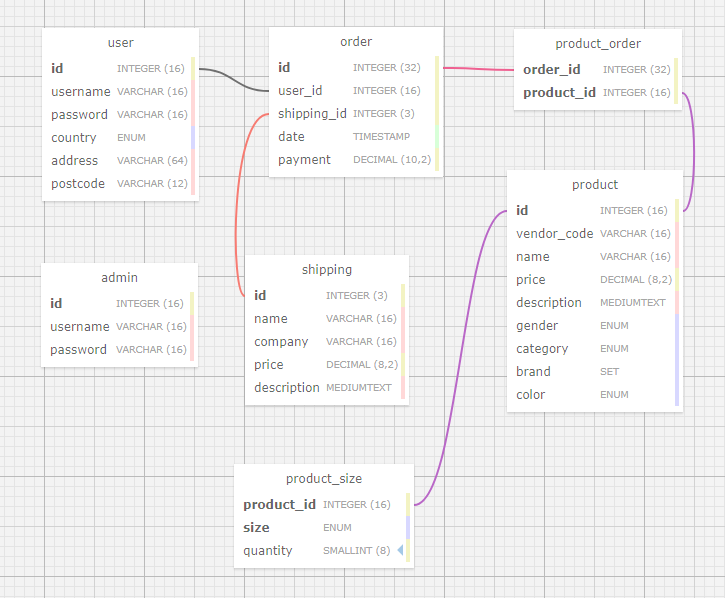
# 1. Создание структуры базы данных

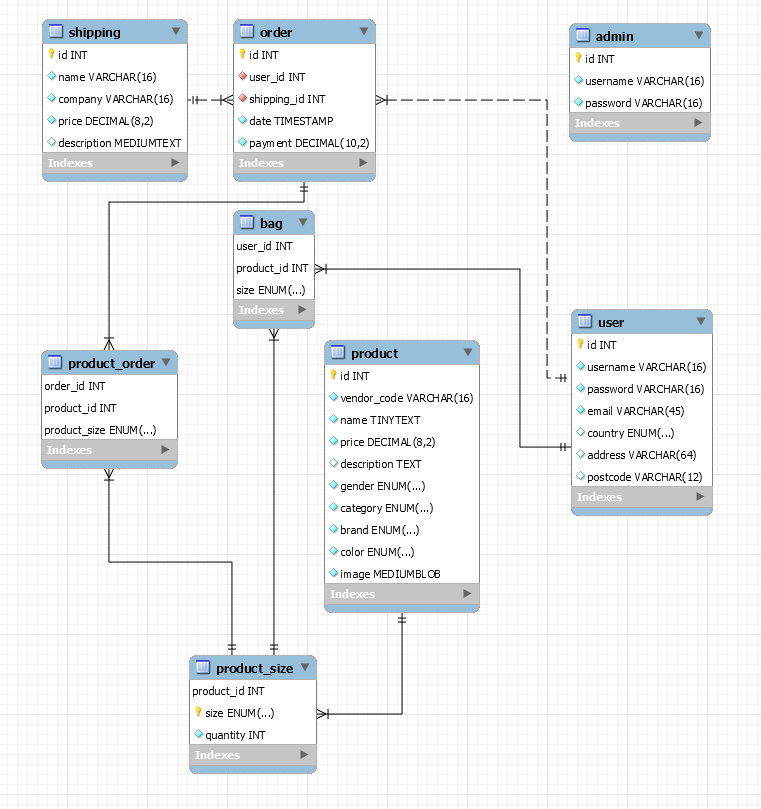
При первоначальном создании структуры базы данных нашего интернет-магазина была придумана следующая схема, созданная при помощи сервиса <https://ondras.zarovi.cz/sql/demo/>



где с помощью таблицы product\_order представлена связь многие-ко-многим между «товарами» и «заказами», а таблица product\_size отражает количество товара конкретного размера в наличии.

Затем с помощью того же сервиса был сгенерирован sql-файл для создания данной структуры, с помощью MySQL Workbench файл был немного скорректирован, и схема базы данных была создана.

Впоследствии при корректировке требований в схему были добавлены новые элементы: в таблице товаров появился столбец для хранения байт-кода фото данного товара, а также появилась таблица “bag”, которая отражает товары, находящиеся в пользовательской корзине. Конечная структура базы данных выглядит следующим образом (схема создана с помощью функции Reverse Engineering в MySQL Workbench):



# 2. Создание проекта Visual Studio

В качестве начальной точки был выбран проект ASP.NET Core Web Application по шаблону Model-View-Controller, так как с этим уже был небольшой опыт и, кроме того, по данной технологии есть очень доступное руководство на сайте <https://metanit.com/>.

Однако в этом руководстве описан подход code-first, при котором схема базы данных создаётся на основе сущностей, определённых классами-моделями, а так как база данных у нас уже есть, необходимо использовать подход database-first. И после недолгих поисков на официальном сайте документации Microsoft была найдена команда dotnet ef dbcontext scaffold, которая позволяет сделать в точности наоборот: на основе имеющейся базы данных создать код моделей и класса контекста базы данных, используемого для подключения.

Итак, был создан отдельный пустой проект библиотеки классов в уже созданном ранее решении, затем с помощью «Управления пакетами NuGet» в него были добавлены необходимые для «скаффолдинга» базы данных MySQL пакеты: Pomelo.EntityFrameworkCore.MySql и Microsoft.EntityFrameworkCore.Design

Затем, чтобы не открывать каждый раз консоль и не вводить команду снова при не обходимости, был создан файл scaffold.bat содержащий две команды:

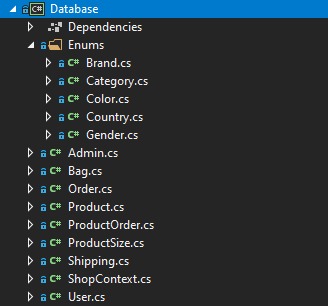
dotnet ef dbcontext scaffold "Server=localhost;Database=shop;User=root;Password=root;" "Pomelo.EntityFrameworkCore.MySql" -c ShopContext -f -d --no-build -p "Database.csproj"

pause

Итак, первый аргумент команды scaffold – строка подключения, второй – сервис, предоставляющий функционал подключения к нашему серверу, затем – название класса-контекста бд, несколько необязательных опций, которые время от времени изменялись, и в конце было указан путь к файлу проекта, куда помещались сгенерированные классы (этот файл и scaffold.bat находятся в одной папке, поэтому указано только имя).

Для упрощения реализации в этом же проекте была создана папка Enums, в которой были созданы специальные перечисления, соответствующие используемым непосредственно в базе данных типам Enum(…)

В результате получилось следующее:

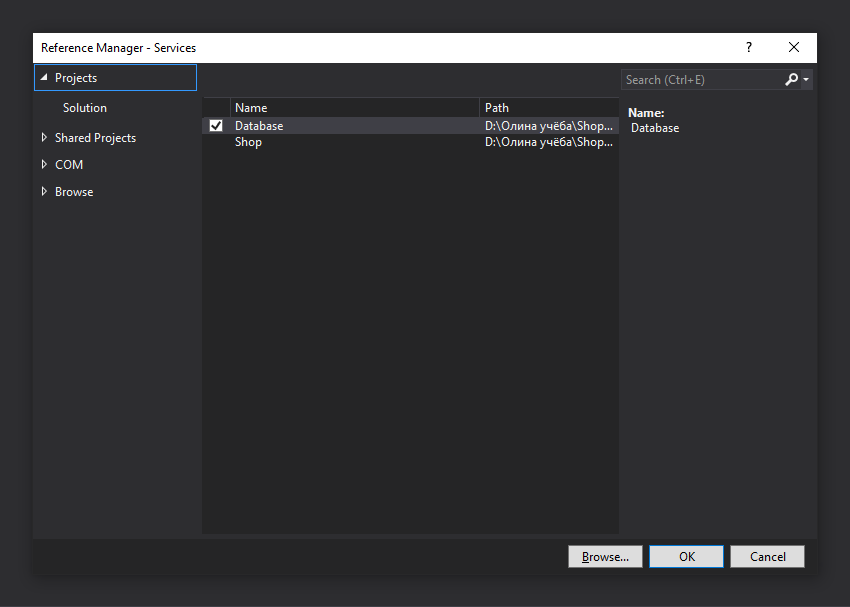


Кроме того, изначально было создано перечисление размеров товаров, но с ним впоследствии возникли непонятные проблемы при создании представлений, поэтому мы от него отказались и оставили везде изначально сгенерированный для полей-размеров тип string.

