МИНИСТЕРСТВО СЕЛЬСКОГО ХОЗЯЙСТВА РОССИЙСКОЙ ФЕДЕРАЦИИ

КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНЫЙ УНИВЕРСИТЕТ

Факультет прикладной информатики

Кафедра информационных систем и технологий

**Методические указания по Разработке   
системы учета документов с использованием языка Программирования Go**

Исполнитель: Газизов Р. Р.,

факультет "Прикладная информатика",

учебная группа ПИ2002,

Руководитель: Кондратьев В.Ю.

**Краснодар 2022**

**СОДЕРЖАНИЕ**

[Аннотация 4](#_Toc129691947)

[1. Веб-разработка 4](#_Toc129691948)

[2. Цели и задачи 4](#_Toc129691949)

[3. Порядок работы 5](#_Toc129691950)

[Лабораторная работа № 0. Знакомство с проектом. 6](#_Toc129691951)

[1. Цель 6](#_Toc129691952)

[2. Структура веб-приложения 6](#_Toc129691953)

[3. Архитектура проекта 7](#_Toc129691954)

[4. Инструменты и технологии 13](#_Toc129691955)

[5. Порядок дальнейшей работы 15](#_Toc129691956)

[6. Инструкция по запуску docker-образа 16](#_Toc129691957)

[Лабораторная работа № 1. СУБД PostgreSQL. 18](#_Toc129691958)

[1. Цель 18](#_Toc129691959)

[2. Что такое SQL и PostgreSQL 18](#_Toc129691960)

[3. Основы работы с SQL в PostgreSQL 19](#_Toc129691961)

[4. Самостоятельное задание 23](#_Toc129691962)

[Лабораторная работа № 2. Golang и Visual Studio Code. 30](#_Toc129691963)

[1. Цель 30](#_Toc129691964)

[2. Установка Visual Studio Code 30](#_Toc129691965)

[3. Установка Golang 30](#_Toc129691966)

[4. Синтаксис языка Go 32](#_Toc129691967)

[4.1 Переменные и типы данных 33](#_Toc129691968)

[4.2 Циклы и условные конструкции 34](#_Toc129691969)

[4.3 Функции 35](#_Toc129691970)

[4.4 Указатели 37](#_Toc129691971)

[4.5 Структуры, интерфейсы, ООП 39](#_Toc129691972)

[5. Отладка кода 40](#_Toc129691973)

[6. Самостоятельное задание 41](#_Toc129691974)

[Лабораторная работа № 3. Создание Backend-микросервиса. 45](#_Toc129691975)

[1. Цель 45](#_Toc129691976)

[2. Определение структуры микросервиса 45](#_Toc129691977)

[3. Чистая архитектура 46](#_Toc129691978)

[4. Поэтапное написание серверной части 47](#_Toc129691979)

[4.1 Изучение схемы данных 47](#_Toc129691980)

[4.2 Реализация domain слоя 47](#_Toc129691981)

[4.3 Интерфейс сервиса 48](#_Toc129691982)

[4.4 Data transfer objects 49](#_Toc129691983)

[4.5 Репозитории 52](#_Toc129691984)

[4.6 Реализация сервиса 58](#_Toc129691985)

[4.7 Эндпоинты 61](#_Toc129691986)

[4.8 Маршрутизатор 65](#_Toc129691987)

[5. Запуск программы 72](#_Toc129691988)

[6. Тестирование 75](#_Toc129691989)

[Лабораторная работа № 4. Создание Frontend-микросервиса 78](#_Toc129691990)

[1. Цель 78](#_Toc129691991)

[2. Немного теории 78](#_Toc129691992)

[2.1 HTML 78](#_Toc129691993)

[2.2 CSS 80](#_Toc129691994)

[2.3 Materialize CSS 81](#_Toc129691995)

[2.4 JavaScript 83](#_Toc129691996)

[2.5 HTML/Templates 83](#_Toc129691997)

[3. Разработка Frontend-микросервиса 85](#_Toc129691998)

[3.1 Структура папок 86](#_Toc129691999)

[3.2 Установка стилей 87](#_Toc129692000)

[3.3 Панели навигации 89](#_Toc129692001)

[3.4 Создание страниц для товаров 92](#_Toc129692002)

[3.5 Создание обработчиков 101](#_Toc129692003)

[3.6 Создание маршрутизатора 110](#_Toc129692004)

[3.7 Запуск фронтэнда 111](#_Toc129692005)

[4. Самостоятельное задание 112](#_Toc129692006)

[Лабораторная работа № 5. Объединение микросервисов в Docker. 114](#_Toc129692007)

[1. Цель 114](#_Toc129692008)

[2. Что такое Docker 114](#_Toc129692009)

[3. Установка Docker 115](#_Toc129692010)

[4. Dockerfile 115](#_Toc129692011)

[5. Использование docker-compose 117](#_Toc129692012)

[6. Самостоятельное задание 122](#_Toc129692013)

[Лабораторная работа № 6. Микросервис доверенностей. 131](#_Toc129692014)

[1. Цель 131](#_Toc129692015)

[2. Отношение «Многие ко многим» 131](#_Toc129692016)

[3. Создание backend-части 132](#_Toc129692017)

[3.1 Domain-cлой 132](#_Toc129692018)

