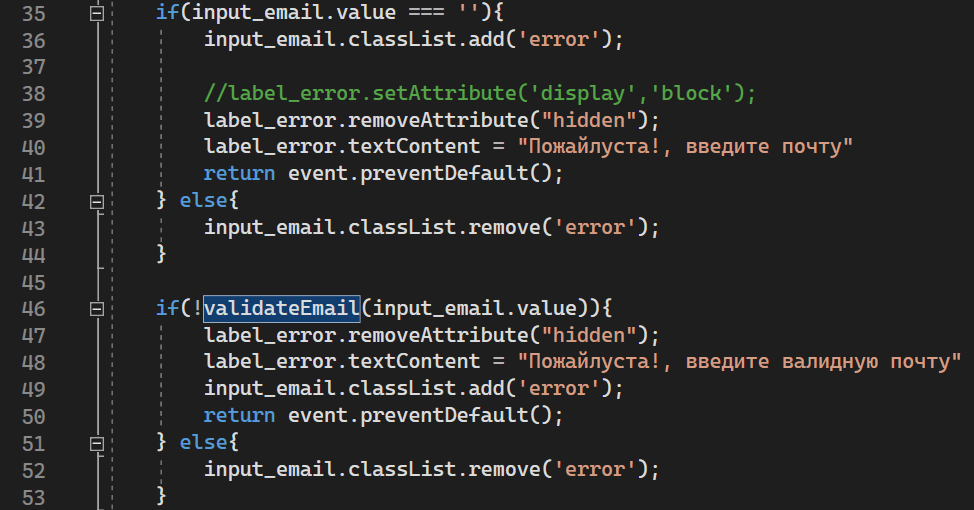
При получений ответа код статусом 200, вставляем в стартовую страницу в тег блока <div> c id= formaRegister, форму регистраций.



**Валидация формы login**

В полях ввода данных проверяется на ввод пустых значений.

Метода validateEmail () проверяет корректность в веденой email почты при нажатиях кнопки на вход и регистрацию пользователя, проверяет с помощью регулярного выражения:

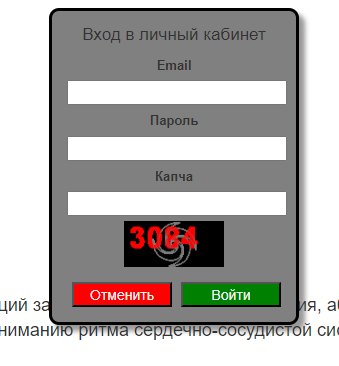


Добавляем в форму css стили.

**Оправка формы Post запросом**

С помощью кнопки «Войти» интерфейса входа в систему, javascript генерирует событие **click:**

btn\_ok.onclick = async function(event){ }



Объекты **FormData** используются, чтобы взять данные из HTML-формы и отправить их с помощью fetch или другого метода для работы с сетью.

**FormData** отправка HTML-форм: с файлами и без, с дополнительными полями и так далее. Объекты **FormData** помогут нам с этим. Это объект, представляющий данные HTML формы.

Конструктор:

const form = new **FormData**(forma\_register\_login.children.login\_form);

Если передать в конструктор элемент HTML-формы form, то создаваемый объект автоматически прочитает из неё поля.

Его особенность заключается в том, что методы для работы с сетью, например fetch, позволяют указать объект **FormData** в свойстве тела запроса body. Он будет соответствующим образом закодирован и отправлен с заголовком Content-Type: multipart/form-data. Метод fetch отправляет POST-запрос с данными формы на адрес "/account/login" и при успешной авторизаций свойство ok вернет true, Мы можем увидеть HTTP-статус в свойствах ответа:

