Институт Информационных технологий и компьютерных наук (ИТКН)

Кафедра Автоматизированных систем управления (АСУ)

             09.03.01 Информатика и вычислительная техника

**КУРСОВАЯ РАБОТА**

**по дисциплине «Разработка клиент-серверных приложений»**

Тема:

«Складской учет, поступление, отгрузка»

Руководитель: Рзазаде У. А.

Студент: Смирнов А.А.

Группа: БИВТ-20-1

Москва, 2022

**Оглавление**

[Введение 3](#_Toc105145242)

[1. Постановка задачи 4](#_Toc105145243)

[2. Описание архитектуры 5](#_Toc105145244)

[3. Описание структуры БД 6](#_Toc105145245)

[4. Описание серверной части 8](#_Toc105145246)

[5. Описание клиентской части 13](#_Toc105145247)

[Заключение 17](#_Toc105145248)

[Список литературы 18](#_Toc105145249)

# Введение

Сегодня в сети Интернет имеется большое количество электронных ресурсов. Пользователи приходят на них, чтобы получить или отправить какую-либо информацию, в результате чего возникает необходимость сбора, хранения и вывод нужной информации в веб-приложении.

Веб-приложение — клиент-серверное приложение, в котором клиент взаимодействует с веб-сервером при помощи браузера. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Главным преимуществом такого подхода является тот факт, что клиенты не зависят от конкретной операционной системы пользователя, поэтому веб-приложения в большинстве случаев являются кроссплатформенными службами.

Для осуществления общения клиента с сервером, необходимо современное программное обеспечение. Сегодня существует множество технологий (такие как PHP, JSP, ASP.NET, Spring), позволяющих создавать надёжные и эффективные web-приложения, отвечающие самым современным требованиям, удобные в использовании, как со стороны продавца, так и со стороны покупателя.

В данной работе будет рассмотрено многостраничное веб-приложение, контролирующее поступление и отгрузку товара, которое будет реализовывать работу склада одежды.

**Целью** данной **работы** является применение на практике знаний, полученных в процессе изучение дисциплины. Для этого выполняется разработка web-приложения по теме №13: Складской учет, поступление, отгрузка.

# Постановка задачи

В данной работе необходимо создать веб-приложение для оптимизации работы компании, предоставляющей склад для одежды и обуви. Эта организация обслуживает определенную базу клиентов и поставщиков. Клиентами считаются компании, которые арендуют место на складе для хранения своей продукции не далеко от своих отделений. Поставщиками считаются компании, которые производят и поставляют продукцию на склад для последующей продажи её клиентами. Организация, предоставляющая склад, считается посредником между клиентами и поставщиками.

Мое веб-приложение должно вести учет поставщиков, клиентов, поставок и контролировать изменение товаров на складе. Причем поставки могут быть двух видов: загрузка, когда товар поступает на склад, и отгрузка, когда товар забирают со склада. Для этого необходимо разработать базу данных, которая будет обеспечивать нужный функционал.

Далее для работы с этой базой данных нужно разработать HTTP-сервер, который будет реализовывать CRUD-операции для каждой созданной сущности, а также дополнительные варианты запросов от клиента.

В конце необходимо разработать клиентскую часть, которая включает в себя веб-интерфейс и программную часть. Через интерфейс будет производиться взаимодействие пользователей с самим приложением. Программная часть будет отвечать за правильное отображение http-страницы, а также отправку запросов серверу и получение ответов от него.

Задачи данной курсовой работы можно кратко описать следующим образом:

* проектирование и разработка архитектуры приложения;
* проектирование и разработка базы данных;
* проектирование и разработка серверной части приложения;
* проектирование и разработка клиентской части приложения;
* тестирование и устранение недостатков созданного приложения.

# Описание архитектуры

Для данной работы используется клиент-серверная архитектура, котораяпозволяет помещать прикладные программы на отдельный сервер, с которыми через API (Application Program Interface) устанавливается связь клиентов. Как можно увидеть на рисунке 2.1 все компоненты отделены друг от друга. Работа клиентской части приложения сводится к вызову необходимых функций сервера. Прикладные программы в свою очередь обращаются к базам данных с помощью SQL запросов. Их совместная работа происходит исключительно с помощью запросов и ответов. Такая архитектура позволяет отгородить клиента от лишних действий, так как основная работа выполняется на сервере.