[3.2 Интерфейс сервиса. 133](#_Toc129692019)

[3.3 Транспортный уровень и репозиторий 134](#_Toc129692020)

[4. Создание frontend-части 135](#_Toc129692021)

[4.1 Определение структур шаблонов 137](#_Toc129692022)

[4.2 Написание HTML-страниц 141](#_Toc129692023)

[4.3 Написание вспомогательных функций 142](#_Toc129692024)

[4.4 Написание обработчиков 144](#_Toc129692025)

[5. Самостоятельное задание 149](#_Toc129692026)

[Заключение 150](#_Toc129692027)

# Аннотация

## Веб-разработка

**Веб-разработка** — одна из самых перспективных и востребованных отраслей в IT. Она включает в себя:

* **Frontend** или **клиентскую** часть. Это создание страниц сайта, анимаций, добавление и отображение динамичного контента, валидация форм. Всё, что мы видим на сайте. Для верстки сайтов используются HTML и CSS, а для создания динамичного контента – JavaScript.
* **Backend** или **серверную** часть. Логика приложения, обработка запросов, использование баз данных, тяжелые вычисления. Для разработки серверной части используется множество языков программирования и технологий. Одно из главных требований — скорость работы. В таких крупных компаниях, как Яндекс, вопреки ожиданиям, используется не Python, а C++ и Go. Для бэкенда очень важна эффективность, поэтому на Python проверяют гипотезы и тесты, а в релизах используют C++ и Go.

В рамках этой методички Вам предстоит с нуля создать веб-приложение, реализовав Frontend и Backend части сайта.

## Цели и задачи

**Задача**: создать систему учета документов.

**Цель**: познакомиться с перечисленными ниже технологиями и инструментами и научиться применять их на практике:

* Язык программирования Go;
* Микросервисная архитектура;
* СУБД PostgreSQL;
* Язык запросов SQL;
* Контейнеризация приложения с помощью Docker;
* Postman для тестирования и отладки API;
* Разметка страниц с помощью HTML и CSS;
* JavaScript для создания динамичного контента страниц.

## Порядок работы

Первым шагом будет знакомство с проектом, его функциональностью и требованиями к разработке.

Далее, в лабораторной работе №1 будет рассмотрена установка и использование СУБД PostgreSQL.

В лабораторной работе №2 мы изучим Golang и научимся работать с интегрированной средой разработки Visual Studio Code.

В лабораторной работе №3 будем создавать backend-микросервис, используя язык программирования Golang.

В лабораторной работе №4 перейдем к созданию frontend-микросервиса с использованием HTML/CSS, JavaScript и библиотеки Materialize CSS.

В лабораторной работе №5 мы объединим микросервисы в Docker-контейнеры для более простой и эффективной работы с проектом.

Наконец, в лабораторной работе №6 мы создадим микросервис доверенностей и добавим свой первый документ в систему. После этого Вам нужно добавить свой документ в систему.

Каждая лабораторная работа позволит изучить и попрактиковаться в использовании различных технологий и инструментов, а полное выполнение всего порядка работы позволит создать полноценную систему учета документов на основе микросервисной архитектуры.

# Лабораторная работа № 0. Знакомство с проектом.

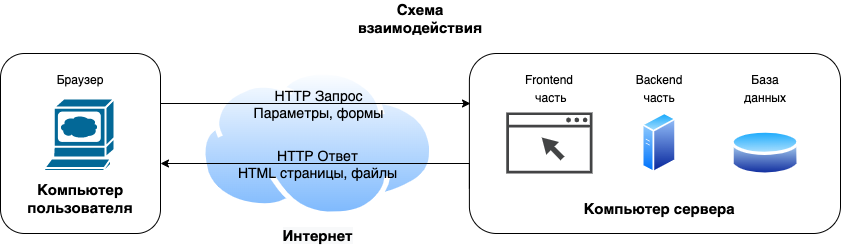
## Цель

Сформировать понимание дальнейшей работы, познакомиться со структурой проекта.

## Структура веб-приложения

Веб-приложение состоит из следующих компонентов:

* **Frontend** (то, что видит пользователь);
* **Backend** (серверная часть);
* **База данных.**



Переходя по адресу в сети, пользователь через интернет передает запросы на сервер (компьютер, на котором работает веб-приложение). Тот адрес, который ввел пользователь, обрабатывается во frontend части приложения:

* Получение параметров;
* Обработка тела запроса;
* Возврат HTML-страниц;
* Маршрутизация;
* Валидация данных;
* и т.д.

Frontend часть приложения предназначена для **отображения** и валидации данных, предоставления нужных страниц и файлов, но ни в коем случае в ней не происходит обращение к базе данных и другие действие, связанные с изменением, созданием и удалением данных.

Frontend является лишь посредником, фильтром между пользователем и бэкендом сайта. Frontend не содержит в себе сложных вычислений, не относящихся к анимациям и отображению. Главная задача frontend – предоставить пользователю удобный интерфейс для взаимодействия с бэкендом.

Так, нажимая кнопку "Создать" мы отправляем на frontend POST-запрос, который содержит в себе данные. Frontend проводит первичную обработку, и, если с данными все в порядке, отправляет запрос на backend, а пользователя уведомляет, что операция прошла успешно.

