Оглавление

[Введение 3](#_Toc57750062)

[Технологический стек 5](#_Toc57750063)

[Сущности 9](#_Toc57750064)

[Архитектура проекта 14](#_Toc57750065)

[Авторизация и аутентификация 20](#_Toc57750066)

[Механизм обмена сообщениями 21](#_Toc57750067)

[Описание работы приложения 24](#_Toc57750068)

[Заключение 33](#_Toc57750069)

[Список литературы 34](#_Toc57750070)

# Введение

Автоматизация рутинных задач и процессов – это одна из тех областей задач, которые решают с помощью информационных технологий. Таким образом с человека снижается нагрузка на тех областях, которые можно описать четкими инструкциями, которые повторяются многократно. А также решение на основе информационных технологий создает информационные системы, которые упрощают взаимодействие человека с этими самыми задачами, делает их простыми и удобными.

Сфера образования актуальна всегда, поэтому автоматизация даже части задач будет очень полезна. Упрощение взаимодействия и коммуникации между обучающимися и обучающими, упростить получение справочной информации и какой-либо отчетности сделает процесс в целом более комфортным.

В качестве сферы образования будем рассматривать среднее образование. Целью нашей работы будет разработка информационной системы, с помощью которой учителя, ученики, а также их родители, смогут взаимодействовать друг с другом, получать нужную информацию быстро и в удобном формате.

Определим задачи для реализации цели:

1. Четкое распределение ролей : ученик, учитель, родитель.
2. Возможность просматривать расписание, в зависимости от роли – разное расписание.
3. Возможность проставлять оценки у учителя.
4. Возможность просматривать оценки у родителя и ученика.
5. Возможность общения через чат между пользователями.
6. Администрирование системы.

Обозначим технические требования:

1. Система должна иметь стабильное хранилище данных.
2. Система должна обладать механизмами авторизации и аутентификации
3. Система должна быть достаточна гибкой и прозрачной к изменениям.
4. Разработка должна происходить на современном стеке технологий.
5. Система должна обладать механизмами общения пользователей в реальном времени с возможность хранения истории сообщений.

# Технологический стек

Для того, чтобы реализовать те задачи, которые мы определили ранее, необходимо выбрать те технологии, которые нам в этом помогут. Прежде всего следует определить тип разрабатываемой системы.

Наше информационная система будет основываться на базе веб-приложения. В пользу такого подхода говорит несколько аргументов:

1. Нет необходимости разрабатывать программное обеспечение под разные операционные системы.
2. Пользователю нет необходимости скачивать и устанавливать необходимое программное обеспечение, чтобы использовать нашу информационную систему. Пользователь лишь должен быть подключен к сети Интернет и иметь браузер.
3. Пользователь не ограничен в том, какой тип устройства использовать : полноценный компьютер на базе Windows или другой операционной системы, устройство на мобильное платформе iOS или Android, игровая приставка либо телевизор – необходимо лишь иметь веб-браузер и выход в сеть.
4. Большой выбор технологий для реализации проекта в рамках такого подхода.

Для того, чтобы реализовать проект в рамках веб-приложения, воспользуемся платформой .NET Core. .NET Core – это решение от компании Microsoft, которое позволяет писать программы независимо от платформы. Предыдущее решения, которое называлось .NET Framework, было ограничено рамками операционной системы Windows, поэтому оно было менее гибко в плане конечного развертывания решения. Современная же платформа позволяет публиковать решения и на других системах : Linux, MacOS.

В частности для реализации веб-приложения .NET Core предлагает решение, которое называется .NET Core AspNET MVC. Разработка веб-приложения подразумевает деление на две крупные части – backend и frontend разработку. Первая подразумевает те механизмы системы, которые скрыты от пользователя : операции с хранилищем данных и непосредственно с данными, вычисление каких-либо расчетов и описание бизнес-логики приложения. Вторая же часть отвечает за всю логику клиентской части, за то, как приложение выглядит для конченого пользователя, а также может производить несложные расчеты и вычисления, для которых нет необходимости обращаться к серверу, то есть к backend части.

Для разработки backend части используется упомянутое ранее решение .NET Core AspNET MVC вкупе с языком программирования C#. Данный язык программирования имеет очень широкое сообщество разработчиков, что позволяет быстро находить информацию о нем, а также решение нетривиальных проблем, возникающих в процессе разработки. C# - это объектно-ориентированный строго типизированный язык, поэтому порой для выполнения каких-либо небольших задач, приходится писать много структурного кода. Но это скорее плюс, чем недостаток потому, что грамотное структурирование кода на основе хорошей архитектуры позволяет содержать программный код в чистоте и с легкостью дописывать новый функционал и искать источники ошибок в текущем решении. Кроме того данный язык программирования содержит большое количество «синтаксического сахара». Под этим определением подразумевают синтаксические конструкции языка, которые упрощают его понимание и написание нового кода, делают код чище и прозрачнее. «Синтаксический сахар» инкапсулирует под собой некий другой код чаще всего на этом же языке программирования. Другими словами это некая надстройка над самим языком. Ради эксперимента можно попробовать декомпилировать решение на языке C# (например, Dynamic Link Library файл с расширением .dll) и, скорее всего, мы получим тот же код, но некоторые фрагменты будут описаны более примитивными конструкциями языка.

Для разработки frontend части будет использована технология той же платформы – Razor Pages. По большому счету эта технология очень похожа на язык гиперразметки текста – HTML, лишь с той разницей, что Razor Pages поддерживает использование C#. Даже исходные файлы этого решения имеют расширение .cshtml, что является сокращением от CSharp и HTML. Данный подход позволяет писать простой пользовательский интерфейс довольно быстро потому, что очень тесно интегрирован язык разработки серверной части C#, а для разработки и разворачивания не требуется дополнительных усилий, как при разработке через современные фреймворки вроде React и Vue – не нужно отдельно изучать и поднимать отдельное серверное решение клиентской части, которое называется NodeJS. Для того, чтобы стилизовать разметку, достаточно использовать классическую каскадную таблицу стилей CSS, а за программирование на клиентской части отвечает язык программирования JavaScript, а в частности библиотека JQuery. На данном этапе может возникнуть вопрос, зачем на клиентской части нужен язык программирования JavaScript, если мы уже используем C#. Дело в том, что в подходе Razor Pages используется так называется серверный рендеринг страниц. То есть по запросу пользователя на клиентской части (например, клик на гиперссылку) ему возвращается страница на HTML, которая уже содержит в себе все стили и скрипты на JavaScript. Но, чтобы сформировать этот HTML, используется смешение языка гиперразметки текста HTML и языка программирования C#. Мы можем генерировать разметку на основе какой-то модели, описанной классом на языке C#, можем генерировать какие-то однотипные теги с помощью циклов, решать нужна ли нам определенная разметка с помощью конструкций условия. Все эти решения мы реализуем с использованием C#. Конечное решение уже содержит в себе dll, в которым генерируется HTML на основе описанного нами кода в .cshtml файлах. К тому же такой подход позволяет нам выносить некоторые повторяющиеся элементы разметки в так называемые компоненты и переиспользовать без нужды в копировании кода.