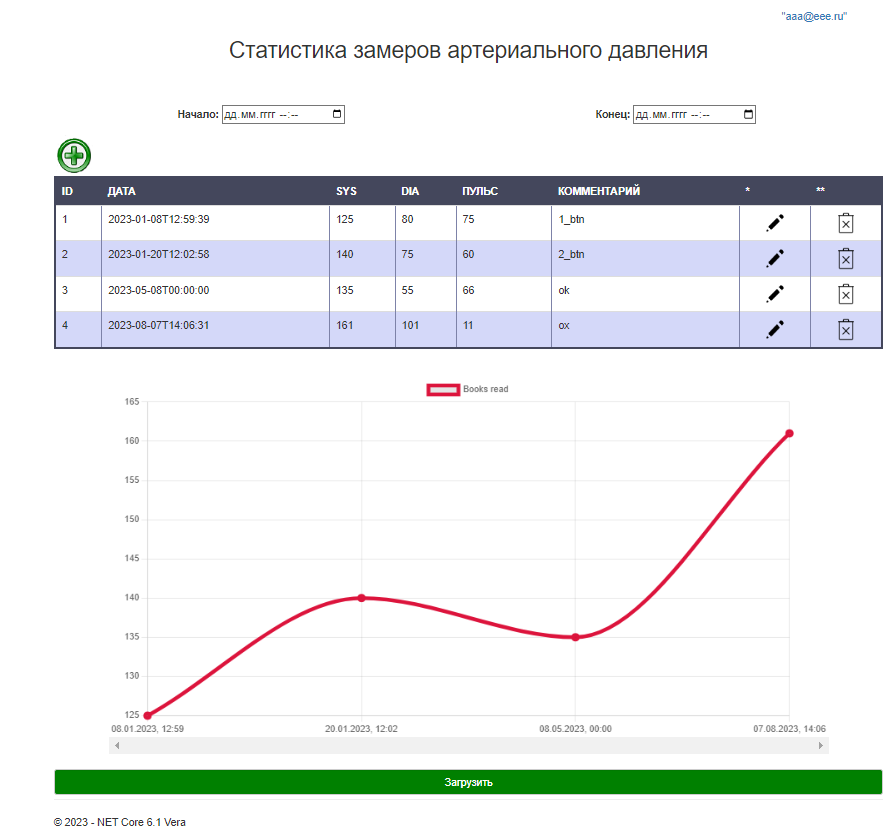
* **status** – код статуса HTTP-запроса, например 200.
* **ok** – логическое значение: будет true, если код HTTP-статуса в диапазоне 200-299.

И в зовершений переходим к текущем URL-адресу document. location = hostname == "localhost»? "/home/index»: "/vera/home/index".

Регистрация пользователя в сервисе происходит примерно по тому же сценарию.

**Интерфейс пользователя**

При входе пользователя в личный кабинет с права в верхнем углу видим логин в данном случае это адрес электронной почты.



async function GetLoginMessage(){

var hostname = $(window.location).attr('hostname');

var url = hostname == "localhost" ? "/api/pressure/login" : "/vera/api/pressure/login";

let response = await fetch( url, {

method: 'GET',

headers: {

"Accept": "application/json",

'Content-Type': 'application/json;charset=utf-8'

}

});

if (response.ok === true) {

return await response.text();

}

}

С помощью метода GetLoginMessage() на клиенте делаем запрос по адресу /api/pressure/login, который летит на сервер к контроллеру «Pressure» к методу с идентичным названием:

[HttpGet("{login}")]

public string GetLoginMessage(){

return HttpContext.User.Identity != null && HttpContext.User.Identity.Name != null ? HttpContext.User. Identity.Name : "";}

Который и возвращает имя логина.

На главной странице пользователя index. chtml в блоке div, видим, как с помощью объекта Vue.js и его свойств подгружается логин пользователя.

<div class="col-md-offset-10 text-align-center"

<a href="/vera/account/logOut" class="login-in">Выход</a><br><br>

<**img** class="img-icon-login-startpage" src="~/images/user-login-icon.png"/><br><br>

<a href="#">{{ login }}</a>

</div>

Пример выше демонстрирует две главные фундаментальные особенности Vue:

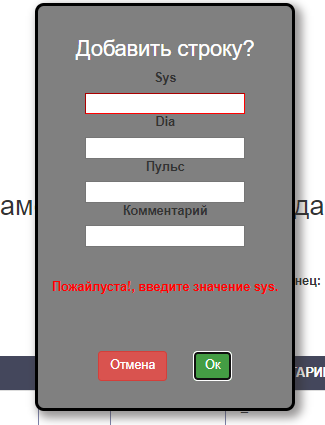
* **Декларативная отрисовка**: Vue расширяет стандартный HTML синтаксисом шаблонов, который позволяет декларативно описывать финальный HTML на основе состояния JavaScript.
* **Реактивность**: Vue автоматически отслеживает изменения состояния JavaScript и эффективно обновляет DOM, когда происходят изменения.
* **Добавление строк в таблицу**

В верхнем левом углу интерфейса пользователя находится иконка с изображением плюс.

<**img** src="~/images/png-icons-plus.png" id="img\_add\_row">

Которая имеет функционал добавление замеров в итоговою таблицу.