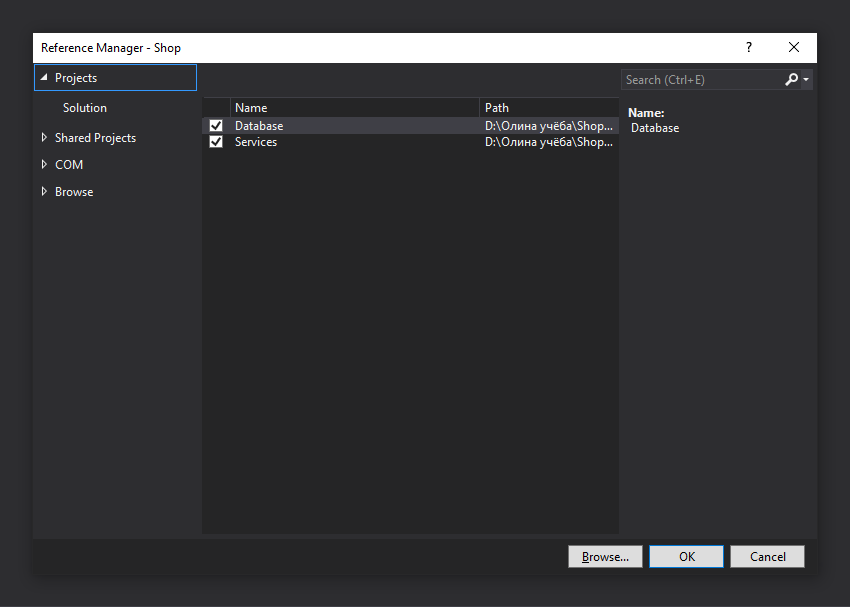
# 3. Определяем функционал (контроллеры и сервисы)

Следующим шагом было определение функционала приложения. При этом меня научили (почти), что правильный подход – это отделять функционал предметной области (бизнес-логику) от функционала, необходимого конкретному приложению (например, авторизацию). Поэтому функционал, заведующий каталогом товаров и заказами был вынесен в отдельный проект-библиотеку классов Services, а остальной остался в контроллерах.

Наверное, стоит сразу упомянуть о зависимостях между проектами. Services ссылается только на проект Database



В свою очередь, главный проект в решении Shop ссылается и на Database, и на Services



Без указания данных зависимостей мы не могли бы из одного проекта пользоваться функционалом другого.

Кстати о зависимостях, сервисы будут представлены интерфейсами, для чего нам понадобится внедрение зависимостей (dependency injection). Можно было бы воспользоваться встроенным функционалом ASP.NET Core для этого, но мне посоветовали воспользоваться IoC-контейнером (Inversion of Control – Инверсия управления) Autofac.

Для этого соответствующий NuGet пакет был добавлен в Services и в Shop, в класс Program в метод CreateHostBuilder была добавлена строчка .UseServiceProviderFactory(new AutofacServiceProviderFactory()), в проектах появились два класса-наследника Autofac.Module: AppModule и ServicesModule, а в классе Startup появился такой метод:

public void ConfigureContainer(ContainerBuilder builder)

{

builder.Register(dbContext => new ShopContext(Configuration

.GetConnectionString("Debug")));

builder.RegisterModule<AppModule>();

}

Первое, что он делает – регистрирует контекст базы данных, т.к. она используется во всём приложении (впрочем, это было бы вполне логично сделать и в AppModule, но эта идея пришла только сейчас). Затем – регистрирует модуль AppModule, то есть, грубо говоря, вызывает метод Load, который был переопределён в этом классе. В нашем случае там находится одна строка builder.RegisterModule<ServicesModule>();

В принципе, я думаю, можно было бы зарегистрировать ServicesModule сразу в ConfigureContainer, зачем нам лишнее звено? Но, как я поняла, AppModule нужен, когда нам нужно зарегистрировать сервисы, подобные логгированию. Впрочем, наверное, действительно правильнее было зарегистрировать контекст базы данных в AppModule.

Что находится в ServicesModule:

protected override void Load(ContainerBuilder builder)

{

builder.RegisterType<CatalogService>().As<ICatalog>();

builder.RegisterType<OrdersService>().As<IOrders>();

builder.RegisterType<ShopConnection>().As<IShopConnection>().SingleInstance();

}

Далее рассмотрим подробнее.

Сейчас я уже не могу вспомнить, что и в каком порядке создавалось, но конечный результат следующий.

##### BaseDbController

Начнём с базового абстрактного класса контроллера, использующего базу данных, BaseDbController, наследуемый от Microsoft.AspNetCore.Mvc.Controller, от которого впоследствии были унаследованы остальные контроллеры, т.к. все контроллеры в данном проекте используют БД, и сервиса IShopConnection, который использует этот контроллер.

Что вообще такое контроллер? Это класс, который содержит набор функций, называемых действиями. Когда пользователь обращается к серверу, выполняется http-запрос, в запросе есть URL, и в этот URL обычно соответствует такому шаблону: «http(s)://{доменное имя или localhost:XXXXX}/{имя контроллера}/{имя действия}}. Так среда и понимает, к чему обращается пользователь и вызывает определённое действие. Это действие принимает контекст запроса, что-то делает и возвращает ответ. Ответом может быть просто статусный код, некоторый текст, некоторый объект, перенаправление на другое действие или же представление (о них позже). Можно так же задавать свои маршруты через атрибуты (тоже позже)

В интерфейсе сервиса IShopConnection определены три функции:

* void BeginTransaction();
* void CommitTransaction();
* void RollbackTransaction();

А также свойство, возвращающее экземпляр класса контекста базы данных.

В контроллере были переопределены два метода базового класса Microsoft.AspNetCore.Mvc.Controller, первый из которых вызывается перед выполнением любого действия в контроллере, а второй - после. Первый открывает транзакцию в базе данных (вызывает BeginTransaction();), второй проверяет наличие ошибок в выполнении действия, и если ошибка имеется, то транзакция откатывается (вызывается RollbackTransaction();) и в ответ на http запрос записывается статусный код 500, обозначающий ошибку на сервере. В случае успешного выполнения действия, транзакция записывается (вызывается CommitTransaction();).

Класс ShopConnection, который был зарегистрирован, как SingleInstance() (время жизни зависимостей я пока не очень хорошо уяснила, но у этого суть в том, что при первом обращении к IShopConnection создаётся экземпляр его реализации, а при следующих просто передаётся он же), реализация данного интерфейса проста – просто вызываются соответствующие методы из Microsoft.EntityFrameworkCore, а также дополнительно определены приватные методы void OpenConnection() и void CloseConnection(), которые также вызывают соответствующие методы из Microsoft.EntityFrameworkCore, а сами при необходимости вызываются соответственно в BeginTransaction() и в CommitTransaction() или RollbackTransaction().

Сервисы же каталога и заказов были просто зарегистрированы, как рабочие реализации своих интерфейсов, насколько я поняла, здесь нам было не принципиально их время жизни. Кстати, все зависимости передаются через конструкторы.

##### CatalogController

Теперь перейдём к CatalogController. Здесь я впервые познакомилась с атрибутами. С помощью них я задаю:

* Маршруты, атрибут [Route(string template)] (кстати, чтобы он корректно работал, в класс Startup необходимо добавить app.UseRouting(); и app.UseEndpoints(endpoints => endpoints.MapControllers());)
* HTTP метод, я использую в основном [HttpGet] и [HttpPost], т.к. для отправки запросов используются формы в представлениях и ссылки, другие методы, как я выяснила, используются в RESTful подходе с использованием JavaScript. Соответственно, в контроллере может быть два действия с одинаковым названием и маршрутом, но среда поймёт, какой именно нужен, если в запрос передан тем или иным методом.
* Необходима ли авторизация, атрибуты [Authorize] и [AllowAnonimus]. Второй используется, если первый был применён сразу ко всему контроллеру, иначе его указывать необязательно, он действует по-умолчанию. В первый так же можно передавать нужную роль пользователя. (да, это app.UseAuthentication(); и app.UseAuthorization(); в Startup)

Итак, какие действия определены в CatalogController:

* Index, принимает фильтры и возвращает представление (о них позже), в которое передаётся список товаров, удовлетворяющих данному фильтру. В случае отсутствия фильтров, очевидно, передаётся весь список товаров.
* Product, принимает id товара, и возвращает представление, которое и показывает заданный товар пользователю.
* AddItem, их два, [HttpGet] и [HttpPost]. Первое просто возвращает форму добавления товара. Данные этой формы потом направляются к post-части этого действия, и оно уже добавляет товар в базу данных, затем перенаправляет на свою get-составляющую. К этому действию есть доступ только у админов.
* Два похожих действия, оба [HttpPost], AddToBag и RemoveFromBag. Оба принимают id товара и размер. Первое, очевидно, добавляет указанный товар указанного размера в корзину, другое – удаляет. Первый потом перенаправляет на Index, второй – на действие GetBag из UsersController, хотя вообще его было бы целесообразно поместить тут же. Доступ есть только у пользователей.

##### UsersController

Раз уж последнее действие перенаправляет на UsersController, то перейдём к нему.

В качестве контекста базы данных я задаю ему тот же контекст, который хранится в IShopConnection, чтобы у нас транзакция и изменения происходили в одном контексте.

За что отвечает контроллер?

Во-первых, конечно же за аутентификацию. Там содержатся такие действия, как регистрация, вход и выход. Причём для первых двух определены и get, и post методы. Get возвращают форму, post выполняют запись в бд в случае регистрации и поиск в бд в случае входа, и саму аутентификацию.

Аутентификация производится с помощью cookie и claims, которые и хранят данные пользователя, включая роль. Чтобы использовать аутентификацию и авторизацию в приложении, класс Startup должен содержать следующее:

services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)

.AddCookie(options => //CookieAuthenticationOptions

{

. . .