В качестве хранилища данных будем использовать реляционную SQL базу данных, СУБД MSSQL. Базы данных позволяют строго структурировать данные и являются качественным хранилищем информации, которое зарекомендовало себя за долгие годы. Чтобы база данных хорошо сочеталась с серверным кодом используют ORM-технологию. Эта технология является некоторой прослойкой между SQL запросами к таблицам в базе данных и моделями и бизнес-логикой приложения, в нашем случае на C#. ORM автоматически транслирует данные с таблицы в объекты классов, а также генерирует запросы на языке SQL. Таким образом программисту достаточно продолжить использовать язык программирования, с которым он работал. В нашем случае ORM технологией будет EntityFramework.

# Сущности

Для того, чтобы начать разработку функциональной части приложения, необходимо описать предметную область, а если говорить подробнее, то сущности базы данных. Сущности в нашем проекте играют одну из ключевых ролей, так как они являются именно тем объектами, вокруг которых крутится все приложение. Мы в работе используем ORM технологию EntityFramework, поэтому можем воспользоваться подходом CodeFirst, с помощью которого мы можем абстрагироваться от базы данных и таблица, а описывать сущности, как классы на языке C#. Ответственность за работу непосредственно с таблицами берет на себя EntityFramework.

В первую очередь необходимо выделить пользователей системы. В нашем случае пользователями системы являются «Учитель», «Ученик», «Родитель». Для того, чтобы различать одного пользователя от другого на уровне ролей, заведем Enum UserRole, в котором выделим роли Student, Parent и Teacher. Однако Enum – это понятие из мира C#. Для базы данных это будет просто числовое значение. Значения перечисляются с нуля, поэтому Student = 0, Parent = 1, Teacher = 2. Так как мы говорим о пользователях, как о пользователях системы, то нужно отметить, что каждый из них имеет уникальное поле, которое идентифицирует его однозначно. Для этого заведем поля Id и Login. Первое будет иметь тип Guid, второе будет иметь строковое значение, для удобства восприятия пользователем потому, что Guid – это статистически уникальный 128-битный идентификатор вида { 6F9619FF-8B86-D011-B42D-00CF4FC964FF }. Таким образом, мы выделили сущность User с тремя полями : Id, Login, UserRole.

Однако описанные ранее роли могут иметь уникальные свойства на уровне ролей. Например, у учителя может быть указано место получения образования и опыт, а у других ролей свойства характерные только им. Мы могли бы наследоваться от сущности User и для каждой роли описать отдельный класс, что вылилось бы в разные таблицы, но так делать не стоит по нескольким причинам. Первая причина – это возможное расширение свойств для пользователей отдельных ролей в процессе добавления будущих изменений, что нарушает один из принципов SOLID.

SOLID – это акроним пяти принципов объектно-ориентированного проектирования, соблюдая которые можно достичь чистую и гибкую архитектура. Так вот, данное решение нарушает принцип «Открытости-Закрытости», которое гласит, что модули должны быть открыты для расширения и закрыты для изменения. Дело в том, что описав логику подобным образом, мы жестко завязываемся на определенные поля. Скажем место получение образования учителем – это поля типа String. В программном коде мы будем вынуждены обращаться к этому полю. Но что если завтра мы будем переименовать его? В таком случае мы будем вынуждены пройтись по всему коду и переименовать его везде. А что если мы будем вынуждены внести новый поля, характеризующие учителя? Придется пройти по коду, и везде вставлять эти поля. Тоже самое касается удаления.

Решим эту проблему следующим образом. Вынесем определение свойств и их значения в отдельную таблицу. Свойства вынесем в таблицу Properties, а значения в таблица PropertyValues. У обеих таблиц будут Id типа Guid. Property содержит название, а PropertyValues ссылку на свойство в виде PropertyId, а также ссылку на UserId. Таким образом мы можем безболезненно расширять свойства потому, как в исходном коде относительно пользователей это можно представить, как коллекцию данных, к которой можно обращаться по ключу. Чтобы решить проблему с переименование и, так называемым «хардкодом», вынесем строковые значения полей в конфигурационный файл. В такому случае, обращения к свойствам можно построить через поля класса, который описывает конфигурацию. Этот класс будет заполнять свои значения на основе конфигурационного файла, например, в формате JSON. Значит, что мы пишем конкретное значение названия поля один раз во всей программе и значит весь наш код по сути начинает зависеть от абстракции в виде полей класса конфигурации. Если нам нужно будет переименовать значения этого поля в БД, значит нам будет необходимо всего лишь обновить файл конфигурации один раз и не бегать по всему коду. Тут мы соблюдаем еще один из принципов SOLID – «Инверсия зависимости», который гласит, что «абстракции не должны зависеть от деталей, а детали должны зависеть от абстракции».

Перейдем к описанию более простых по своей структуре сущностей. Для описания класса учеников будем использовать сущность, которую назовем Class и которая будет иметь Id типа Guid, целочисленное поле номера класса Number и char поле Letter, которое будет отвечать за литеру. Для того, чтобы хранить сообщения в базе данных опишем соответствующую сущность Message, в которой будем хранить ее уникальный идентификатор Id типу Guid, а также поля, которые хранят информацию о том от кого и кому было отравлено сообщения с логинами пользователей From и To соответственно, полем даты и времени отправки Date типа DateTime, полем тела сообщения MessageBody типа string. Описание класса предмета Subject, который преподают, состоит только из названия предмете – строкового поля Name и уникального идентификатора Id типа Guid. Структура расписания предметов будет завязана на одном простом классе Schedule, который также состоит из уникального идентификатора Id типа Guid, даты Date типа DateTime, а также Guid полей сущностей предмета, учителя и класса – SubjectId, TeacherId, ClassId. Оценки, которые ставит учитель ученикам за работу на уроках и прочие заслуги и не очень, отдадим на откуп классу Score, который аналогично аналогам выше имеет уникальный идентификатор Id типа Guid, а также одноименное поле нового типа перечисления ScoreResult, которое отражает оценку, а также строковый комментарий Comment и Guid расписания и студента – ScheduleId, StudentId.

Для того, что связать учеников и родителей создадим связующую сущность ChildParentRelationship, которая будет иметь три поля Guid – Id, ChildId и ParentId : первое – уникальный идентификатор, второе и третье id объектов таблицы User. Подобным образом свяжем учеников и классы. В ответе за это будет класс ClassStudentRelationship, который также будет иметь три Guid поля – Id, ClassId, StudentId : первое – уникальный идентификатор, второе – id сущности класса Class, третье – id сущности класса User.

Как мы могли заметить, из класса в класс кочует поле уникального идентификатора, что не очень хорошо так, как приходится прибегать к копированию и вставке, что обычно является показателем плохого кода. Вынесем уникальный идентификатор Id типа Guid в отдельный абстрактный класс Entity, от которого будем наследовать все наши сущности.

Взаимосвязи и описание сущностей в виде диаграммы описано ниже на рисунке 1.