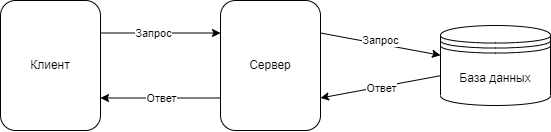


Рисунок 2.1 – Клиент-серверная архитектура

Работу серверной части в общем случае можно описать так: контроллер получает запрос от клиента и отправляет его сервису, который обрабатывает запрос и передает его репозиторию, который в свою очередь через контекст отправляет запрос в базу данных, после чего ответ от базы данных идет в обратном направлении и отправляется клиенту. Добавление сервиса и репозитория помогает разделить участки кода с разными обязанностями и тестировать отдельные участки кода.

Клиентская часть реализована с помощью веб-платформы Blazor, поэтому она состоит не только из пользовательского интерфейса. В ней также присутствуют модели данных, провайдер для обращения к серверу и некоторый код в html-страницах для настройки работы самого веб-интерфейса. Данная структура также позволяет разделять участки кода с разными обязанностями.

База данных представляет собой СУБД PostgreSQL, которая выполняет функцию хранилища данных.

# Описание структуры БД

Так как мое веб-приложение должно вести учет поставщиков, клиентов, поступления и отгрузки товаров, то в базе данных необходимо создать сущности: поставщики, клиенты, поставки и товары.

Главной сущностью оперируемой пользователем является поставка. В ней хранятся данные по отделу (division), типу (type), дате (date) и времени (time) поставки. Отдел хранит в себе краткое описание, понятное для работников склада, расположение товара на складе. Тип поставки говорит, будет ли товар загружаться на склад или отгружаться с него. Дата и время описывают то, когда будет производиться выгрузка или отгрузка продукции.

С помощью вторичного ключа сущность поставки связана с сущностями поставщик и клиент, в которых хранятся название компании (company\_name), текущий адрес (address) и телефон для связи (phone). В клиенте хранится еще и информация о ФИО переговорщика (negotiator), дате начала (date\_start) и конца (date\_end) сотрудничества. Так как предполагается, что и клиенты могут делать много заказов аренды с различными поставщиками, то связи между сущностями поставки (supplies) и клиенты (clients), а также поставки (supplies) и поставщики (suppliers) многие к одному.

Также поставка связана с товарами вторичным ключом, которые будут складироваться на складе или же наоборот доставаться со склада. В сущности товары находится информация о кратком названии (product\_name), бренде (brand), цене (price), размере (size), цвете (color) и количестве (number). Связь между сущность товары и сущностью поставка многие ко одному.

Для реализации хранения базы данных я выбрал СУБД (система управления базами данных) PostgreSQL — мощная система объектно-реляционных баз данных с открытым исходным кодом, активно разрабатываемая более 30 лет и заслужившая прочную репутацию за надежность, надежность функций и производительность [2]. В приложении pgAdmin 4, предоставляемое данной СУБД, я создал ERD-диаграмму (рисунок 3.1), на которой изобразил структуру своей БД.