Да, frontend и backend – это независимые друга от друга приложения, работающие на разных портах ([http://localhost:номер-порта](http://localhost:%D0%BD%D0%BE%D0%BC%D0%B5%D1%80-%D0%BF%D0%BE%D1%80%D1%82%D0%B0)) или даже на разных компьютерах. Общаются frontendи backend с помощью сетевых протоколов и передают данные, например, через JSON или XML.

**Веб-приложение состоит из трёх основных частей: клиентской части, серверной части и базы данных. Клиентская часть является интерфейсом для пользователя и написана на HTML, CSS, JavaScript. Серверная часть обрабатывает запросы от клиента и отвечает на них. База данных хранит информацию, которую использует веб-приложение, например, данные пользователей или контент.**

## Архитектура проекта

Что такое **микросервисная архитектура**?

Говоря о микросервисной архитектуре, можно провести аналогию с большим домом, где каждая комната, по сути, является **независимым**, самодостаточным микросервисом, а соединены комнаты через двери, то есть, **интерфейсы**, у которых есть лишь один метод: СменитьКомнату(комната Room).

В нашем случае микросервисом является приложение, слушающее запросы по определенному порту и выдающее требуемые данные. **Порт** – это номер квартиры в многоэтажном доме. Вводя в строке браузера адрес http://localhost:8080/, пользователь хочет обратиться к приложению, "*слушающему*" порт 8080. Вводя адрес http://localhost:80/, пользователь обращается к приложению, работающему на порту 80.

Вот, как это выглядит в коде:

package main // главный модуль программы

import "net/http" // модуль для работы в сети

func main() {

handler := getHandler() // Получить экземпляр маршрутизатора

http.ListenAndServe(":8080", handler) // Слушать и обслуживать порт ":8080" обработчиком запросов handler

}

func getHandler() http.Handler {

// ...

}

**Таким образом, весь сайт построен на небольших независимых приложениях, общающихся между собой по сети.**

Вот так выглядит структура папок этого проекта:

project

├── docker-compose.yml

├── go.mod

├── go.sum

├── services

│ ├── account

│ ├── base

│ ├── customer

│ ├── documents

│ ├── employee

│ ├── home

│ ├── organization

│ ├── product

│ │ ├── backend

│ │ │ ├── cmd

│ │ │ ├── implementation

│ │ │ ├── repo

│ │ │ ├── service

│ │ │ └── transport

│ │ │ └── http

│ │ ├── domain

│ │ ├── dto

│ │ └── frontend

│ │ ├── cmd

│ │ ├── handlers

│ │ ├── router

│ │ ├── static

│ │ │ ├── html

│ │ │ └── js

│ │ └── utils

│ ├── proxy

│ └── utils

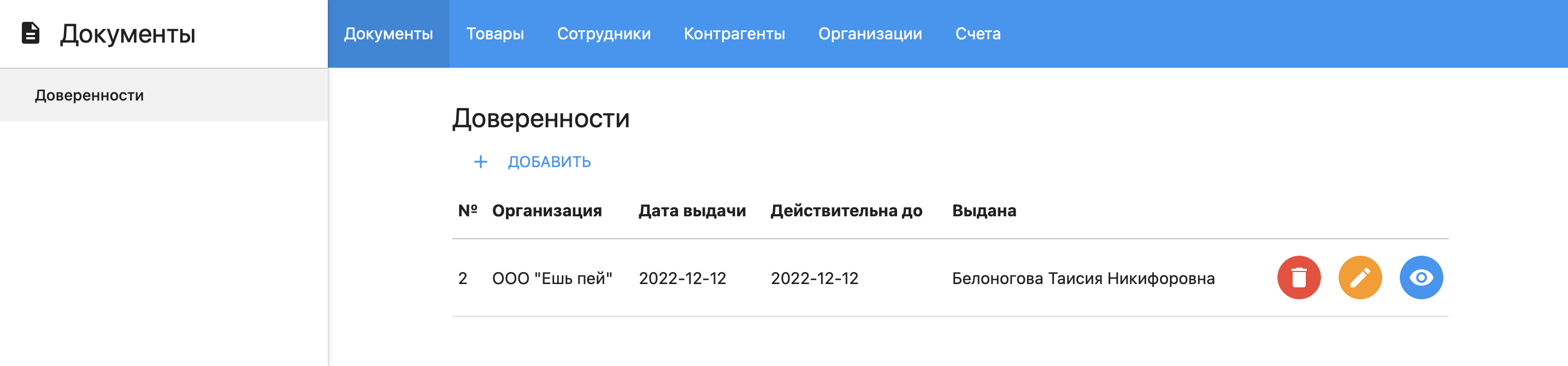
├── sql

├── static

│ ├── css

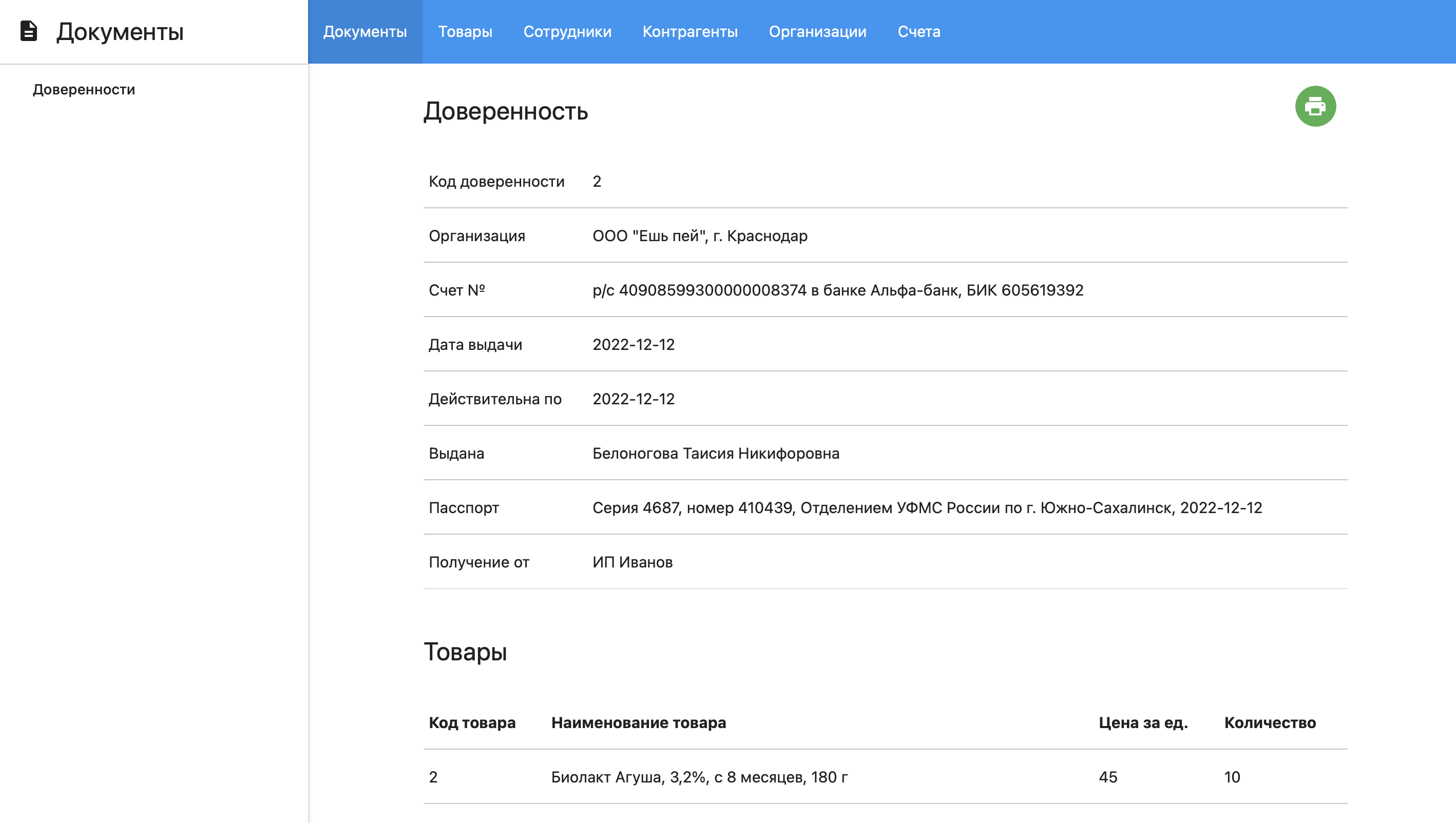
│ └── js

└── vendor

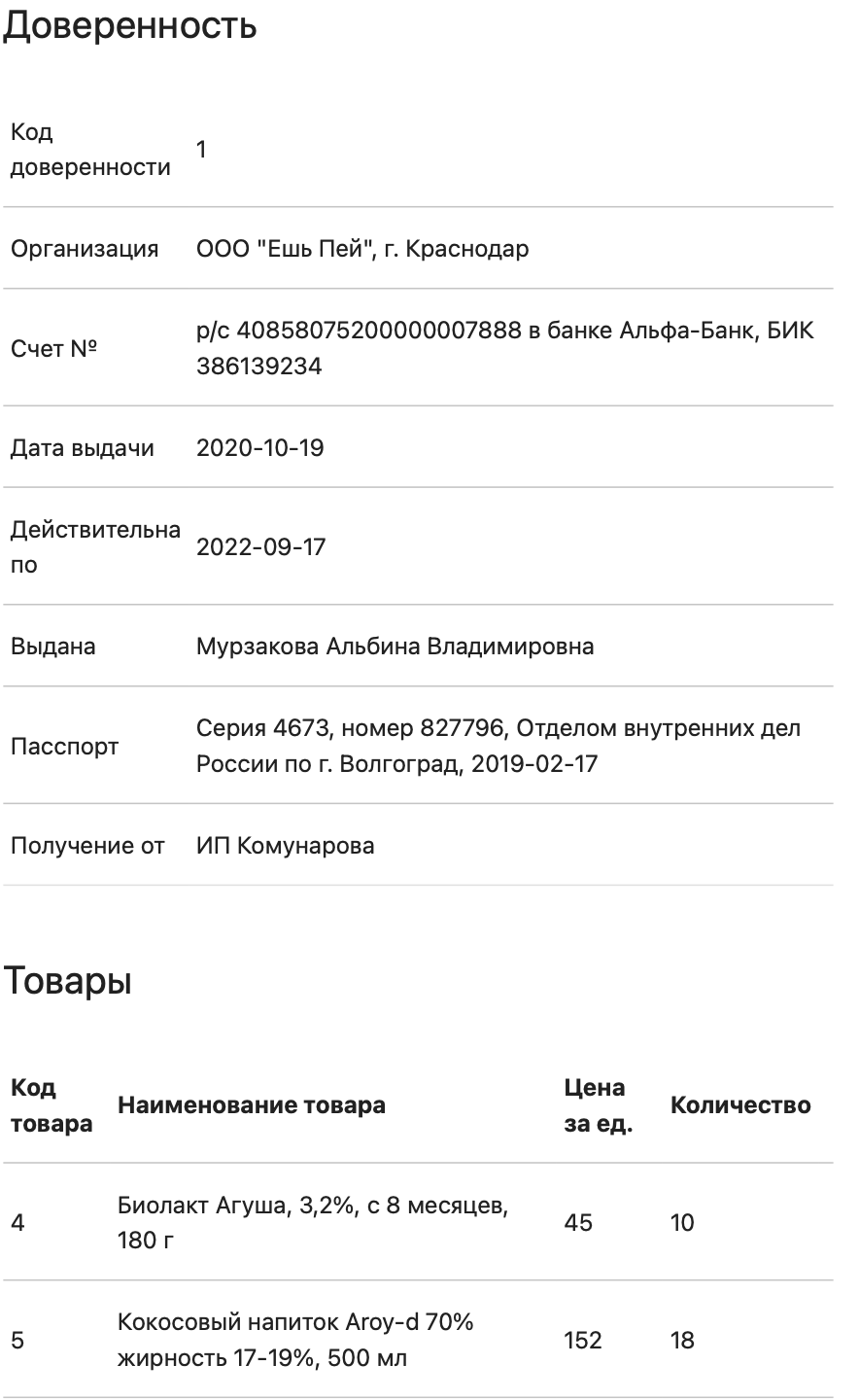


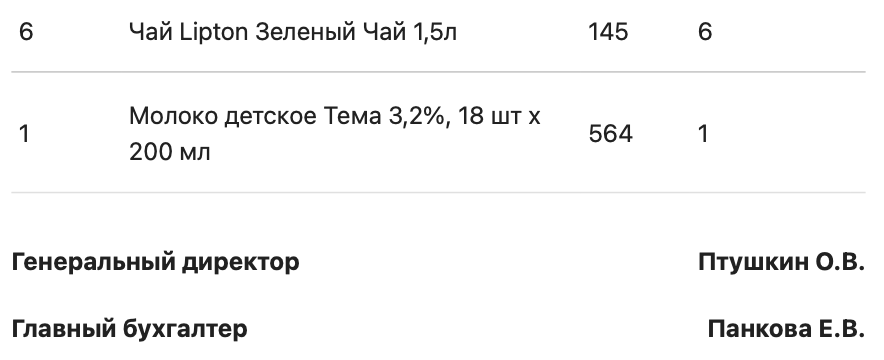
Интерфейс пользователя представляет собой сайт с верхней и боковой панелями навигации. Сверху расположены основные элементы системы, а сбоку их детализация.

Основная часть и суть этой системы заложена во вкладке «Документы». При нажатии на голубую кнопку напротив доверенности пользователь переходит на подробную страницу.