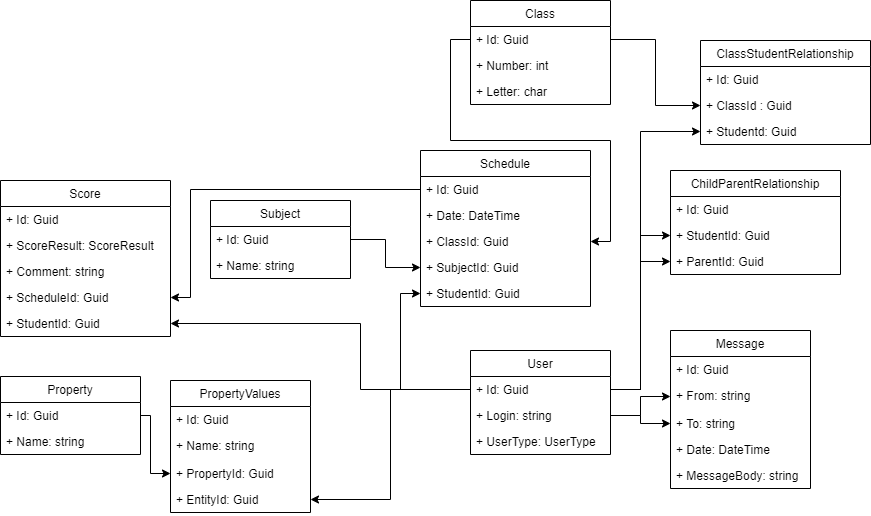


Рисунок Диаграмма сущностей

# Архитектура проекта

Непосредственно техническая сторона разработки проекта информационной системы начинается с архитектуры. От того, сколько времени и внимания мы заострим на этом аспекте непосредственно до написания реализации, зависит то, сколько усилий и ресурсов нам понадобится будущем на поддержание и развитие проекта. Все жесткосвязанные зависимости и непеременные элементы подвергаются изменениям, переплетаясь с серьезными трудозатратами.

Программировать следует на уровне абстракций, а не на уровне реализации. Поэтому построим взаимодействие внутреннего функционала нашего приложения на сервисах и интерфейсах. Внутренний функционал значит, что это те методы и механизмы, которые не видны конечному пользователю, но из них и их взаимодействия состоит тот функционал, который он использует.

Первое, что следует абстрагировать – это обращение к базе данных. Мы должны построить общение с источником данных таким образом, чтобы решение смены, скажем, SQL Server на PostgreSQL оказался для нас безболезненным. Обратимся к паттерна проектирования. Паттерны проектирования – это выработанный с годами опыта других программистов подход к написание кода. В программировании с большей долей вероятности с теми проблемами с которыми мы сталкиваемся, кто-то уже решил до нас, нужно лишь правильно найти ответ.

Используем паттерн «Репозиторий». В данный паттерн мы выносим основной перечень CRUD операций. CRUD – это акроним от операций создания Create, чтения Read, обновления Update и удаления Delete. Опишем абстрактный интерфейс на основе дженериков, таким образом, чтобы используемый тип, был наследником Entity.

public interface IRepository<T>

{

IEnumerable<T> GetAllItems();

T GetItem(Guid id);

void Create(T item);

void Update(T item);

void Delete(Guid id);

}

Реализацию данного интерфейса будем считать базовым репозиторием, с помощью которого можно работать с любой сущностью из базы данных. Удобство в том, что получился универсальный код. Нам не нужно писать взаимодействие с каждой сущностью отдельно. А при добавлении новой сущности, нужно будет всего лишь описать его класс и добавить в класс контекста. Обращение в сторонних сервисах будет доступно сразу.

Однако этого функционала может не хватить в некоторых случаях. Например, нам нужно запрашивать сущность пользователя не по id, а по логину. В таком случае мы должны создать новый интерфейс репозитория, который наследует интерфейс базового, также реализовать этот интерфейс, наследую базовый репозиторий. Также можно перекрывать базовую реализацию. В этом случае мы будем передавать не базовый репозиторий, а репозиторий конкретной сущности.

Уровень общения с данными мы определили. На данном этапе мы имеем сущности базы данных и технику взаимодействия с ними. Теперь мы должны обратиться к следующему слою – так называемой бизнес-логикой. По сути, бизнес-логика – это ядро нашего приложения, его движок. Все основные манипуляции с данными происходят именно здесь. Это промежуточный слой между «сырыми данными» и клиентом. Здесь мы описываем сервисы взаимодействия с сущностями и реализуем вспомогательный функционал. Тут следует отметить, что для работы с бизнес-логикой, следует использовать бизнес-модели (далее просто «модели»). Они отличаются от сущностей тем, что могу нести в себе дополнительную информацию, собранную из нескольких сущностей. Например, модель «Класса» будет содержать в себе не только литеру и номер, но и список всех учеников, которые будут представлять из себя коллекция моделей «Пользователь». Модель «Пользователь» в свою очередь имеет поле «Полное имя», которое в сущности не представлено, а вынесено таблицы Property и PropertyValues.

Межсервисное взаимодействие и взаимодействие с базой данных производится на основе интерфейсов сервисов, а не конкретно объектов сервисов. Мы не можем создавать экземпляры сервисов с помощью ключевого слова new потому, что это привнесёт жесткие связи. Чем же это чревато и почему это плохо? Это плохо тем, что если будут произведены изменения, то, скорее всего, мы будем вынуждены пробежаться по всему исходному коду, исправляя ошибки компиляции, где производится вызов, скажем, конструктора. Если же будет произведена полная замена реализации, то времени и сил будет потрачено в разы больше. Другое дело, если общение сервисов друг с другом будет построено на основе интерфейсов. Нам абсолютно нет никакого дело до того, как что-то работает, нам лишь важен контракт взаимодействия. Мы знаем, что мы передаем сервису и что он возвращает или не возвращает. Таким образом мы снова затрагиваем тему «Инверсии Зависимостей».

Для того, чтобы реализовать подобный подход, в .NET Core предусмотрен механизм, который называется «Инъекция зависимостей». Суть в том, что мы указываем в методе конфигурации проекта, какие интерфейсы мы помещаем в контейнер зависимостей и какие реализации им соответствуют. Этот механизм самостоятельно инициализирует необходимые сервисы, когда это необходимо. Вызов нужного сервиса осуществляется с помощью интерфейсов, прокинутых через конструкторы самих сервисов, важно лишь, чтобы этот сервис и все зависимости были определены в контейнере зависимостей.

На данном этапе мы определили, то как будет устроен основной функционал, общение с источником данных и взаимосвязи ключевых сущностей. Осталось лишь описать архитектуру взаимодействия с клиентом. На помощь нам придет проверенный временем подход разделения MVC – Model, View, Controller.