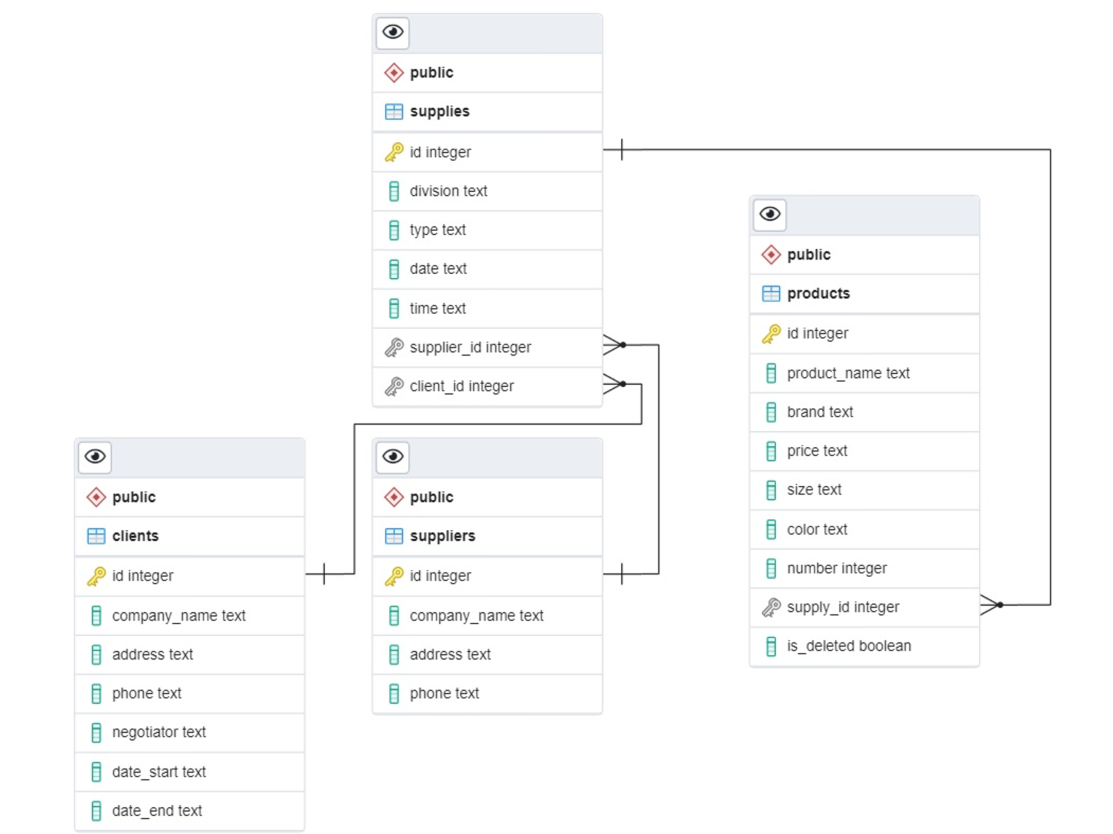


Рисунок 3.1 – Графическое представление структуры базы данных.

# Описание серверной части

Для реализации серверной части мною была выбрана платформа разработки веб-приложений ASP.NET, в состав которой входят: веб-сервисы, программная инфраструктура, модель программирования – от компании Майкрософт. Архитектура сервера построена на основе REST (Representational State Transfer) API – набор правил организации серверного приложения для удобного обмена данными с клиентом – разделена на несколько слоёв абстракции (рисунок 4.1): контроллер, сервис, репозиторий.

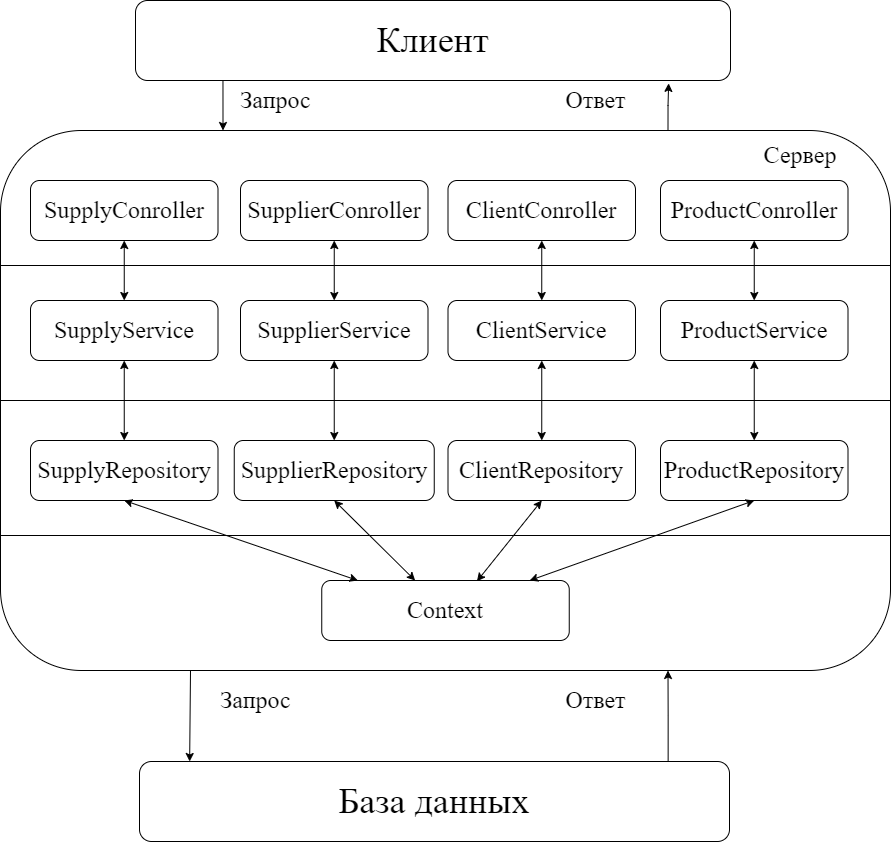


Рисунок 4.1 – Архитектура сервера.