});

в методе ConfigureServices и

app.UseAuthentication();

app.UseAuthorization();

в Configure

Также в контроллере есть действие по изменению данных доставки. На него пользователь перенаправляется после регистрации. Это вынесено в отдельное действие, чтобы пользователь мог пропустить этот шаг и сделать это позже.

Ну и, наконец, get-запрос на корзину. Просто возвращается представление, в котором отображены товары в корзине пользователя. Id пользователя как раз получается из claims.

##### AdminsController

Раз поговорили об обычных пользователях, поговорим о другой их касте – администраторах.

Их контроллер имеет следующие действия:

* Index, возвращает главную админскую страницу, где перечислены все администраторы. Через неё можно добавить нового админа или удалить кого-то из старых.
* Login, get, как обычно, возвращает форму, post выполняет вход. Здесь нет регистрации, предполагается, что добавить администратора может только другой администратор.
* Logout, конечно же.
* И методы добавления и удаления администраторов. При добавлении генерируется случайный пароль, который после добавления выводится единожды в Index`е. Такая вот странная мера безопасности.

##### OrdersController

И последний контроллер, отвечающий за заказы:

* My, принимает id заказа и отображает его детали. В случае, если этот параметр не был передан (пользователь перешёл просто на страницу /order/my), отображает краткую информацию обо всех сделанных пользователем заказах.
* Два администраторских действия: All и ChangeOrderStatus. Первое отображает все заказы, второе позволяет изменить статус заказа.
* ShippingSelection, выбор способа доставки. Сюда в первую очередь идёт пользователь, который хочет сделать заказ. Выбрав, он вызывает следующее действие.
* Index, собственно, выполнение заказа, добавление его в базу данных. Принимает id способа доставки, берёт id пользователя из claims, проверяет у него наличие данных доставки (в случае отсутствия перенаправляет на соответствующее действие контроллера пользователей), и делает заказ, собирая товары из корзины пользователя и добавляет в базу заказ с нужными товарами и выбранным способом доставки.

# Представления (Views)

Каждое представление – .cshtml файл – это файл html-страниц, в которые можно вставить код на c#. Контроллер вызывает представлении с помощью функции View(). Нужное представление среда ищет по шаблону: Views/{Имя контроллера}/{Имя действия}.cshtml. Так же можно указать, какое представление нужно, передав путь в качестве аргумента (так делается, например в действиях My и All контроллера заказов).

Также в представления могут передаваться некоторые модели, которые могут обрабатываться внутри представления, например, для отображения конкретных значений. Можно передавать различные списки, чтобы формировать таблицы с переменным количеством строк/столбцов посредством цикла foreach.

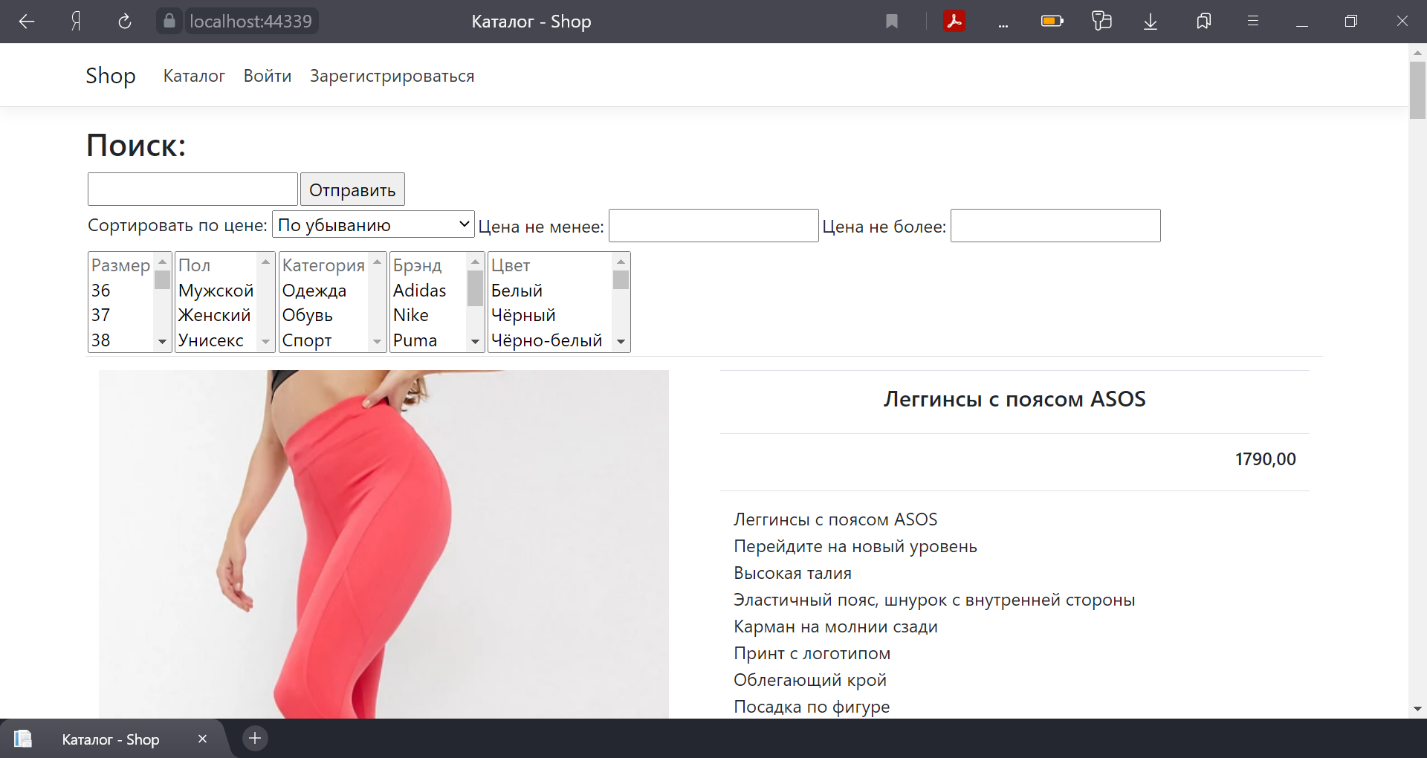
Также чтобы передавать данные в представление можно пользоваться переменной-словарём ViewData. В BaseDbController я передаю туда значения “admin” и “user”, которые показывают, авторизован ли пользователь и под какой ролью, и используются для отображения правильных ссылок в навигационной панели. Так же если пользователь не авторизован, ему недоступна кнопка «добавить в корзину» на странице товара. Хотя, наверное, тут правильнее было бы просто перенаправлять его на страницу входа.

По-умолчанию в проекте была задействована мастер-страница, \_Layout.cshtml. Она находится в папке Shared. Это общая для всех других представлений страница, на которой находятся навигационная панель и футер (у меня он пустой). А контент конкретной страницы добавляется с помощью метода RenderBody(). Есть файл \_ViewStart.cshtml, который применяется к каждой странице в папке Views и указывает, что в качестве мастер-страницы используется именно \_Layout.cshtml, то есть в принципе не обязательно иметь одну мастер-страницу на всех, они могут быть разными и задаваться в начале каждого представления через переменную Layout. Также в начале каждого представления задаётся переменная ViewData["Title"], значение которой помещается под тэг <title> в мастер-странице.