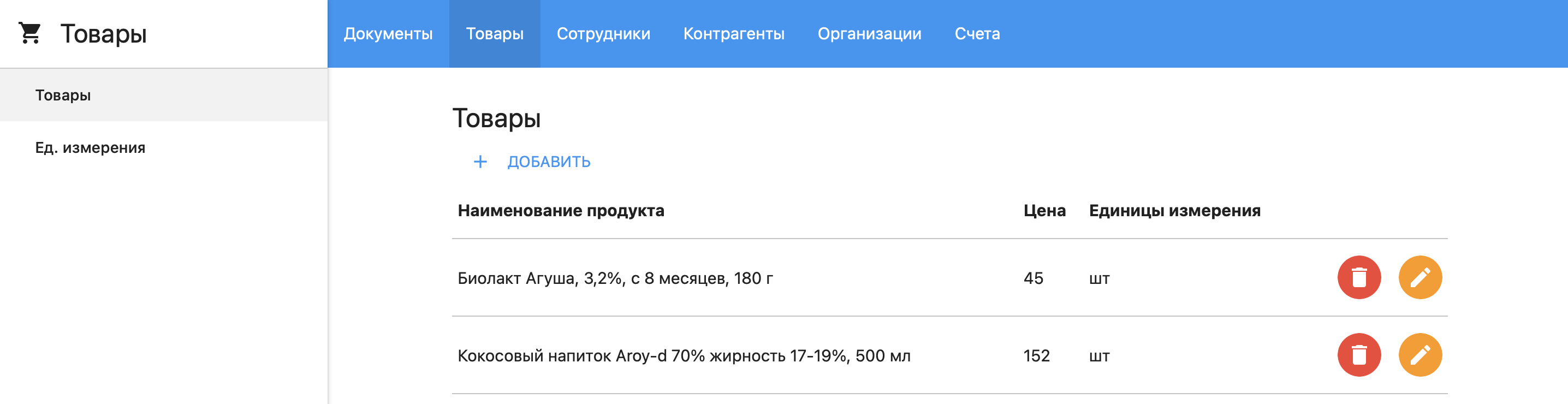


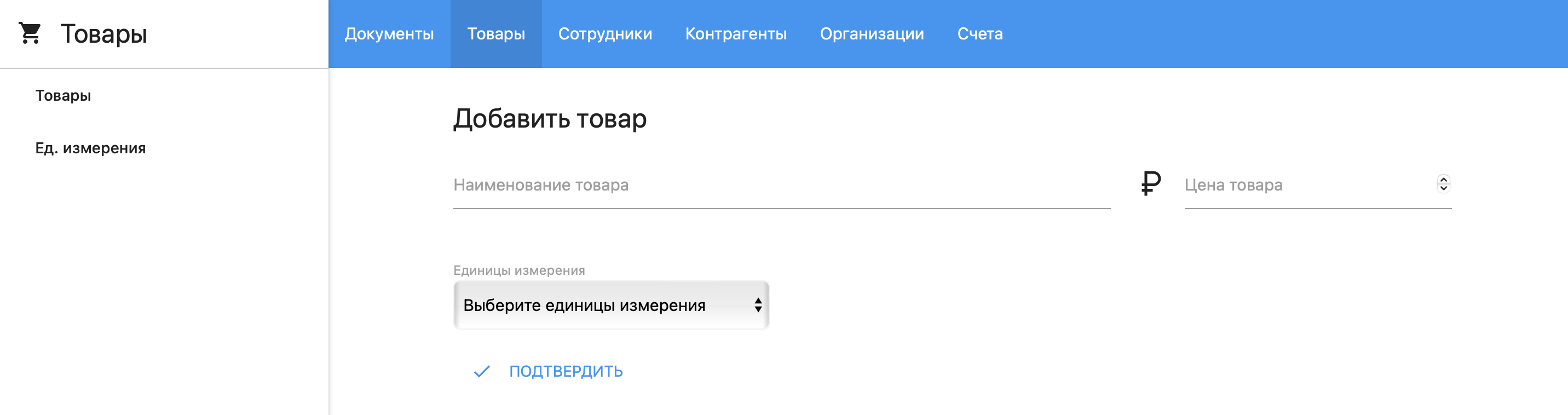
Документы в системе должны иметь возможность быть распечатанными. При нажатии на зеленую кнопку печати из страницы создается лист для печати:





Формы для прочих данных гораздо проще, чем для документов:





Стоит принимать во внимание, что во избежание усложнения системы многие элементы были упрощены. Например, цена товара – это целое число, а не дробное, так как для обработки денежных величин требуется подключать специальные библиотеки.

Работа с организациями, товарами, клиентами и т.д. вынесена в отдельные сервисы. Внутри этих сервисов есть разделение на следующие директории:

* backend;
* frontend;
* domain;
* dto.

**«domain»** включает в себя модель данных или **entity**. Она общая для обоих микросервисов и вынесена в отдельную папку. Директории «backend**»** и «frontend**»** — два независимых приложения, работающих самостоятельно. Часто при попытке обратиться к неработающему сервису мы все равно видим страницу с надписью "Закручиваем гайки". Это означает, что сломался backend, но frontend всё равно работает. «**dto»** — data transfer objects. Структуры, которые содержат модель данных для JSON.

**Микросервисный подход облегчает процесс разработки, уменьшая область видимости разработчика. Чем меньше сущностей требуется держать в голове, тем легче сконцентрироваться на текущей задаче**

## Инструменты и технологии

**Язык программирования Go**

В рамках всего проекта будет использоваться язык программирования **Go**.

**Go** (часто также **Golang**) — компилируемый многопоточный язык программирования, разработанный внутри компании Google. Язык Go разрабатывался как язык программирования для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах. Он может рассматриваться, как попытка создать замену языкам Си и C++ с учётом изменившихся компьютерных технологий и накопленного опыта разработки крупных систем. По словам Роба Пайка, «Go был разработан для решения реальных проблем, возникающих при разработке программного обеспечения в Google»

Основными требованиями к языку стали:

* язык должен предоставлять небольшое число средств, не повторяющих функциональность друг друга;
* простая и регулярная грамматика. Минимум ключевых слов, простая, легко разбираемая грамматическая структура, легко читаемый код;
* простая работа с типами. Типизация должна обеспечивать безопасность, но не превращаться в бюрократию, лишь увеличивающую код. Отказ от иерархии типов, но с сохранением объектно-ориентированных возможностей;
* отсутствие неявных преобразований;
* сборка мусора;
* встроенные средства распараллеливания, простые и эффективные;
* поддержка строк, ассоциативных массивов и коммуникационных каналов;
* чёткое разделение интерфейса и реализации;
* эффективная система пакетов с явным указанием зависимостей, обеспечивающая быструю сборку.

**Golang принято использовать для серверной части, но во избежание увеличения сложности проекта, Go будет использоваться и во frontend части.**

**HTML, CSS, JavaScript**

Весь веб-контент строится с помощью HTML, CSS и JavaScript.