Связь между уровнями устанавливается через внедрение зависимостей (dependency injection), которое в удобном виде реализовано в ASP.NET. В классе program через переменную builder, используемую для построения API (программного интерфейса приложения), я добавил контекст и установил получение нового сервиса и репозитория при каждом новом обращении (свойство AddTrasient). Также в builder подключаются контроллеры и сваггер для автодокументирования кода. Для правильной связи сервера и клиента я подключил Cross-Origin Resource Sharing (CORS) — механизм, использующий дополнительные HTTP-заголовки, чтобы дать возможность агенту пользователя получать разрешения на доступ к выбранным ресурсам с сервера на источнике (домене), отличном от того, что сайт использует в данный момент [3].

Для программной работы с базой данных я использую фреймворк Entity Framework core 6 (EF), который представляет собой удобную ORM-технологию для языка C#. Контекст базы данных — это основной класс, который координирует функциональные возможности EF для определенной модели данных [4]. По сути вся работа с базой данных ведется через этот контекст, а именно создание миграций и отправка запросов.

Также для работы с БД я создал модели данных: поставка, товар, клиент и поставщик – идентичные сущностям в базе данных. Для того, чтобы модели правильно воспринимались в C# существуют атрибуты. С помощью атрибутов я определяю столбец ключа (атрибут Key) и столбец сортировки (атрибут Column(Order)), указываю на поле, которое должно заполняться автоматически (атрибут DatabaseGenerated), а также явно преобразовываю типы данных (атрибут Column(Typename).

Пример модели:

public class Product

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column(Order = 1)]

public int Id { get; set; }

public string ProductName { get; set; }

public string Brand { get; set; }

public string Price { get; set; }

public string Size { get; set; }

public string Color { get; set; }

public int Number { get; set; }

[Column(TypeName = "boolean")]

public bool IsDeleted { get; set; }

public int SupplyId { get; set; }

public virtual Supply? Supply { get; set; }

}

Кроме моделей я создал объекты DTO (Data Transfer Object) –  один из шаблонов проектирования, используется для передачи данных между подсистемами приложения – для удобной и безопасной передачи данных от клиента к серверу и наоборот. Например, на одной из веб-страниц я вывожу список объектов DTO, который содержит информацию из нескольких таблиц. По своей структуре модели DTO не отличаются от основных моделей, за исключением отсутствия атрибутов.

Вся работа с базой данных на программном уровне ведется через репозиторий – это слой абстракции, включающий в себе всё, что относится к доступу к данным. Такой паттерн используется для разделения бизнес-логики от деталей реализации слоя доступа к данным. EF позволяет удобно работать с данными как с коллекциями, а язык запросов LINQ позволяет делать SQL запросы на языке C#, всё это сильно облегчает работу с БД и делает код понятнее. Такие операции как добавление, изменение и удалении в автоматическом режиме изменяют и смежные таблицы (EntityState.Modified).

Примеры методов из репозитория:

Метод получения поставки по идентификатору:

public async Task<Supply?> GetSupplyByIdAsync(int Id)

{

return await \_storageContext.Supplies.Include(supply => supply.Supplier.Include(supply => supply.Products)

.Include(supply => supply.Client)

.FirstOrDefaultAsync(supply => supply.Id == Id);

}

Метод изменения поставки:

public async Task<Supply?> UpdateSupplyAsync(Supply supply)

{

var result = \_storageContext.Update(supply);

\_storageContext.Entry(supply).State = EntityState.Modified;

await \_storageContext.SaveChangesAsync();

return await Task.FromResult(result.Entity);

}

Вся логика программы находится в сервисном слое – это шаблон проектирования, который отделяет бизнес логику приложения и определяет набор допустимых операций с точки зрения взаимодействующих с ним контроллеров. В основном в них я реализую преобразование данных полученных от клиента в модели, соединенные с базой данных, или наоборот. То есть при запросе на сохранение или изменение данных я из DTO или не полностью заполненной модели делаю готовый объект, который передается в репозиторий, а при запросе на предоставление данных я в сервисе получаю объект из БД и преобразовываю его в необходимый для клиента вид.