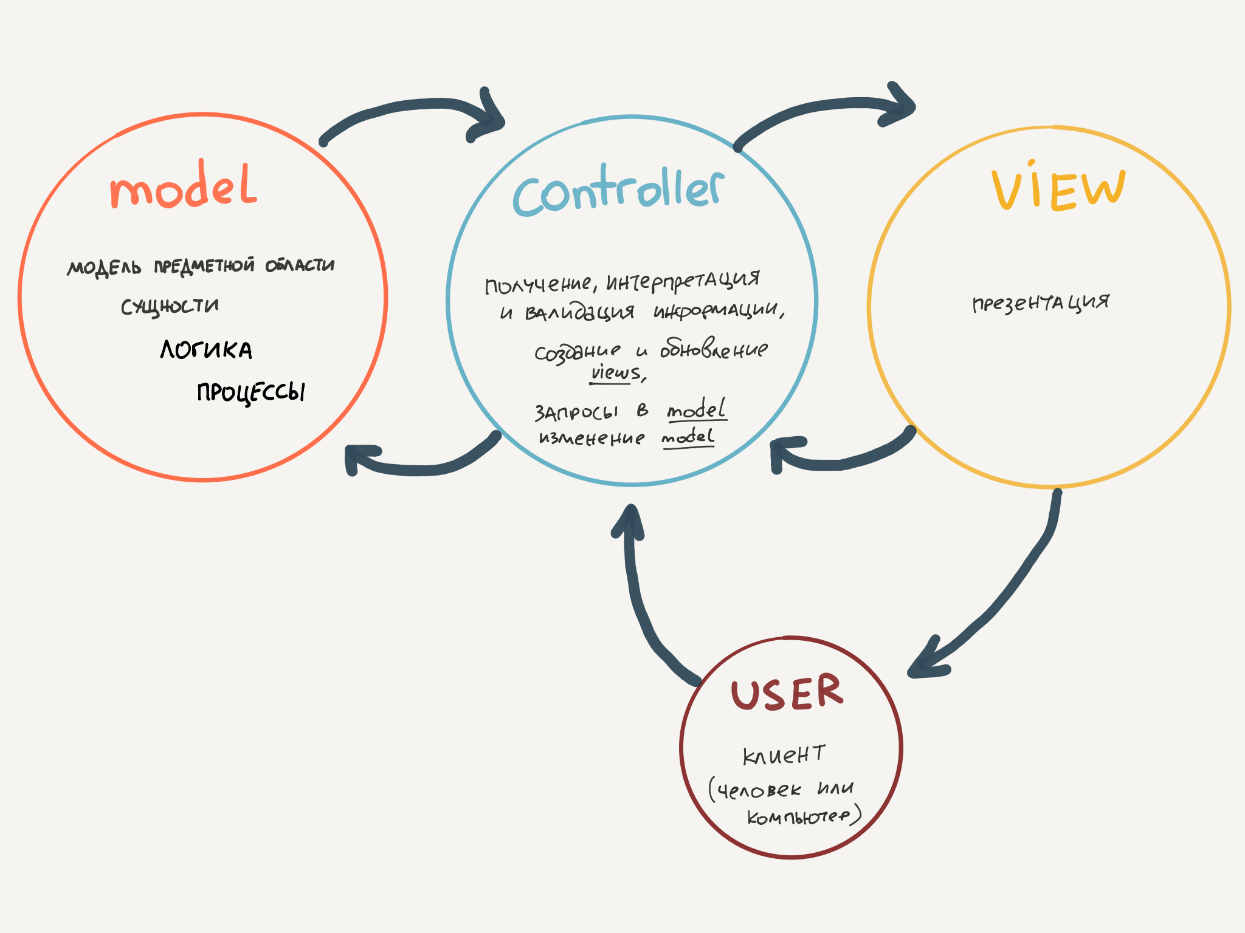


Рисунок Схематичное представление MVC

MVC – можно тоже назвать в некотором роде паттерном проектирования. MVC разделяет работу с пользователем на три слоя логики. Model – это непосредственная работа с данными, которую не видит конечный пользователь. В нашем случае это вся так архитектура, о которой мы говорили ранее : база данных и работа с источником данных, бизнес логика. Следующий слой – это Controller. На этом этапе мы обрабатываем запросы клиента и отдаем ему какой-то ответ. Это промежуточное звено, в котором мы решаем, например, какие методы из Model (бизнес-логики) вызвать, чтобы получить какие-то данные, либо совершить изменения над ними, и сообщить об этом клиенту. Чаще всего ответом является следующий слой – View. View, или представление, это непосредственно то, как пользователь видит нашу систему, чаще всего графический интерфейс.

Таким образом мы можем разбить работу с пользователем на три этапа : отобразить ему то, что он запросил, обработать его ответ и в случае, если ответ валиден, отправить его дальше для автоматической обработке уже непосредственно в ядре приложения.

Подытожим все, о чем мы говорили в этом разделе. Мы имеем четкую и гибкую архитектуру, в которой попытались предугадать возможные узкие места. Архитектура получилась отчасти абстрактной благодаря следования принципам SOLID. То, как данные циркулируют в нашем приложении, представлено ниже.

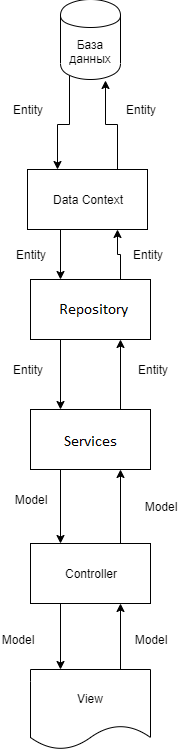


Рисунок Круговорот данных в приложении

# Авторизация и аутентификация

Мы определили три четкие роли в нашей системе «Учитель», «Ученик» и «Родитель». На самом деле есть еще одна роль, которая имеет самый широкий доступ к работе с данными в приложении «Администратор». Для того, чтобы реализовать этот функционал, мы должны четко понимать то, какой пользователь работает с системой в данный момент времени, какие права у него есть, что ему можно показывать, а что нет, что он может делать в системе, а что не может.

Для этого введем в нашу систему механизмы авторизации и аутентификации. Аутентификация – это проверка подлинности введенных пользователем данных для входа в систему. В нашем случае пользователь будет идентифицироваться по логину, так что для входа в систему ему нужен будет также пароль. Пользователей создает администратор системы, поэтому логины будут автогенерируемы путем инкрементирования значения постфикса логина. Префиксом же будет служить роль в системе : Student, Parent, Teacher. Авторизация – это процесс разграничения прав на определенные действия и ресурсы пользователя. Например, ученик не может ставить оценки, а учитель может.

В свете того факта, что мы работаем с .NET Core, нам предоставлен готовый механизм, который заключен в библиотеке решения Identity. Для работы с этим механизмом будет развернута дополнительная одноименная база данных, в которую будут включены необходимые таблицы работы с пользователями и ролями. Нам лишь необходимо связать пользователей базы данных приложения и базы данных Identity. Сделаем это на основе логина. Также следует понимать, что база данных Identity не содержит в себе паролей пользователей, она их хеширует инкапсулированной внутри себя логикой. Поэтому открытого доступа к этим данным злоумышленники получить не смогут, что добавляет нашей системе надежности с точки зрения безопасности.