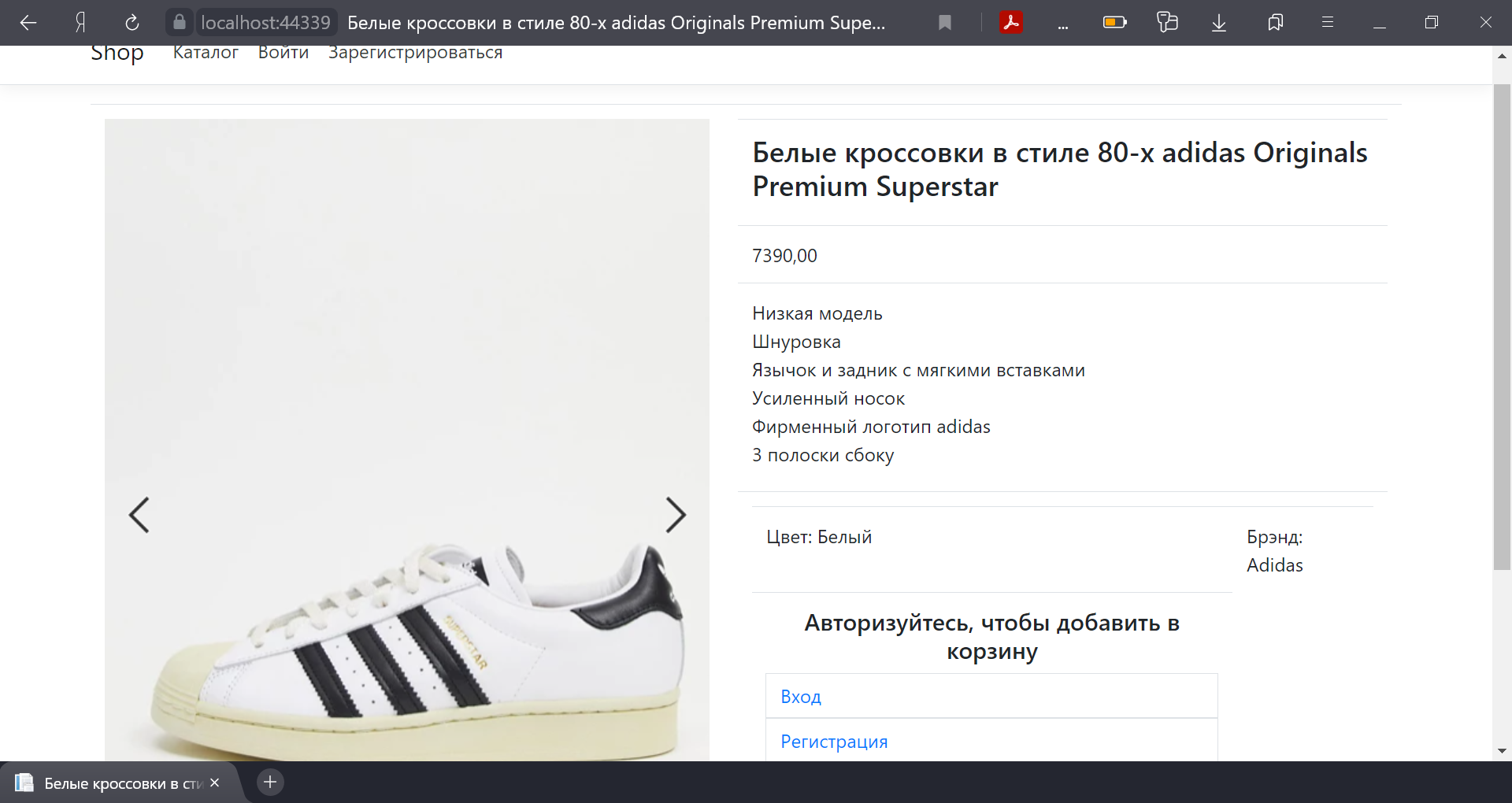
# Результат

Что же в конечном итоге получилось?

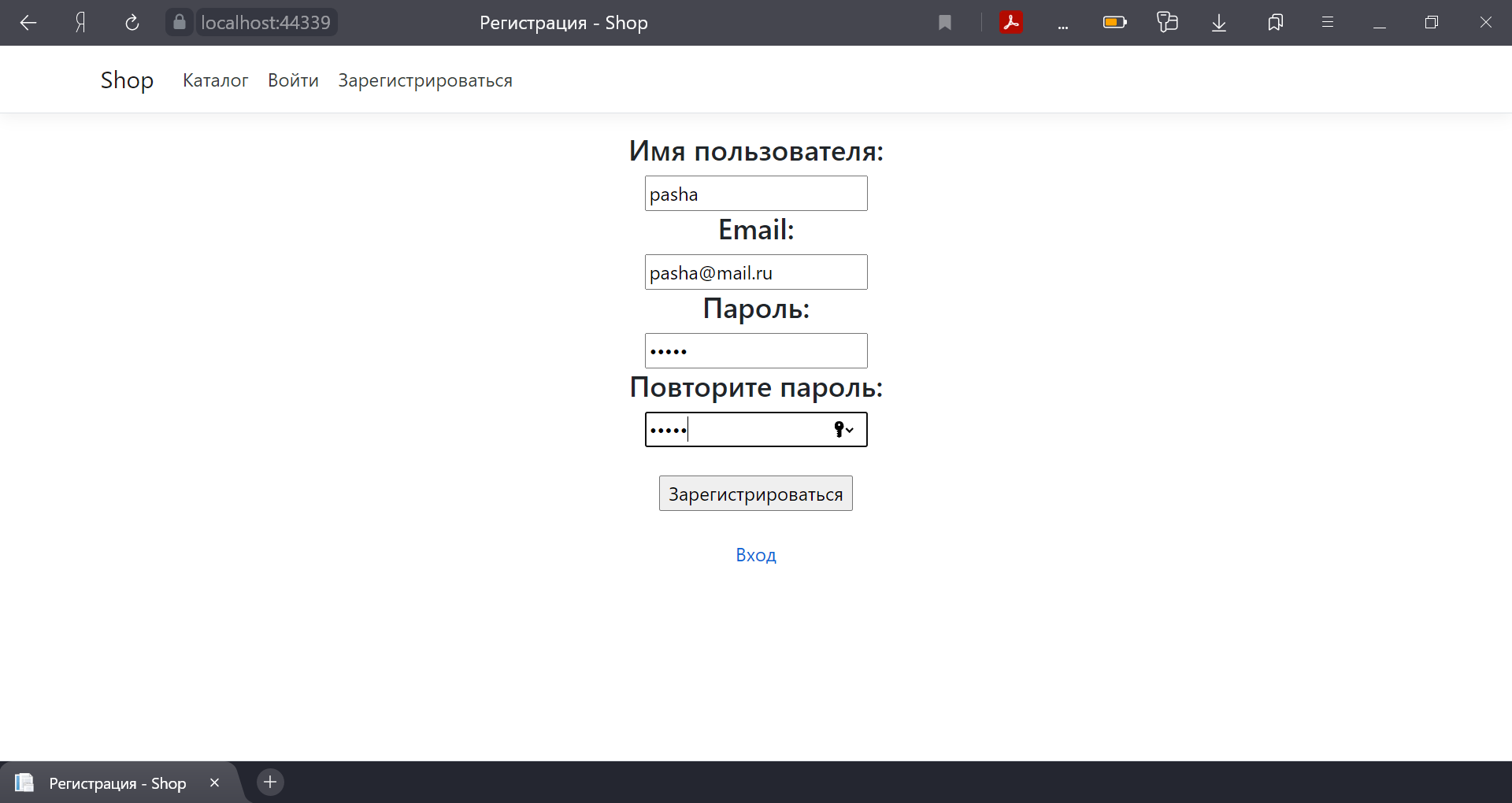
Заходим на главную страницу.



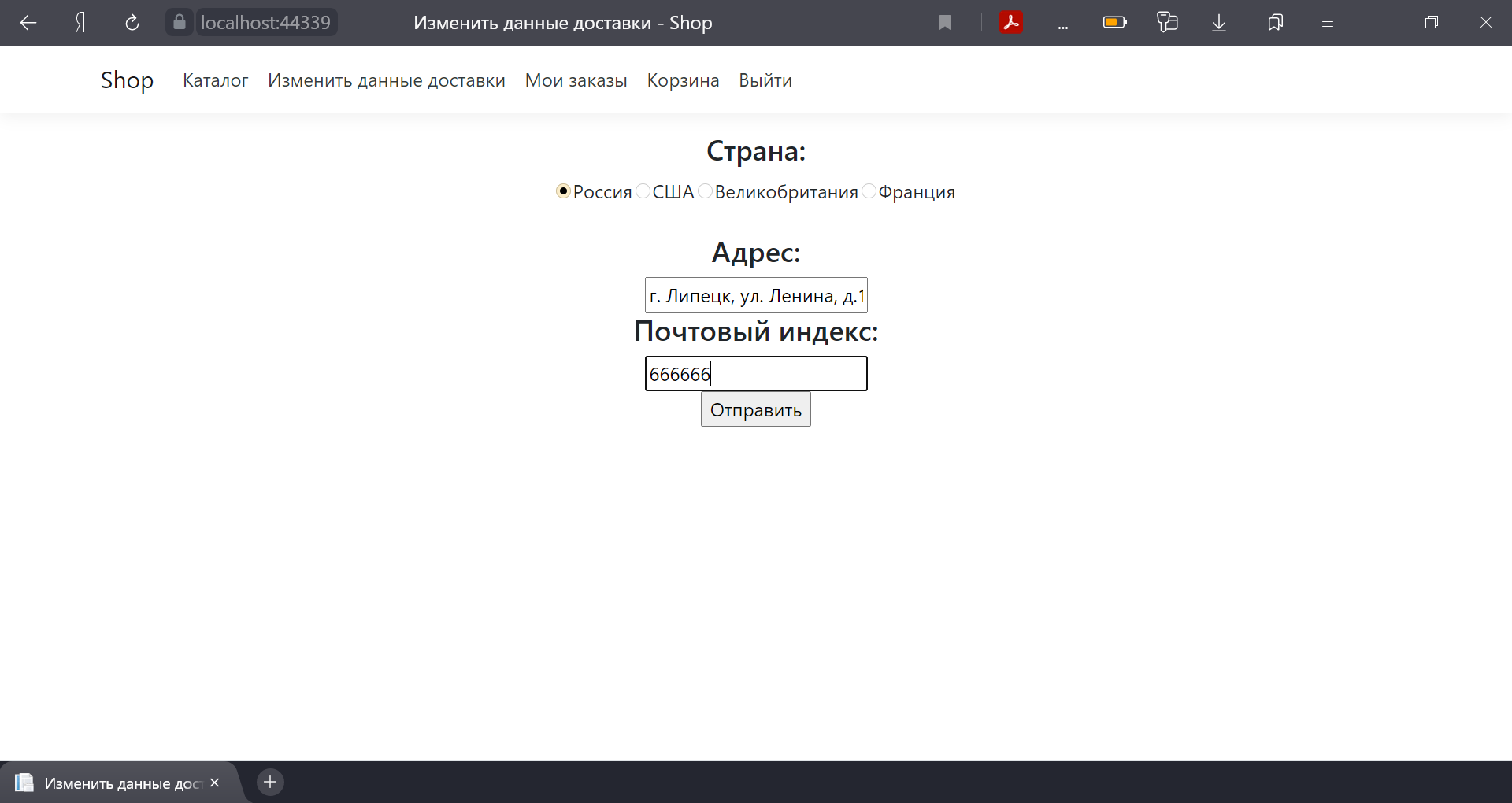
Переходим к интересующему товару.