**HTML** – язык разметки страниц. Состоит из открывающих и закрывающих тегов и представляет **содержание** страницы.

**CSS** – набор стилей, описывающих оформление страницы.

**JavaScript** – язык, на котором работает Web. Установлен во всех браузерах по умолчанию. JavaScript используется для того, чтобы "оживить" страницу. Анимации, валидаторы, уведомления, связь с серверной частью — все это осуществляется через JavaScript.

Так как HTML, CSS и JavaScript заслуживают отдельной дисциплины, в проекте будут использоваться самые базовые вещи, а также готовые стили CSS. Это не помешает понять принципы работы frontend, но заметно упростит разработку сервиса.

**HTML — содержание страницы, CSS — оформление страницы,   
JavaScript — динамика страницы.**

**Visual Studio Code —** бесплатная легковесная IDE. Поддерживает множество языков программирования, имеет большое сообщество, одна из самых популярных IDE в мире.

**Docker** — платформа, позволяющая "запаковывать" приложения в виде образов (OS Linux + приложение и все необходимые ресурсы и библиотеки) и воспроизводить их в контейнере (виртуальная машина, на которой запущен образ).

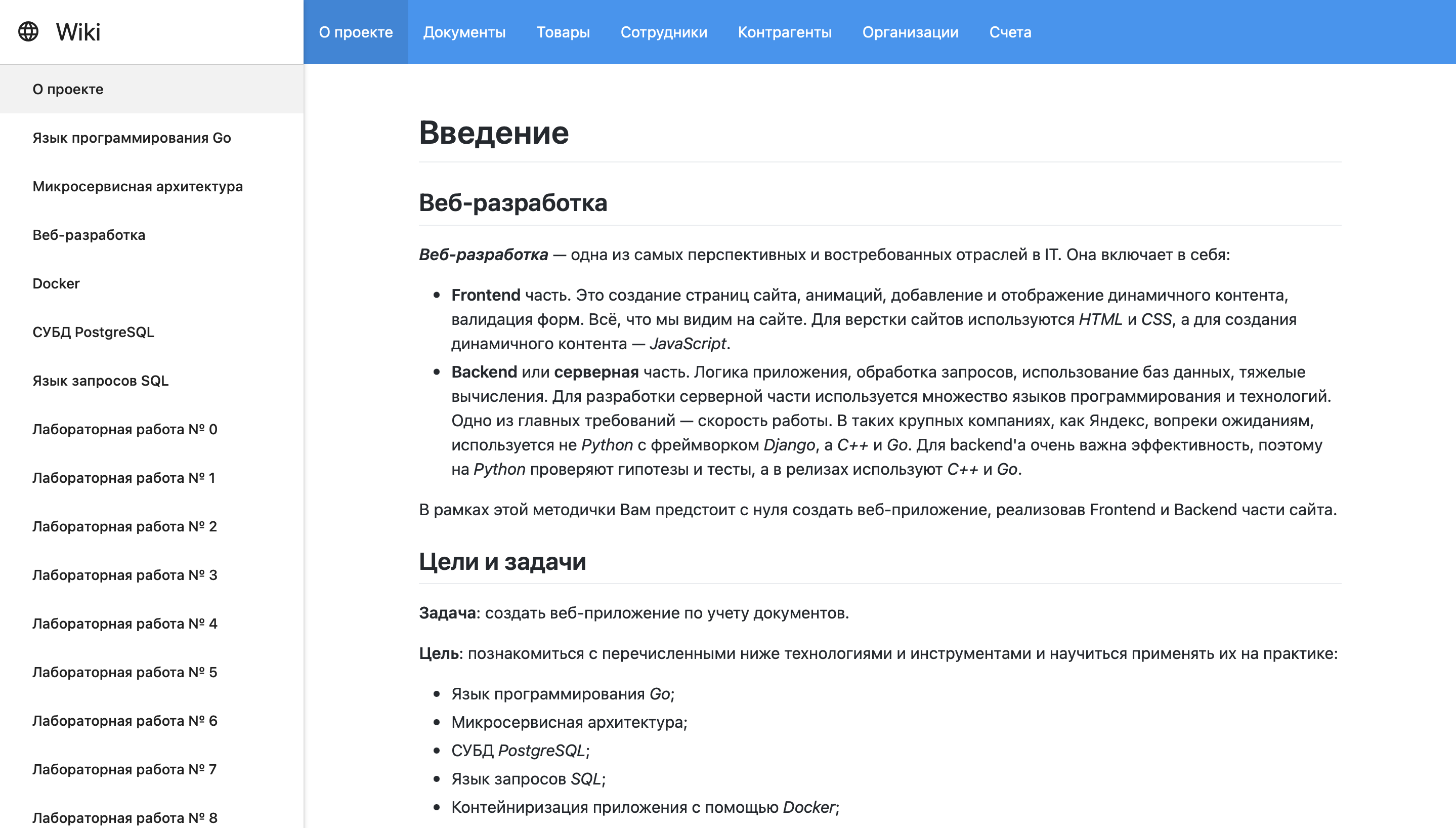
**PostgreSQL** — это мощная и надежная система управления реляционными базами данных (СУБД) с открытым исходным кодом.

## Порядок дальнейшей работы

Работа над проектом начнется с определения **ER-модели** (схемы данных). Основываясь на этой модели, необходимо сформировать перечень микросервисов, номера портов и затрагиваемые таблицы для каждого микросервиса.