# Механизм обмена сообщениями

Для удобства коммуникации пользователей между собой внедрим механизм обмена сообщениями. Обратимся к библиотеке SignalR, которая нам в этом поможет. SignalR использует двунаправленную связь для обмена сообщениями между клиентом и сервером, благодаря чему сервер может отправлять ответ в режиме реального времени.

Построим механизм обмена сообщениями по следующему алгоритму:

1. Пользователь вводит сообщение и нажимает «Отправить».

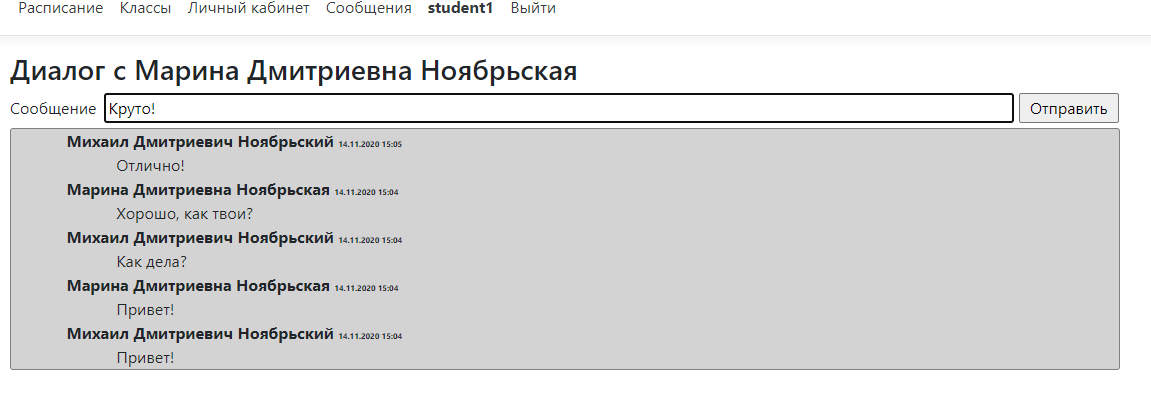


Рисунок Интерфейс обмена сообщениями

1. С помощью AJAX запроса на клиенткой части приложения отправляем запрос на специальную конечную точку, которая принимает в себе логин получателя и тело сообщения.

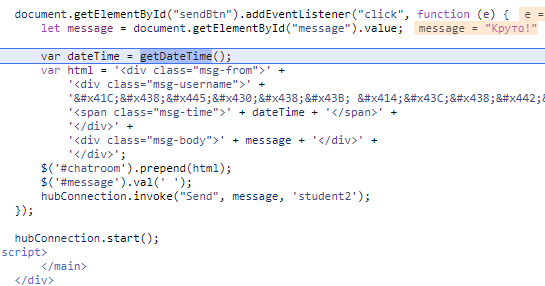


Рисунок Исходный код взаимодействия с SignalR на клиентской стороне – JavaSctipt

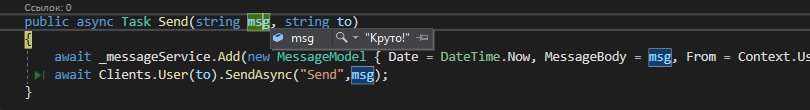


Рисунок Исходный код взаимодействия с SignalR на серверной стороне – C#

1. На сервере определяем точное время сообщения и вместе с переданными данными с клиента кладем данные о сообщении в базу данных.

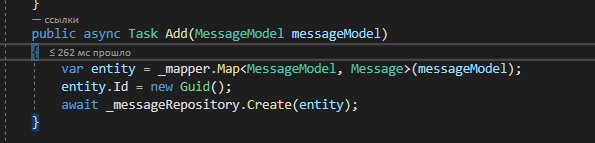


Рисунок Фиксирования сообщения в базе данных

1. Оповещаем собеседника нашего пользователя посредством SignalR. У собеседника на клиента должен быть также запущен код на языке JavaScript, чтобы «слушать» поток сообщения и при получении рисовать их в графическом интерфейсе.

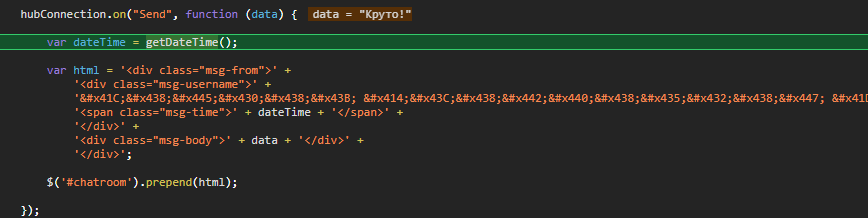


Рисунок Исходный код на клиентской стороне для принятия сообщения через SignalR

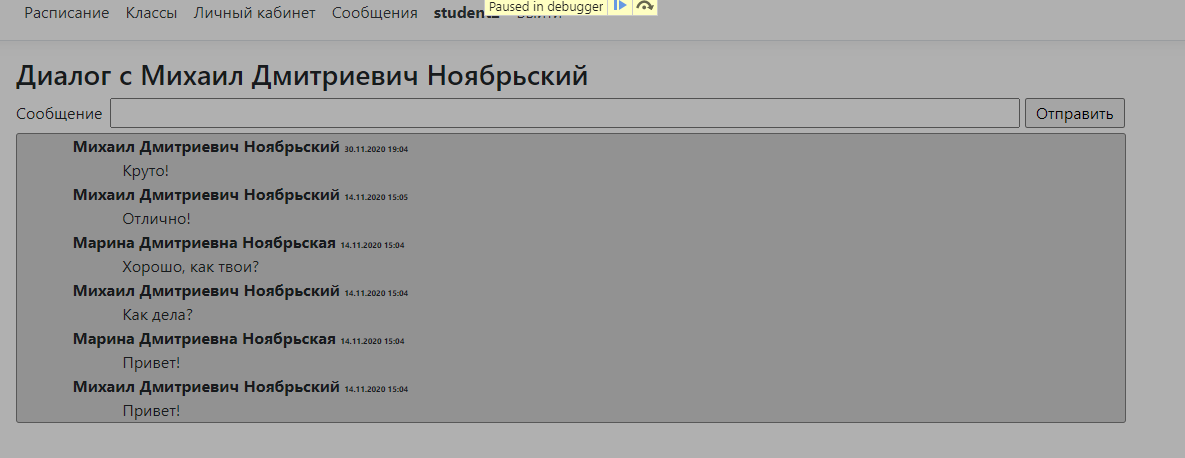


Рисунок Отрендеренносое полученное сообщение

# Описание работы приложения

Перейдем к описанию работы приложения. Первым делом неавторизованный пользователь попадает на страницу входа в систему.