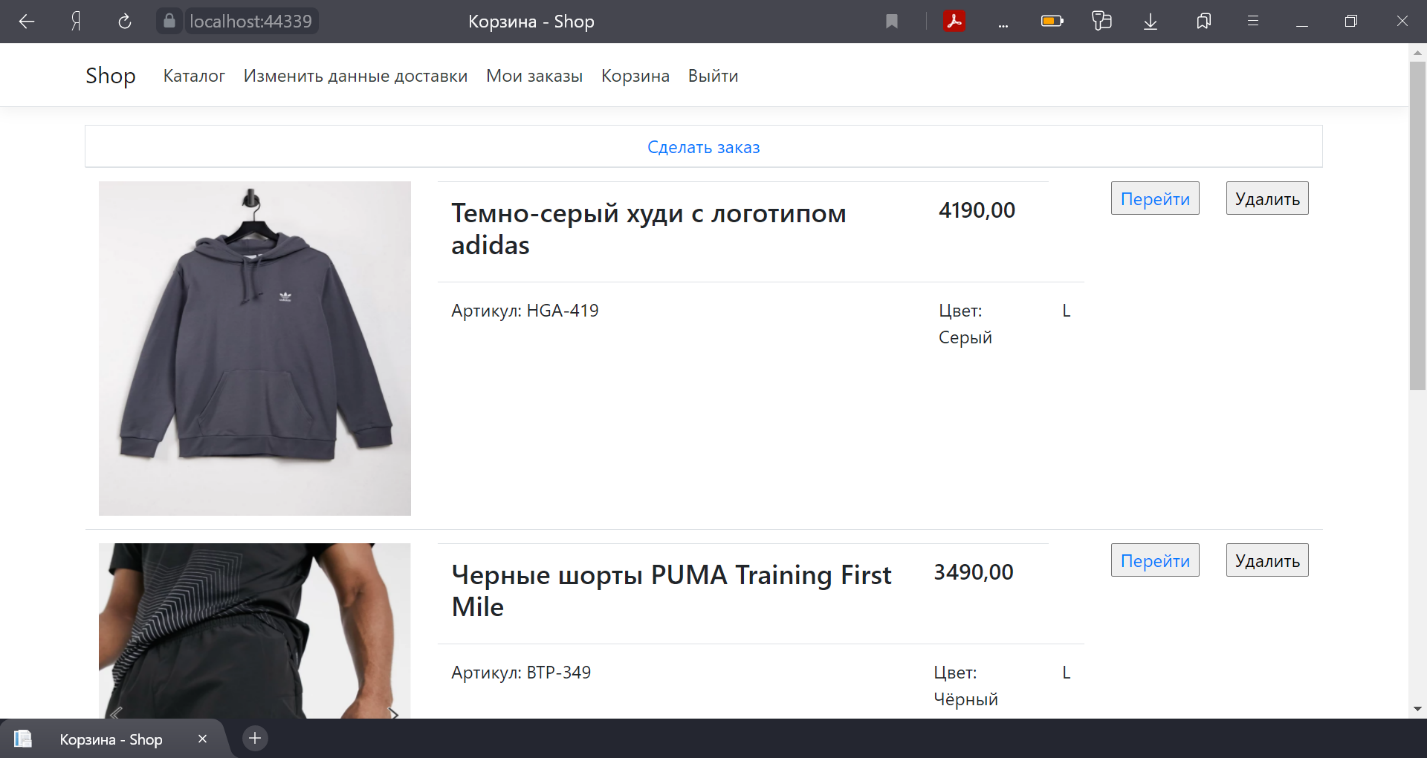
Нельзя добавить в корзину без авторизации, регистрируемся.



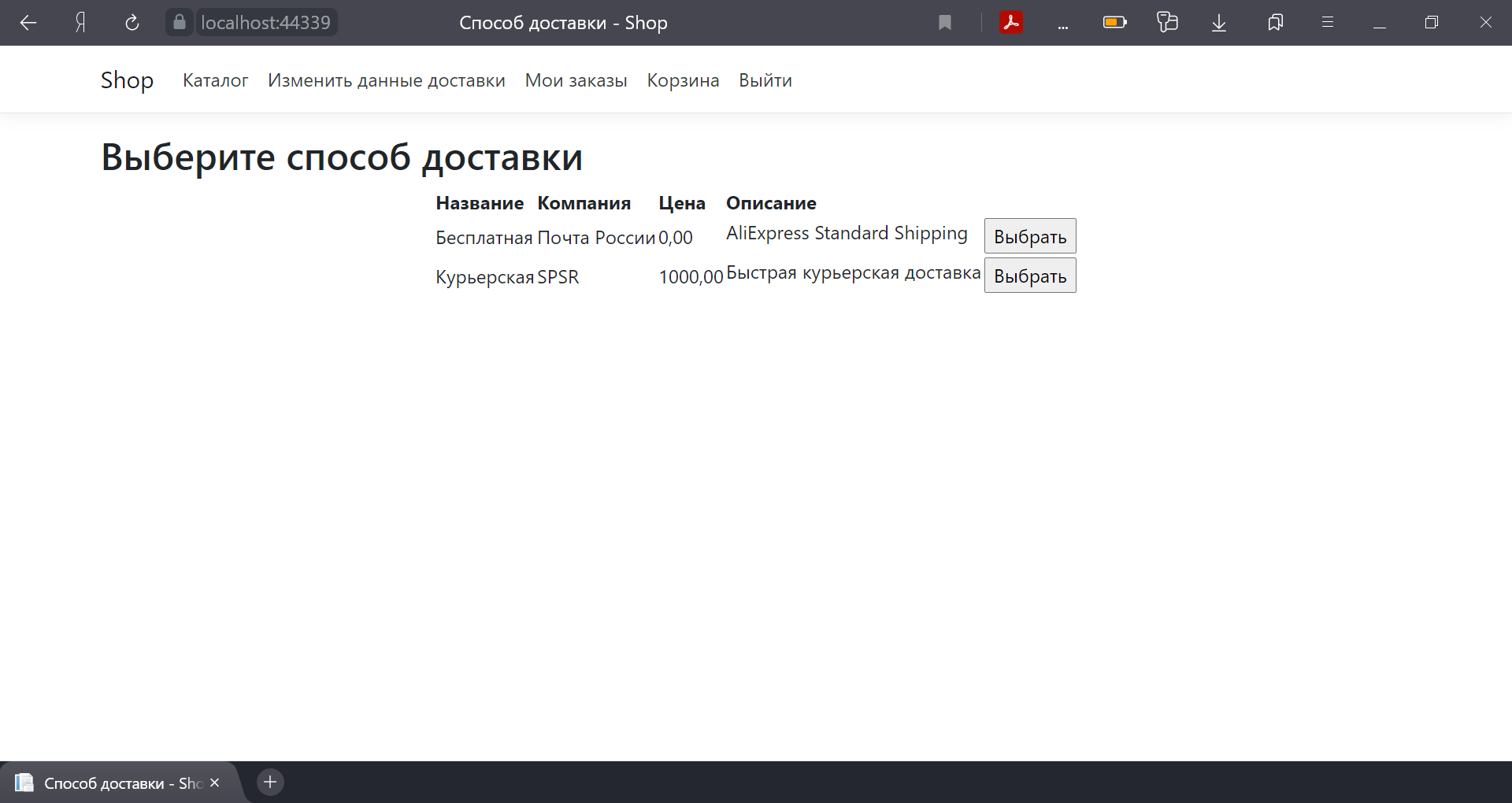
Вносим данные доставки (можно пропустить этот шаг).