При нажатий иконки генерируем событие onclick и в теле подгружаем метод RowsAddUpdateTable который делает Get запрос /Home/TableAddUpdate" и возвращает частичное представление формы добавления замеров артериального давления. Также в теле происходит добавление и удаление свойств css.

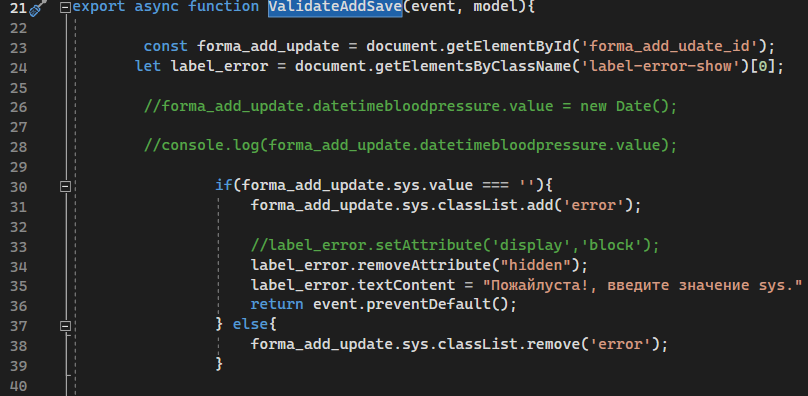


При нажатий кнопки Ok , без добавления новых замеров, срабатывает событие валидаций:

btn\_ok.onclick = async function(event){

(await validate\_add\_save).ValidateAddSave(even, result.children[1]); }

Метод ValidateAddSave() вообщем выполняет простую проверку на клиенте и если в свойствах формы будет пустое значение, то будет подгружено css свойтсво error, которое оповестит пользователя.



Если же валидация прошла успешна, то подгружаем метод GetDate(), в теле метода которого устанавливаем дату выборки загружаемых значений.

let date\_start = document.getElementById('datetime\_start');

let date\_end = document.getElementById('datetime\_end');

На Api контроллер делаем Get запрос: "/api/pressure/" и передаем два параметра в виде двух дат. Запрос обрабатывается на сервере и в ответе получаем массив json – строк в виде свойств объекта BloodPressure,

const result = await response.json();

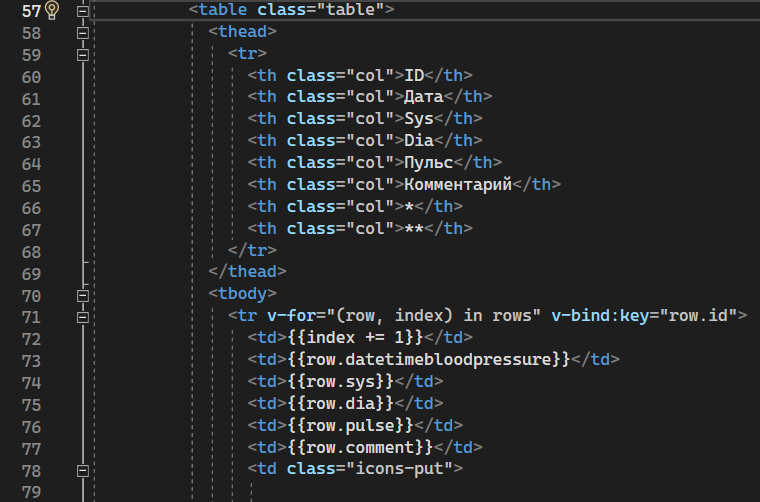
Результат вставляется в объект Vue.js, через ссылку по имени firstTable, в свойство rows с помощью метода push(), в таблицу пользователя на клиенте.

for(let x = 0; x < result.length; x++){

//таблица

firstTable.rows.push(result[x]); }

Для работы с массивами во Vue предназначена директива v-for.



При переборе элементов с помощью дополнительного параметра в v-for мы можем получить индекс элемента в массиве с помощью следующего синтаксиса:

v-for="(row, index) in rows"

Где row — это текущий перебираемый элемент в массиве rows, а index - индекс этого элемента в массиве.

Для привязки к атрибутам html-элементов предназначена директива v-bind. Для привязки к атрибуту после v-bind через двоеточие указывается непосредственно сам атрибут, к которому надо выполнять привязку.