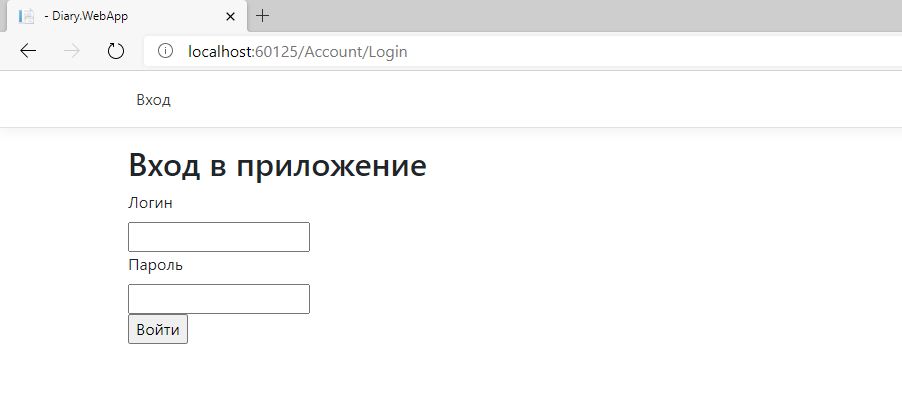


Рисунок Страница входа в приложение

Попадая на страницу входа, пользователь должен ввести пару логин/пароль. Регистрация любого пользователя невозможна, за это отвечает администратор системы.

Рассмотрим функционал, доступный ученику.

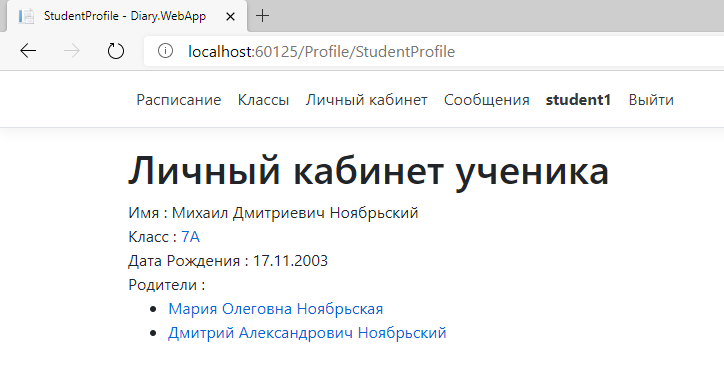


Рисунок Личный кабинет ученика

Как мы можем видеть на рисунке 11, в личном кабинете ученика видно его полное имя, его класс, с ссылкой на страницу класса, а также его дата рождения и ссылки на страницы его родителей.

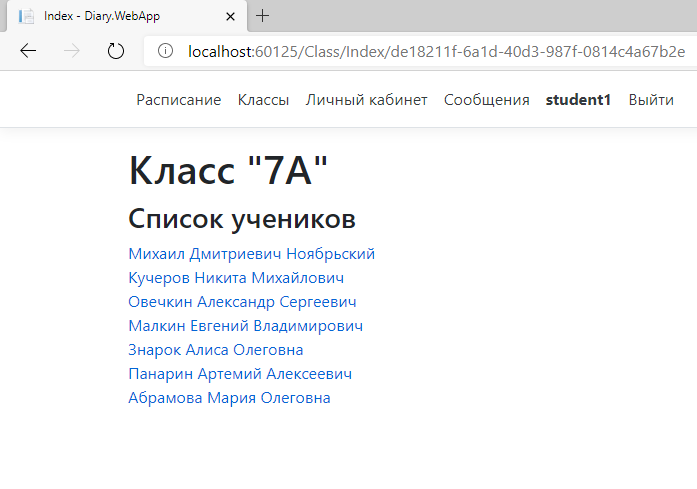


Рисунок Страница Класса со списком учеников

На рисунке 12 представлен список классов. Перейдя по ссылке, мы попадаем на страницу ученика, откуда можем узнать о нем информацию и написать сообщение.

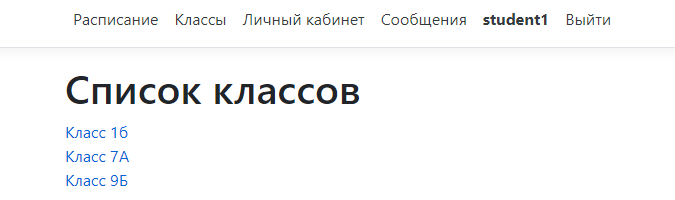


Рисунок Список классов

Перейдя по ссылке «Классы» в шапке системы, мы попадаем на список классов, через который можем попасть на страницу, представленную ранее.

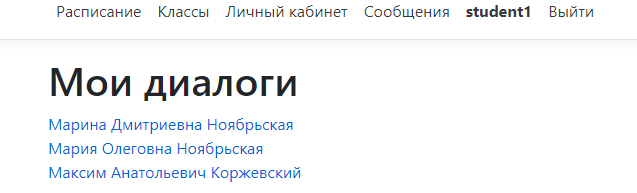


Рисунок Страница списка бесед

Перейдя по ссылке «Сообщения» в шапке системы, мы попадаем на список диалогов, через который можем попасть на страницу чата. Интерфейс механизма обмена сообщениями был представлен в предыдущем разделе.

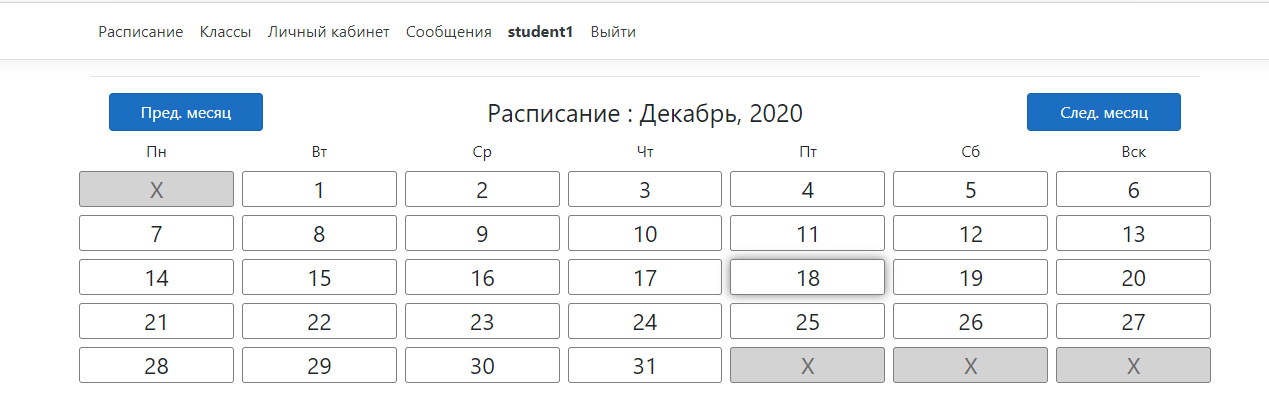


Рисунок Страница календаря расписания

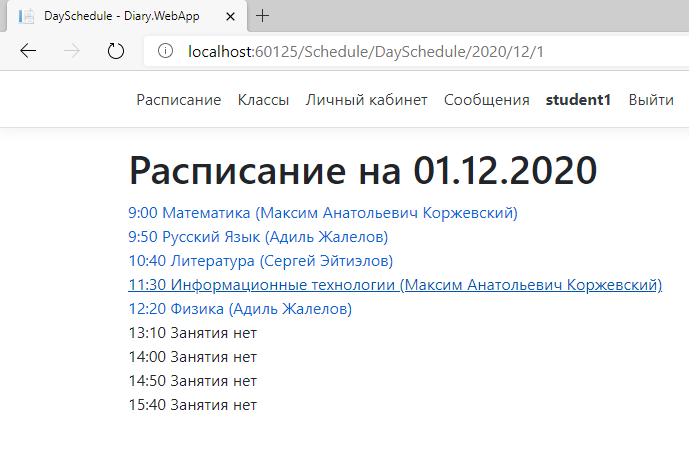


Рисунок Страница дня расписания

Попадая в раздел «Расписания», пользователь видит календарь, как на рисунке 15. Выбрав день, он попадает в конкретный день, где видит расписание. На рисунке 16 мы видим, что каждая ссылка на урок состоит из времени, названия предмета и учителя.