Как пример приведу метод для сохранения поставки отгрузки в сервисе SupplyService, в котором я получаю объект DTO и преобразую его в модель Supply:

public async Task<bool> AddSupplyOut(SupplyOUTDTO supply)

{

Supply lastSupply = await \_supplyRepository.GetSupplyByIdAsync(supply.SupplyINId);

if (lastSupply == null) { return false; }

Supply newSupply = new Supply()

{

Division = supply.Division,

Type = supply.Type,

Time = supply.Time,

Date = supply.Date,

Client = lastSupply.Client,

Supplier = lastSupply.Supplier,

Products = lastSupply.Products

};

if (newSupply.Products == null)

{

return true;

}

List<Product> products = await \_supplyRepository.GetProductsAsync(newSupply.Products.Select(pr => pr.Id).ToArray());

for (int i = 0; i < products.Count; i++)

{

products[i].IsDeleted = true;

await \_supplyRepository.ChangeExistProducts(products[i]);

}

return true;

}

Для общения клиента с сервером я использую контроллеры. Чтобы сервер понял к какому контроллеру обращается клиент в REST API используют маршруты (Route; URL к которому обращается клиент) и эндпоинты (Endpoint; обращение к маршрутам).

Пример контроллера:

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

[ApiController]

public class SupplyController : ControllerBase

{

private readonly SupplyService \_context;

public SupplyController(SupplyService context)

{

\_context = context;

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Supply>>> GetSupplys()

{

return await \_context.GetSupplies();

}

[HttpGet("{id}")]

public async Task<ActionResult<Supply>> GetSupply(int id)

{

var supply = await \_context.GetSupply(id);

if (supply == null)

{

return NotFound();

}

return supply;

}

[HttpGet("DTO/{id}")]

public async Task<ActionResult<List<SupplyDTO>>> GetAllSupplyDTO()

{

var supply = await \_context.GetSupplyDTO();

if (supply == null)

{

return NotFound();

}

return supply;

}

# Описание клиентской части

Клиентскую часть я решил реализовать с использованием фреймворка Blazor – веб-фреймворк, выпущенный Microsoft и позволяющий создавать браузерные приложения, используя, помимо HTML и CSS, язык C# и синтаксис Razor.

Для данного проекта точкой входя является класс program.cs, который инициализирует все базовые элементы (HttpClient, App и т.д.), зависимости для провайдеров и т.д.

Для работы с данными я создал классы моделей и DTO идентичные тем, что находятся в серверной части. Также я создал интерфейсы ко всем провайдерам для ограничения возможности для взаимодействия с сервером. Далее я создал классы провайдеров для каждой сущности: поставка, товар, поставщик, клиент.

Пример интерфейса провайдера:

public interface ISupplyProvider

{

Task<List<Supply>> GetAll();

Task<Supply> GetOne(int id);

Task<bool> Add(Supply supply);

Task<Supply> Edit(Supply supply);

Task<bool> Remove(int id);

}

Так как передача данных не может идти непосредственно через модели, то перед отправкой мне необходимо конвертировать объект модели в байт код. А перед получением данных нужно сериализовать данные из JSON-формата.

Пример метода получения данных в провайдере SupplyProvider:

public async Task<List<Supply>?> GetAll()

{

return await \_httpClient.GetFromJsonAsync<List<Supply>>("/api/Supply");

}

Пример метода отправки данных в провайдере SupplyProvider:

public async Task<bool> Add(Supply product)

{

string data = JsonConvert.SerializeObject(product);

StringContent httpContent = new StringContent(data, System.Text.Encoding.UTF8, "application/json");

var responce = await \_httpClient.PostAsync("/api/Supply", httpContent);

Console.WriteLine(await responce.Content.ReadAsStringAsync());

return await Task.FromResult(responce.IsSuccessStatusCode);

}

После создания основы получилась такая структура:

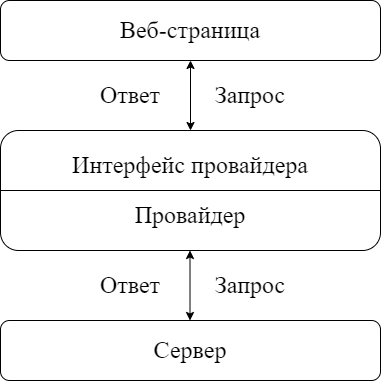


Рисунок 5.1 – Структура клиентской части.