**Редактирование строк таблицы**

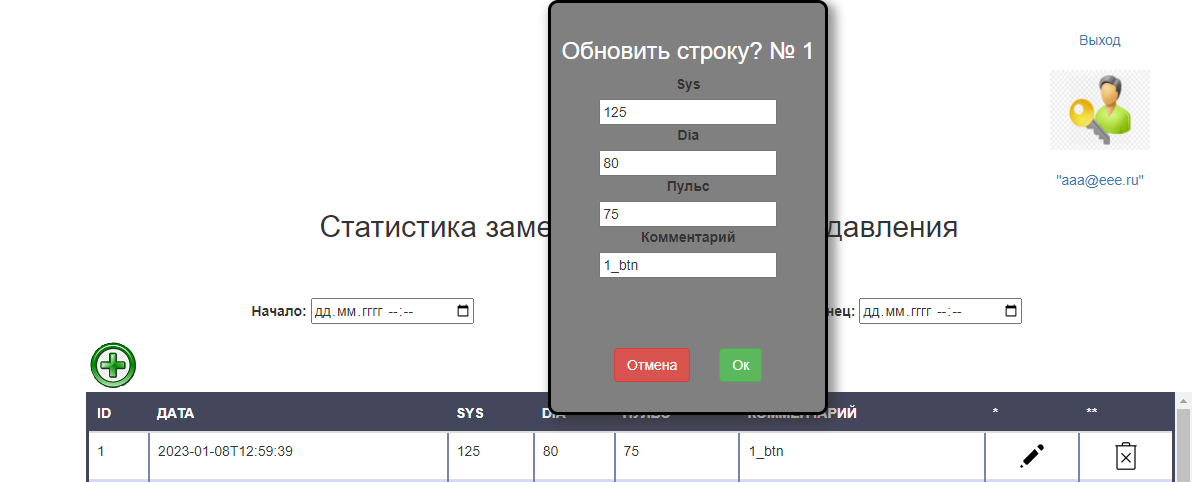
Для обработки ввода пользователя и событий элементов html в Vue применяется директива v-on, после которой через двоеточие указывается тип обрабатываемого события:

* v-on:click="действия"

В качестве типа события используется любое стандартное событие элементов html на веб-странице. Затем этой директиве в кавычках передаются те действия, которые должны выполняться при возникновении события.

<**img** v-bind:id="'btn\_up' + row.id" v-on:click="eventUpdate(row.id, $event, index)" src="~/images/png-icons-put.png" >

Метод eventUpdate() объекта Vue.js, вызывает метод PutUpData() из кода js, который делает Post запрос к серверу по адресу /api/pressure/ и передает в аргументах метода id строки таблицы на обновление:



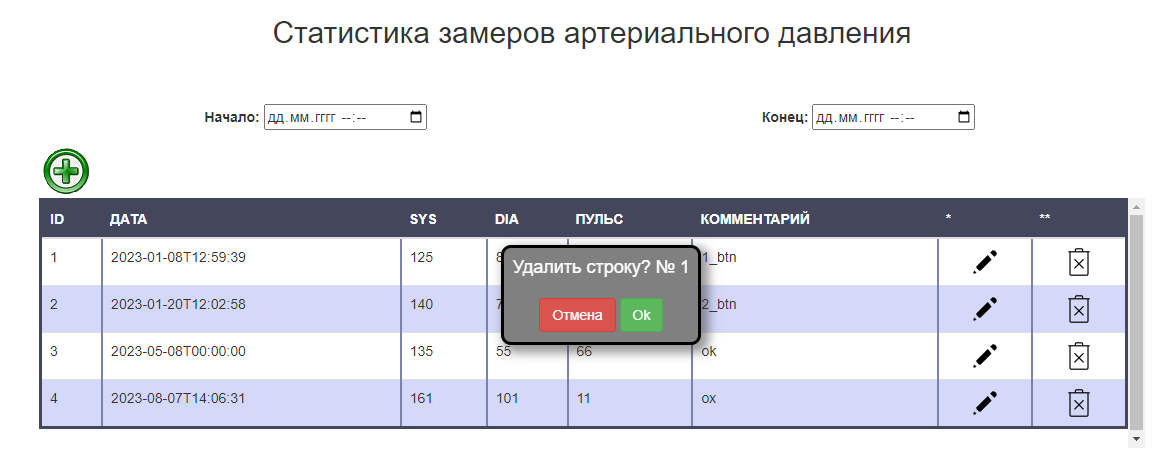
И при нажатиях кнопки Ok база данных пользователя и таблица обновляет измененные значения.