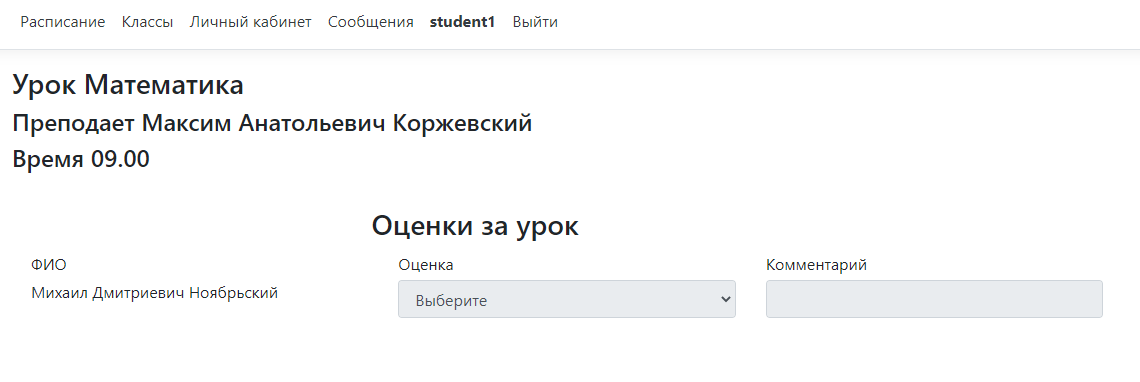


Рисунок Страница урока

На странице урока пользователи «Родитель» и «Ученик» могут видеть оценки и комментарий к ним, а пользователь роли «Учитель» может изменять их.

По большому счету, функционал пользователя «Родитель» повторяет функционал «Ученика». «Учитель» же видит свое расписание, а на его страницу выводится также место получения высшего образования и стаж.

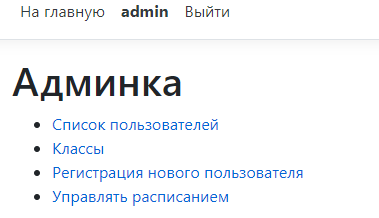
Отдельно нужно выделить администратора системы. 

Рисунок Главная страница администратора системы

Администратор может вносить новых пользователей и изменять текущих, создавать новые классы и изменять имеющиеся и управлять расписанием.

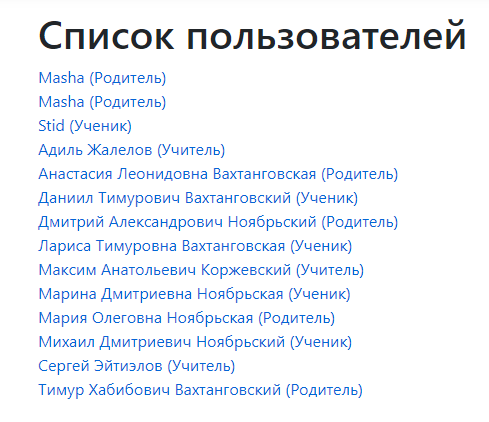


Рисунок Страница списка пользователей

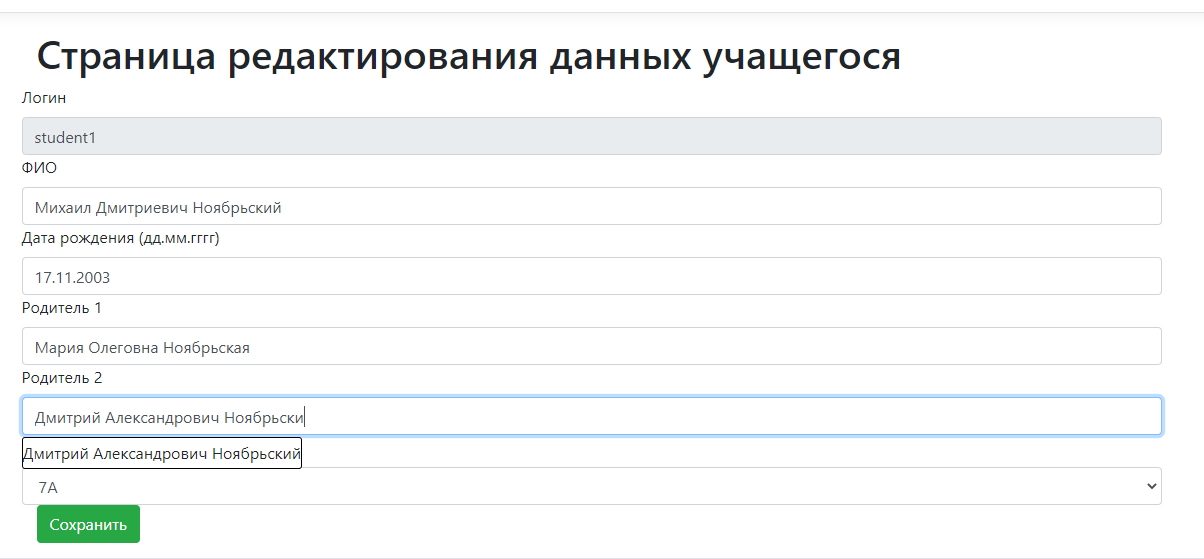


Рисунок Страница редактирования данных учащегося

Попадая на страницу редактирования списка пользователей, администратор мало того, что видит этот список (рисунок 19), так он еще и может переходить по ссылкам на них и редактировать с помощью форм (рисунок 20). Стоит обратить внимание, что некоторые поля, вроде поля «Родитель N» построены на механизме асинхронного запроса AJAX. Вводя какие-либо данные в строку текстового поля, начиная с некой длинны, данные отправляются на сервер, далее происходит поиск по данным из базы данных, где ключ поиска содержится во вхождениях. Далее результаты возвращаются на клиент и обрисовываются с помощью HTML и CSS в выпадающий список.