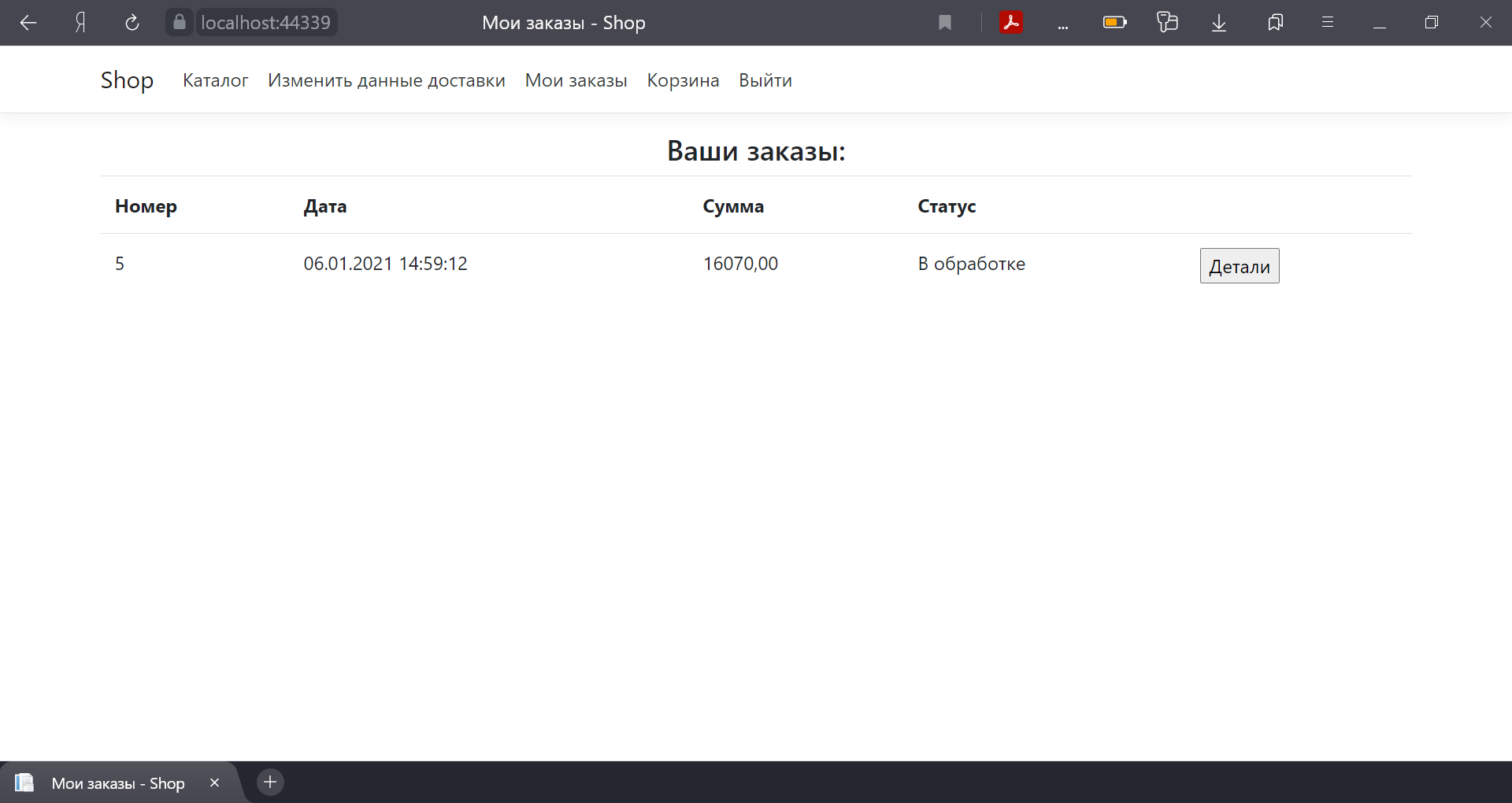
Добавляем интересующие товары в корзину и переходим в неё.



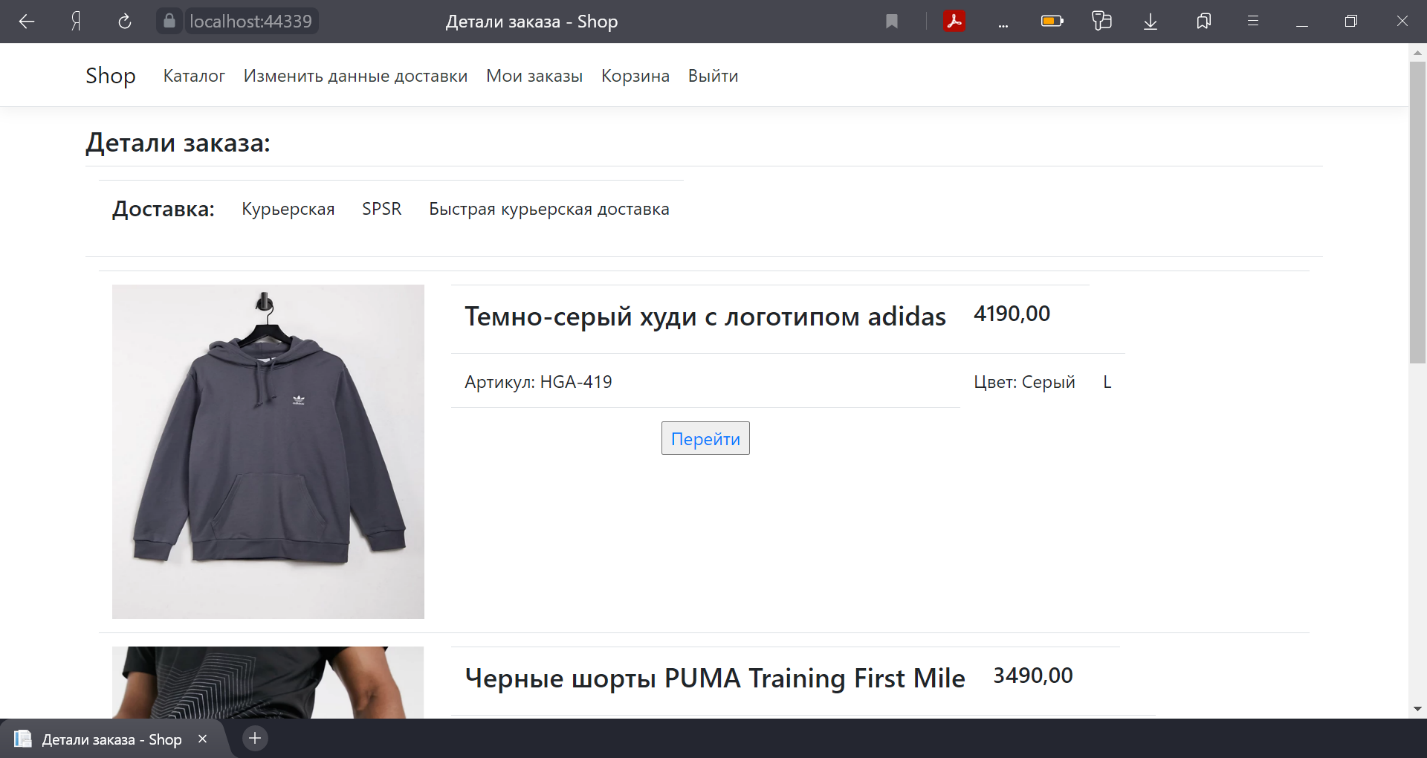
Кликаем кнопку «Сделать заказ», выбираем способ доставки.



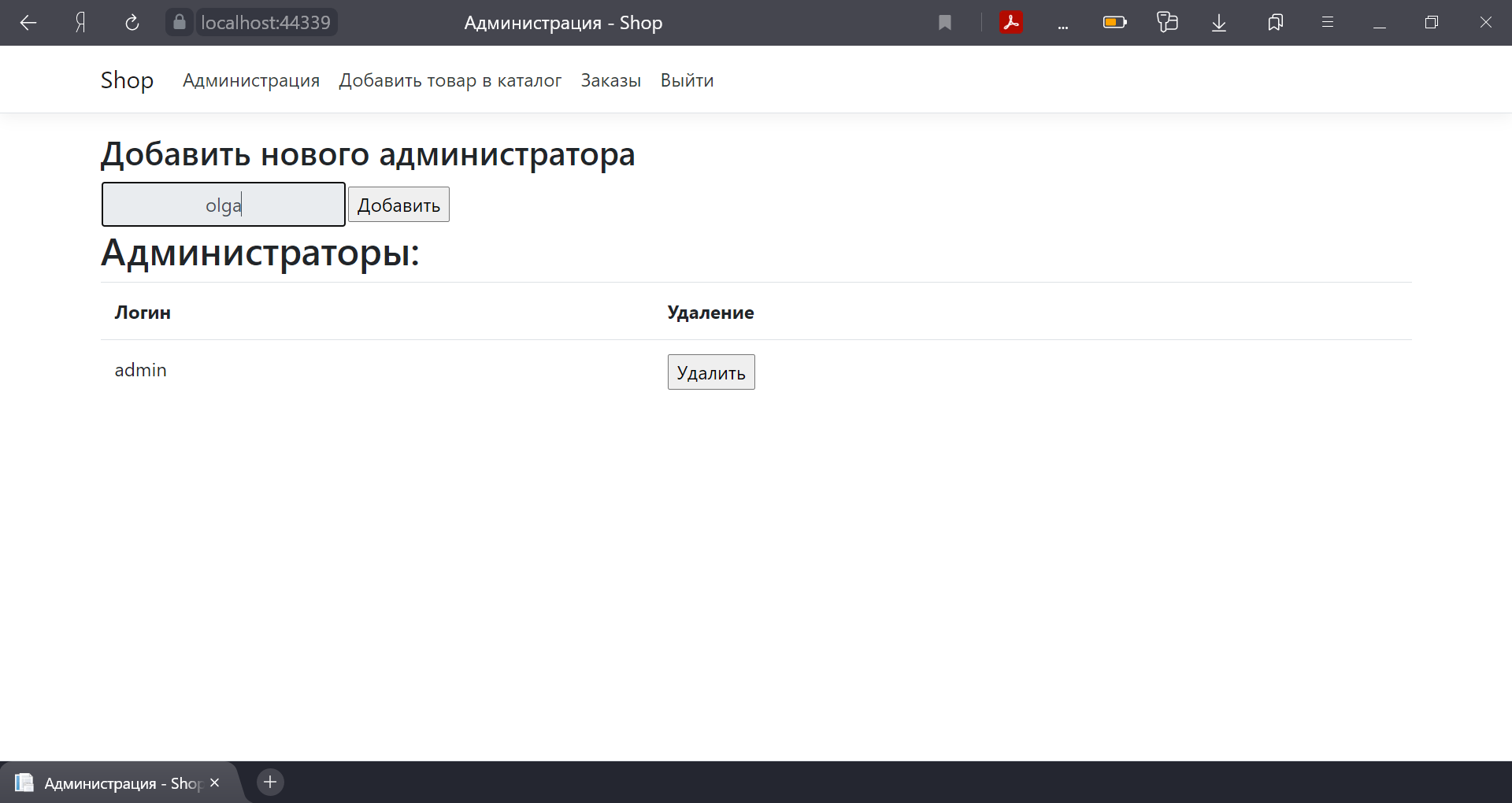
Заказ в обработке.