**Удаление строк в таблице**

Удаление строк из таблицы происходит по тому же сценарию:

Метод eventDelete () объекта Vue.js, вызывает метод DeleteData () из кода js, который делает Post запрос к серверу по адресу /home/deletedata и передает в аргументах метода id строки таблицы на удаление из базы данных:

<**img** v-bind:id="'btn\_del' + row.id" v-on:click="eventDelete(row.id, $event, index)" src="~/images/png-icons-delete.png">



**Начало работы с Chart.js**

Библиотека подключается на страницу как обычный JavaScript-файл. После подключения библиотеки Chart.js можно обращаться к ее методам через глобальную переменную с именем Chart. Чтобы нарисовать диаграмму необходимо на странице разместить объект canvas и создать Chart-объект с передачей ему 2d-контекста canvas. Следующий код демонстрирует подключение графической библиотеки Chart.js, скриптовых файлов данных и создание диаграммы :

<**script** src="~/js/chart.min.js"></**script**>

Ниже таблицы в фале html, подгружает блок кода, который и визуализирует графику:

<div class="app-chart\_\_canvas">

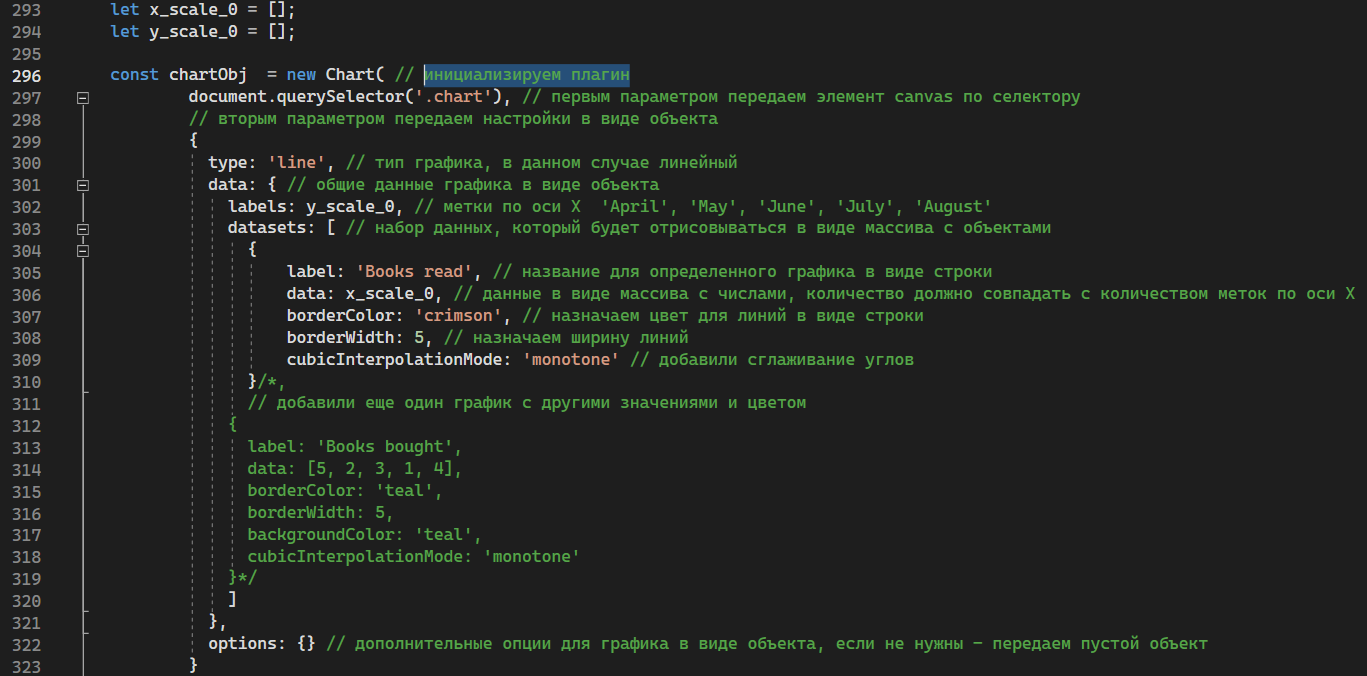
<div class="chartAreaWrapper">

<canvas class="chart"></canvas>

</div>

</div>

В js коде инициализируем плагин:



* Первым параметром передаем элемент canvas по селектору
* Вторым параметром передаем настройки в виде объекта
* type: 'line' тип графика, в данном случае линейный
* data общие данные графика в виде объекта
* labels: y\_scale\_0, // метки по оси X 'April', 'May', 'June', 'July', 'August'
* datasets: набор данных, который будет отрисовываться в виде массива с объектами
* label: 'Books read', название для определенного графика в виде строки
* data: x\_scale\_0, данные в виде массива с числами, количество должно совпадать с количеством меток по оси X
* borderColor: 'crimson’, назначаем цвет для линий в виде строки
* borderWidth: 5, назначаем ширину линий
* cubicInterpolationMode: 'monotone’ добавили сглаживание углов
* options: {} дополнительные опции для графика в виде объекта, если не нужны - передаем пустой объект

Делаем запрос:

var hostname = $(window.location).attr('hostname');

var url = hostname == "localhost»? "/api/pressure»: "/vera/api/pressure";

let response = await fetch (url, {

headers: {

'Accept': 'application/json',

'Content-Type':'application/json;charset=utf-8' }});

let json\_comit = await response.json(); // читаем ответ в формате JSON

Получаем замеры добавленные ранее и js коде реализовываем следующий метод ChartShow():

export function ChartShow(){

x\_scale\_0.splice (0, x\_scale\_0.length);

y\_scale\_0.splice (0, y\_scale\_0.length);

for(let x = 0; x < json\_comit.length; x++){

//графика

x\_scale\_0.push(json\_comit[x].sys);

y\_scale\_0.push(new Intl.DateTimeFormat("ru", {dateStyle: "short", timeStyle: "short"}).format(new Date(json\_comit[x].datetimebloodpressure)));}

chartObj.update();}

В массивах x\_scale, y\_scale, удаляются элементы из массива. В цикле for(), заносим в массив x\_scale\_0 значения свойства sys, а y\_scale\_0 добавляем дату внесенного значения и методом update() объекта Chart через ссылку chartObj, обновляем новыми данными графику.

**Отправка почты. SmtpClient**

Для работы с протоколом SMTP и отправки электронной почты в .NET предназначен класс SmtpClient из пространства имен System.Net.Mail.

Этот класс определяет ряд свойств, которые позволяют настроить отправку:

* Host: smtp-сервер, с которого производится отправление почты. Например, smtp.yandex.ru
* Port: порт, используемый smp-сервером. Если не указан, то по умолчанию используется 25 порт.
* Credentials: аутентификационные данные отправителя
* EnableSsl: указывает, будет ли использоваться протокол SSL при отправке

Еще одним ключевым классом, который используется при отправке, является MailMessage. Данный класс представляет собой отправляемое сообщение. Среди его свойств можно выделить следующие:

* Attachments: содержит все прикрепления к письму
* Body: непосредственно текст письма
* From: адрес отправителя. Представляет объект MailAddress
* To: адрес получателя. Также представляет объект MailAddress
* Subject: определяет тему письма
* IsBodyHtml: указывает, представляет ли письмо содержимое с кодом html

Используем эти классы и выполним отправку письма:

internal async Task SendEmailAsync(string? email, string? subject, string? message) {

// отправитель - устанавливаем адрес и отображаемое в письме имя

MailAddress? from = new("serega-a-02@yandex.ru", "Администрация сервиса дум.su/vera");

// кому отправляем

MailAddress? to = email != null ? new MailAddress(email) : new MailAddress("s\_antonov02@rambler.ru");

// создаем объект сообщения

using MailMessage m = new(from, to);

//К письму мы можем прикрепить вложения с помощью свойства Attachments. Каждое вложение представляет объект System.Net.Mail.Attachment:

//if (path != null) m.Attachments.Add(new Attachment(path));

m.BodyEncoding = Encoding.UTF8;

m.SubjectEncoding = Encoding.UTF8;

m.IsBodyHtml = true;

m.Subject = subject;

m.Body = message;

using (var client = new SmtpClient("smtp.yandex.ru", 587))

{

client.EnableSsl = true;

client.Credentials = new NetworkCredential("указываем почту", "пороль от почты");

await client.SendMailAsync(m);

}

m.Dispose();

email = null;

subject = null;

message = null;

//path = null;

from = null;

to = null}

Отправка на почту некоторых ссылок, так называемого токена с помощью **SmtpClient** объекта, применяется для двухфакторной аутентификации, который при подтверждениях на почте в виде запроса летит на сервер и проверяет, соответствует ли токен подтверждения по электронной почте указанному <имя параметра="пользователь"/> и если true, то в базе данных проставляет флаг. Пользователь, который авторизовывается в сервисе как раз код проверяет этот флаг.