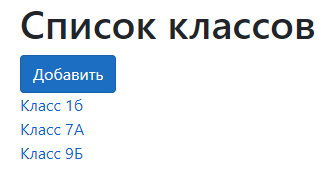


Рисунок Список классов

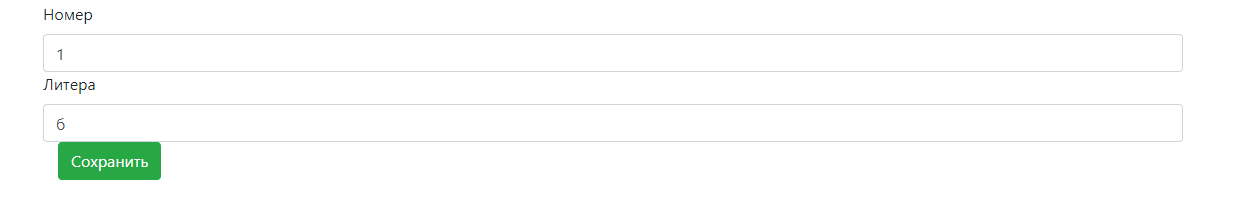


Рисунок Форма классов

На рисунках 21 и 22 представлены список классов и страница создания и редактирования.

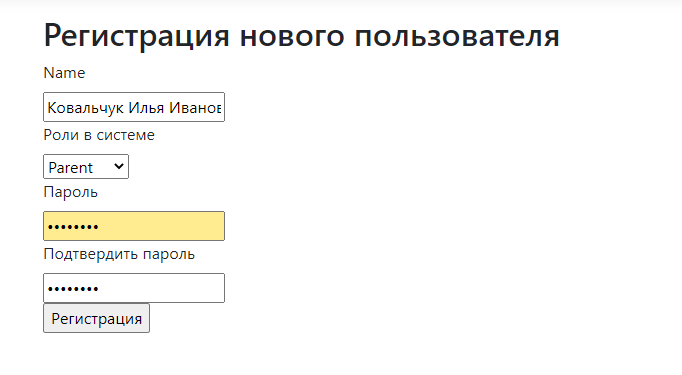


Рисунок Регистрация нового пользователя

Страница создания нового пользователя представлена на рисунке 23.

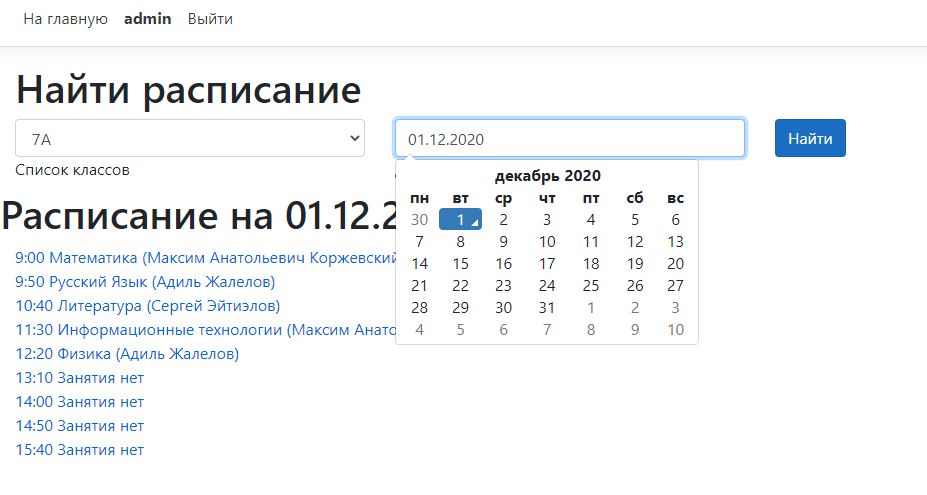


Рисунок Страница выбора дня расписания

На рисунке 24 видно, как происходит поиск нужного дня для редактирования расписания. Администратор вводит нужный класс и дату. Обратим внимание на то, что поиск даты выполнен с помощью механизма ввода даты. Это лишает пользователя возможности ввести дату в неверном формате и выглядит более эстетично.

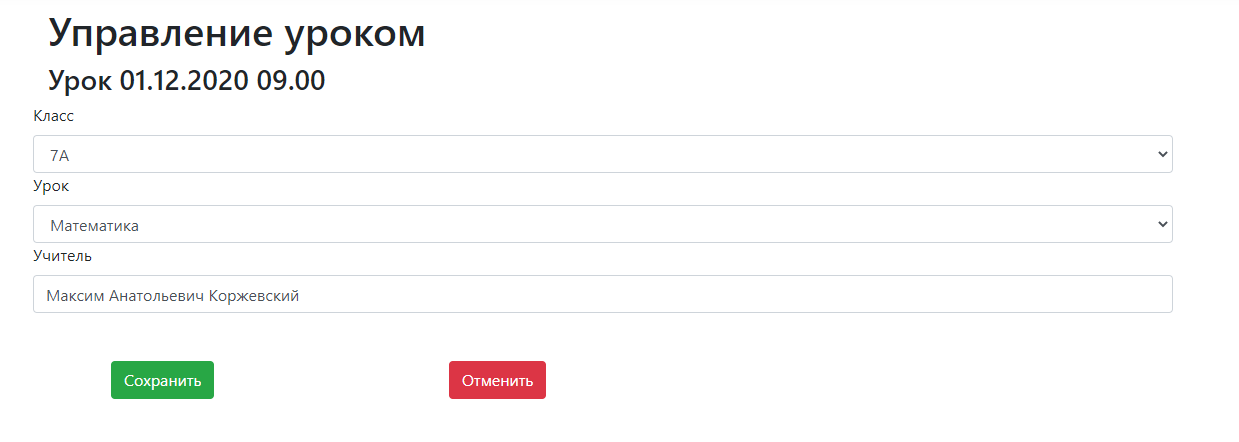


Рисунок Страница редактирования урока

На рисунке 25 мы можем увидеть непосредственно саму форму изменения расписания. Здесь есть выпадающий список для выбора нужного класса, а также асинхронный поиск учителя на подобии аналогичного поиска родителя на странице ученика.

# Заключение

Результатом данной работы является информационная система, которая автоматизирует рутинные вопросы учеников, родителей и учителей среднего звена. В результате разработки были выполнены все задачи и реализованы все технические требования, в итоге была достигнута поставленная цель. В ходе работы были получены необходимые навыки и знания для дальнейшего развития в этой области.

# Список литературы

1. Обзор Web-разработки (<https://sky-rzn.ru/blog/interesnoe/chto-takoe-web/>)
2. Создание архитектуры программы (<https://habr.com/ru/post/276593/?_ga=2.126598857.533583176.1589107565-932482008.1578951813>)
3. Схема разделения данных Model-View-Controller (<https://javarush.ru/quests/lectures/questcollections.level06.lecture01>)
4. Реляционная база данных (<https://aws.amazon.com/ru/relational-database/>)
5. Руководство по Entity Framework Core (<https://metanit.com/sharp/entityframeworkcore/>)
6. Введение в ASP.NET Core Identity (<https://metanit.com/sharp/aspnet5/16.1.php>)
7. ASP.NET Core MVC (<https://metanit.com/sharp/aspnet5/3.1.php>)
8. SOLID (<https://medium.com/webbdev/solid-4ffc018077da>)
9. Dependency Injection (https://metanit.com/sharp/aspnet5/6.1.php)