Для построения страниц Blazor использует компоненты, которые хранят в себе части html-страницы и логику к ней. Из компонентов данный фреймворк собирает страницы, при чем этот сбор происходит таким образом:

* Подключение статических ресурсов из wwwroot;
* Подключение сторонних глобальных библиотек ко всем razor страницам в import.razor;
* Маршрутизация веб-страницы в App.razor;
* Формирование MainLayout.razor путем добавления шаблонов (Header, Footer) и заполнение @body начальной страницей (по стандарту Index.razor);
* Формирование Pages из компонентов

В своем веб-интерфейсе я сделал несколько страниц для вывода информации в таблицах и одну для добавления новой поставки. Начальная страница отображает все поставки, отсортированные по дате. Через них можно перейти к полной информации о поставке, а именно узнать какие товары были добавлены в неё. Так как Blazor предоставляет возможность изменять html-страницу в онлайн режиме, то я сделал сортировку товаров в поставке, а также выбор нужных по размеру и цвету. Так как данные функции работают примерно одинаково, то я опишу только выбор размеров товаров.

Пример работы checkbox:

<input type="checkbox" name="options" id="option1" @onchange="eventArgs => {CheckboxSizesClicked(size, eventArgs.Value); }" checked>

Я создаю список таких отмеченных (checked) checkbox с событием onchange, которое срабатывает при изменении, что в данном случае означает нажатие, и передает значение true/false через значение аргумента события (eventArgs.Value) в функцию CheckboxSizesClicked (size, eventArgs.Value), которая отвечает за изменение списка при изменении checkbox. Сама функция представлена ниже:

void CheckboxSizesClicked(string size, object checkedValue)

{

if ((bool)checkedValue)

{

if (!chosenSizes.Contains(size))

{

chosenSizes.Add(size);

}

}

else

{

if (chosenSizes.Contains(size))

{

chosenSizes.Remove(size);

}

}

products = CurrentSupply.Products.Where(pr => chosenSizes.Contains(pr.Size)).ToList();

}

Далее идет страница товаров, которая работает схожим со страницей с полной информацией о поставке образом. И в конце форма для добавления поставки. В форме есть отдельные поля самой поставки и поля для добавления продуктов в неё, а также таблица для вывода списка товаров. Таблица обновляется каждый раз при добавлении нового продукта. Поля для типа поставки, дате, времени и отделу заполняются выбором одного из предложенных элементов. Поля для поставщика и клиента заполняются также, но данные для них берутся гет-запросом из сервера, когда для остальных полей они хранятся на стороне клиента, так как эти данные не имеют большой ценности и имеют небольшой объем.