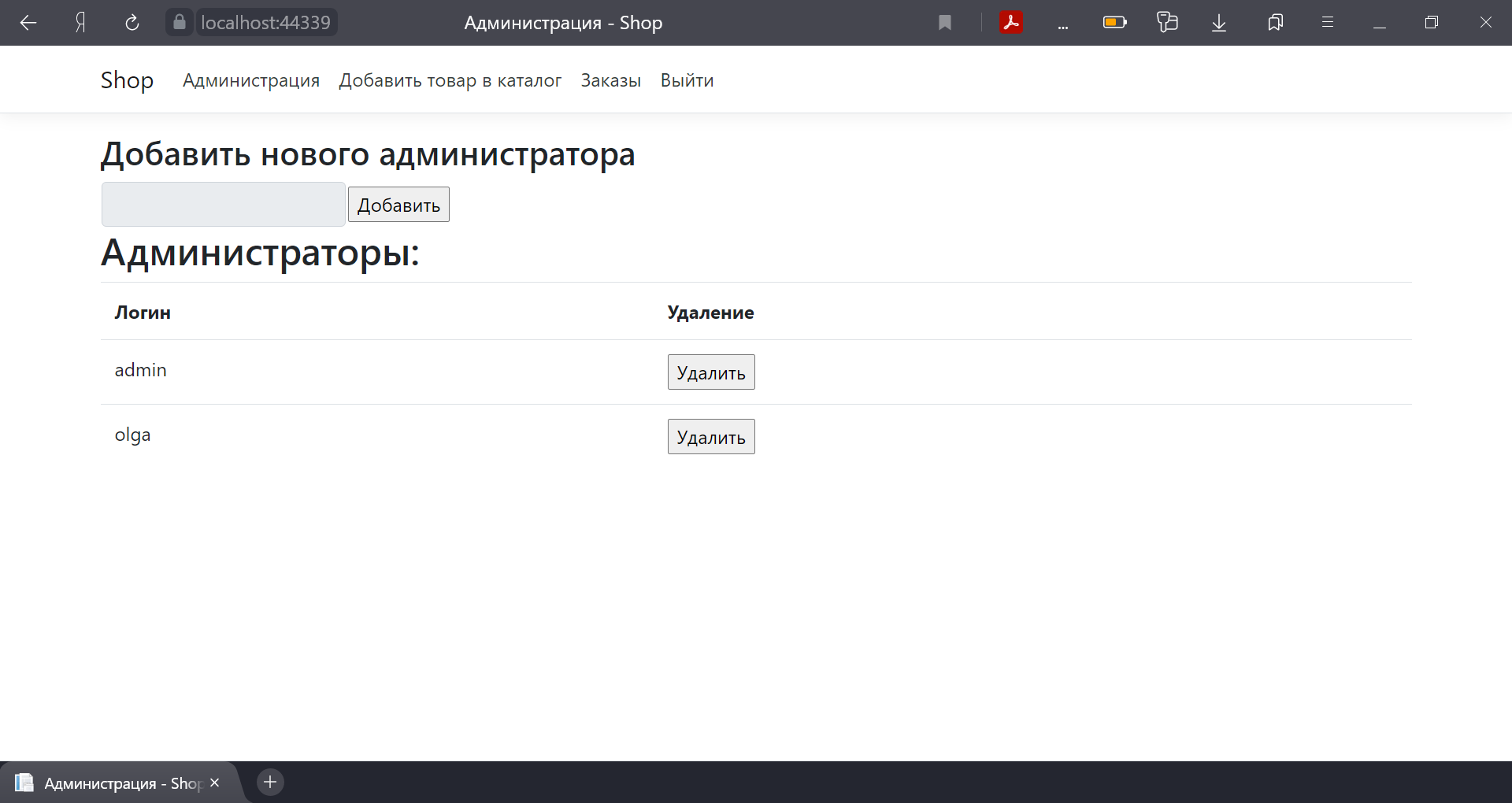
Детали заказа.



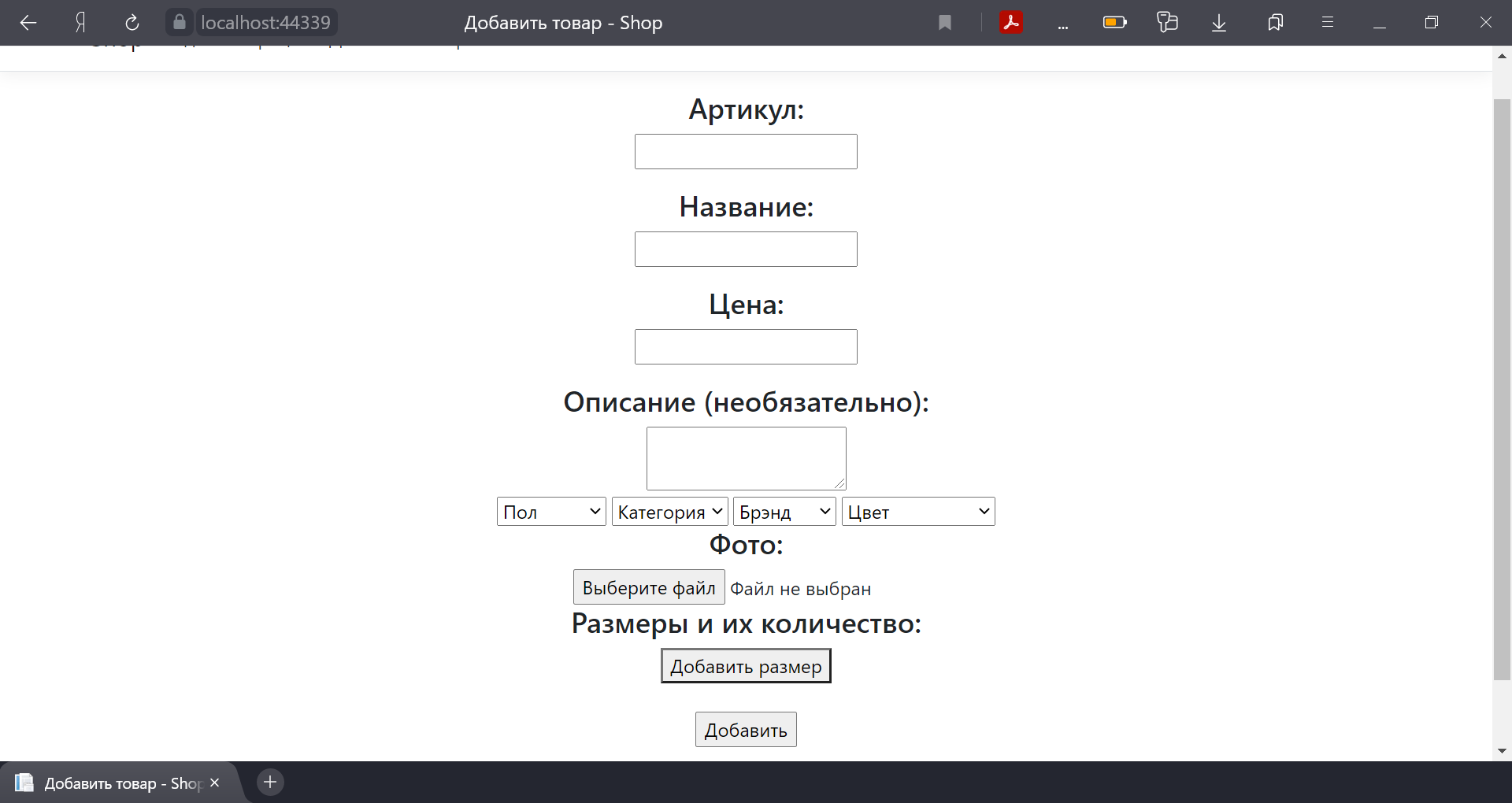
Перейдём в администрацию. Главная страница. Можем добавить администратора.



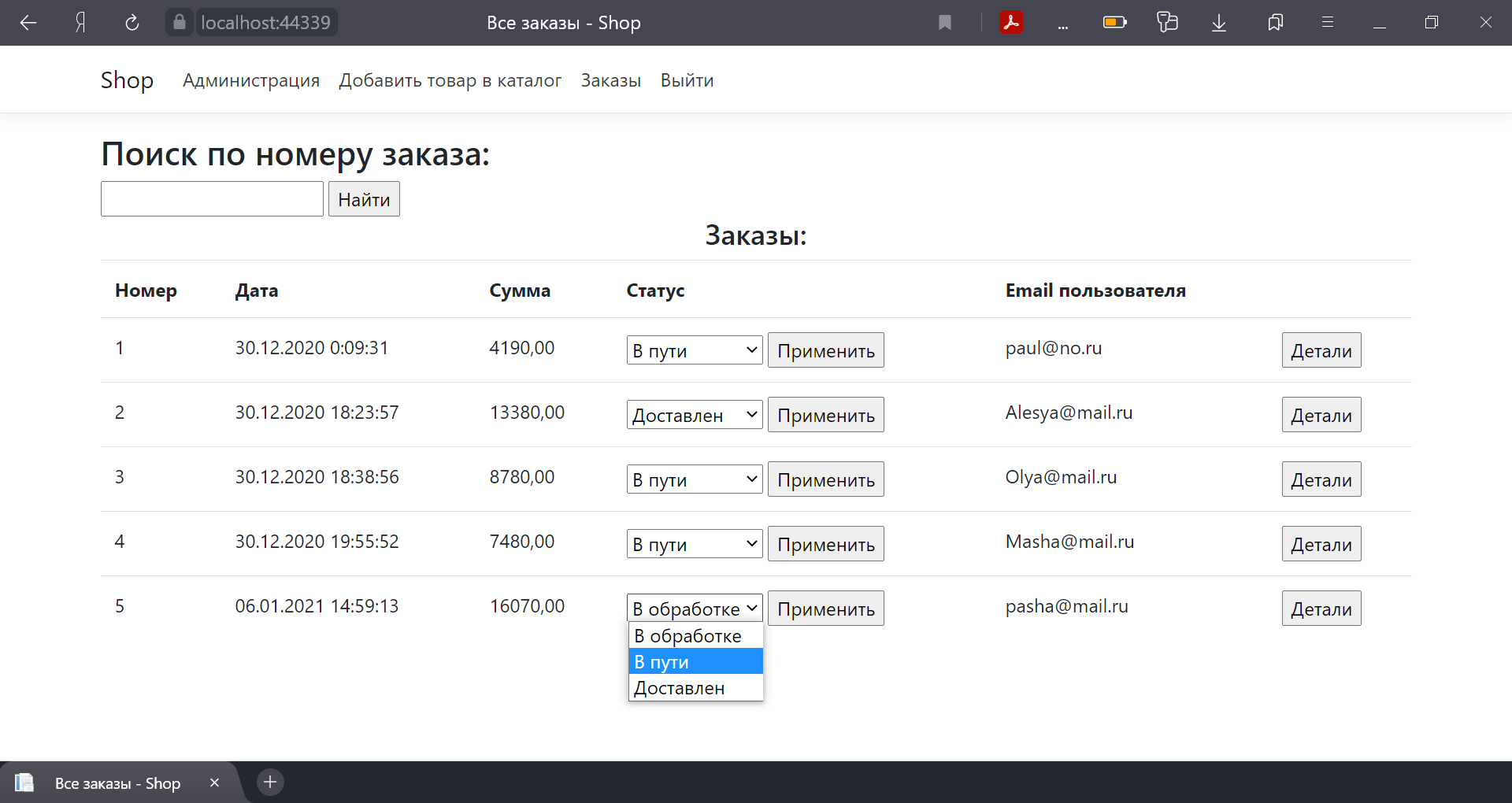
Добавлено.



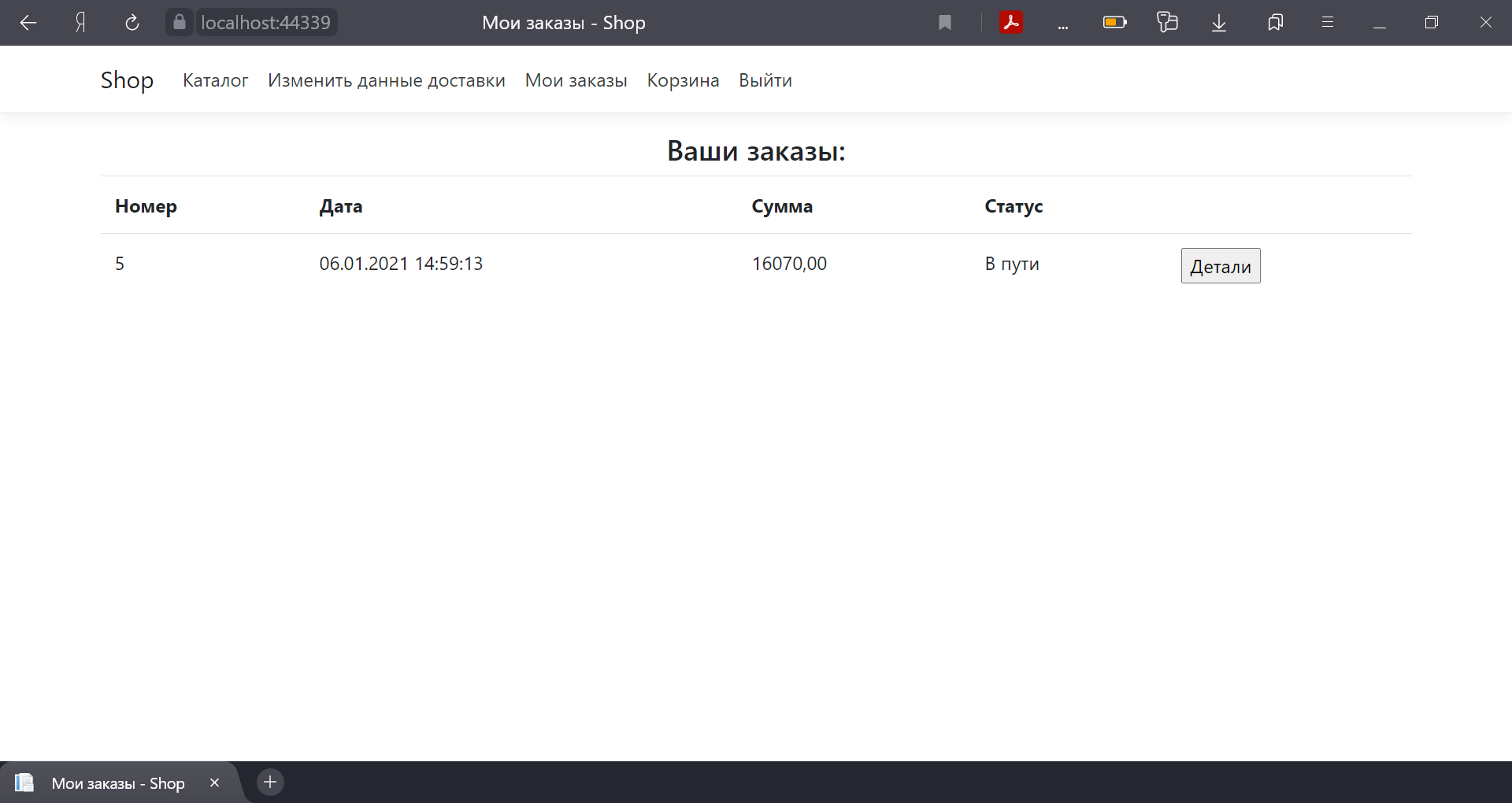
Можем добавить новый товар (все имеющиеся добавлялись через эту форму).



Посмотреть все заказы, изменить их статус. Изменим статус последнего заказа, который мы сделали.



Теперь можем зайти в качестве недавно созданного пользователя, от имени которого мы делали заказ, зайти в «Мои заказы» и увидеть, что статус изменился.



# Список литературы

1. Руководство по ASP.NET Core 5 - <https://metanit.com/sharp/aspnet5/>
   1. Конкретно MVC - <https://metanit.com/sharp/aspnet5/3.1.php>
2. Реструктурирование (scaffold) - <https://docs.microsoft.com/ru-ru/ef/core/managing-schemas/scaffolding?tabs=dotnet-core-cli>
3. Асинхронное программирование - <https://metanit.com/sharp/tutorial/13.3.php>
4. Документация Autofac - <https://autofaccn.readthedocs.io/en/latest/index.html>
5. Передача файлов в ASP.NET Core - <https://docs.microsoft.com/ru-ru/aspnet/core/mvc/models/file-uploads?view=aspnetcore-5.0#upload-small-files-with-buffered-model-binding-to-a-database>