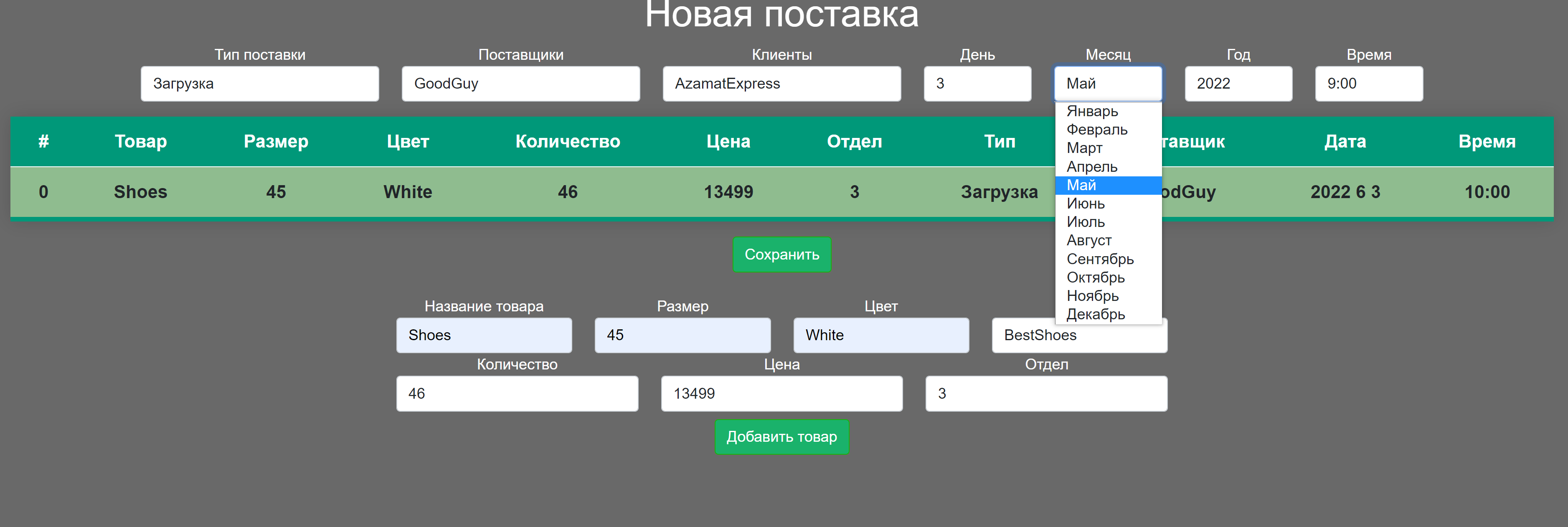


Рисунок 5.2 – Интерфейс формы добавления поставки.

Для работы с формой в Blazor я использовал привязки данных с событием onchange, которое срабатывает после изменения фокуса пользователя. При срабатывании события у элементов input для поставки у меня изменяется переменная отвечающая за этот элемент, а если у элементов input товара – элемент товара, который при нажатии на кнопку «Добавить товар» добавляется в таблицу. Пример привязки представлен ниже:

<input type="text" class="form-control" id="inputPassword4" placeholder="Размер" @bind-Value="@newProduct.Size" @bind-Value:event="onchange">

Кнопка «Сохранить» отвечает за пост запрос к серверу, который передает объект поставки для добавления в БД.

# Заключение

В процессе выполнения курсовой работы было реализовано клиент-серверное приложение по теме «Складской учет, поступление, отгрузка» с использованием веб-платформы Blazor и платформы разработки ASP.NET.

Данная работа помогла посмотреть на веб-разработку, как и со стороны front-end программиста, так и со стороны back-end. C# предоставляет много удобных инструментов для создания полноценных веб-инструментов. Особенно хочется отметить Blazor, который позволяет писать логику отображения html-страницы без использования языка Javacript. Также благодаря данной курсовой работе стала понятна архитектура веб-приложения.

Созданное приложение можно в дальнейшем доработать, добавив в него функцию изменения поставки, так как на практике часто бывают случаи, когда заказы меняются, особенно в небольших компаниях.

# Список литературы

1. Andrew Lock « ASP.NET Core in Action, Second Edition » – 2021 – 832 с;
2. Официальный сайт PostgreSQL. Режим доступа: – <https://www.postgresql.org/>, свободный. – (дата обращения 02:06:2022);
3. Cross-Origin Resource Sharing (CORS). Режим доступа: – <https://developer.mozilla.org/ru/docs/Web/HTTP/CORS>, свободный. – (дата обращения 02:06:2022);
4. Учебник. Начало работы с EF Core в веб-приложении MVC ASP.NET. Режим доступа: – <https://docs.microsoft.com/ru-ru/aspnet/core/data/ef-mvc/intro?view=aspnetcore-6.0>, свободный. – (дата обращения 15.05.2022);
5. Stack Overflow – Where Developers Learn, Share, & Build Careers. – Режим доступа: <https://stackoverflow.com/>, свободный. – (дата обращения: 26.05.2022);
6. ASP.NET Core Blazor. – Режим доступа: <https://docs.microsoft.com/ru-ru/aspnet/core/blazor/?view=aspnetcore-6.0>, свободный. – (дата обращения: 25.05.2022);
7. Ссылка на репозиторий приложения. – Режим доступа: <https://edu.gitflic.ru/project/alexsmirnov/bivt-20-1-smirnov-aa-13-asu/file> , приватный. – (дата обращения: 03.06.2022).