Следующий этап — создание базы данных и проектирование микросервисов один за другим. Проектирование включает (в порядке реализации):

* создание таблиц;
* backend-часть;
* тестирование API с помощью postman;
* frontend-часть;
* размещение микросервиса с помощью Docker.



## Инструкция по запуску docker-образа

Эта часть необязательна, но она поможет понять, какой продукт будет на выходе, а также познакомит Вас с платформой Docker.

Весь проект сразу доступен для запуска с помощью докера. Чтобы запустить проект, установите Docker согласно инструкции из пятой лабораторной работы. Затем возьмите исходный код у ведущего преподавателя и скопируйте его к себе на компьютер. Откройте папку с исходным кодом в терминале и запустите две команды: docker-compose build (команда может работать вплоть до пяти минут на слабом ПК), затем docker-compose up.

Откройте браузер и перейдите по адресу: http://localhost/documents/proxies. Изучите сайт.

**Так, реализовывая микросервисы один за другим, получится создать масштабируемую, модульную систему управления документами в виде веб-приложения, контейнеризированного с помощью Docker.**

**Да прибудет с вами сила!**

# Лабораторная работа № 1. СУБД PostgreSQL.

## Цель

Создать базу данных для системы учета документов.

## Что такое SQL и PostgreSQL

**SQL** **(Structured Query Language)** - это язык программирования, который используется для работы с реляционными базами данных. SQL позволяет создавать, изменять, удалять и управлять данными в таблицах базы данных.

**PostgreSQL** - это реляционная СУБД (система управления базами данных), которая использует язык SQL для работы с данными. PostgreSQL поддерживает множество функций, включая транзакции, поддержку многопоточности, репликацию данных и многое другое.

PostgreSQL является бесплатной и открытой системой управления базами данных, доступной для использования на различных платформах, включая Windows, Linux и MacOS. PostgreSQL также обладает большим и активным сообществом пользователей и разработчиков, которые создают дополнительные функции и инструменты для улучшения работы с этой СУБД.

Некоторые из наиболее распространенных СУБД, которые поддерживают SQL, включают в себя:

* MySQL - это бесплатная и открытая реляционная СУБД, которая широко используется веб-приложениями.
* Oracle Database - это коммерческая СУБД от компании Oracle, которая используется для хранения и управления большими объемами данных в предприятиях.
* Microsoft SQL Server - это коммерческая СУБД от компании Microsoft, которая используется в Windows-средах для управления данными и создания веб-приложений.
* SQLite - это легковесная и быстрая СУБД, которая используется в мобильных приложениях.

PostgreSQL является мощной, открытой и бесплатной реляционной СУБД, которая имеет множество преимуществ перед другими СУБД:

1. **Надежность**: PostgreSQL является одной из самых надежных СУБД и предлагает механизмы восстановления после сбоев, резервное копирование и многое другое, что гарантирует сохранность данных.
2. **Масштабируемость**: PostgreSQL может эффективно обрабатывать миллионы записей и поддерживать многопоточность, что позволяет легко масштабировать базу данных при росте количества данных.
3. **Гибкость**: PostgreSQL позволяет создавать пользовательские функции, определять новые типы данных и планировать задачи, что дает разработчикам большую гибкость и контроль над базой данных.
4. **Безопасность**: PostgreSQL предлагает высокий уровень безопасности и может использоваться в критических приложениях, таких как финансовые системы и системы управления персональными данными.
5. **Поддержка**: PostgreSQL является открытой СУБД, что означает, что она имеет активное сообщество разработчиков.

## Основы работы с SQL в PostgreSQL

* Основные типы в PostgreSQL:

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| INTEGER или INT | Этот тип используется для хранения целых чисел в диапазоне от -2147483648 до 2147483647. |
| BIGINT | Этот тип используется для хранения целых чисел в диапазоне от -9223372036854775808 до 9223372036854775807. |
| REAL и DOUBLE PRECISION | Вещественные числа (real и double precision): типы данных, которые представляют числа с плавающей запятой. Тип real используется для хранения чисел с одинарной точностью, а тип double precision - для хранения чисел с двойной точностью. |
| TIMESTAMP и DATE | Дата и время (timestamp и date): типы данных, которые представляют дату и время. Тип timestamp используется для хранения даты и времени с точностью до микросекунд, а тип date - для хранения только даты. |
| VARCHAR(N) | Символьные строки (varchar): тип данных, который представляет строку переменной длины N. Этот тип используется для хранения текстовых данных. |
| TEXT | Тип text представляет строку переменной длины без ограничения на длину. Он часто используется для хранения текстовых данных, которые могут быть различных размеров. |
| SERIAL | Тип serial используется для создания столбцов, которые автоматически генерируют уникальные числовые значения при вставке новых строк в таблицу. Тип serial фактически является аналогом для целочисленного типа данных bigint и предоставляет удобный способ создания автоматически инкрементируемых столбцов с уникальными значениями. |
| Массивы(ARRAY), JSON, XML и т.д. | PostgreSQL позволяет хранить данные в виде массивов, а также JSON и XML объектов |

* **Создание.** Для того чтобы создать таблицу, необходимо выполнить следующий SQL-запрос:

CREATE TABLE table\_name (

id SERIAL PRIMARY KEY, -- ключевое поле, по которому идентифицируется запись

field\_name TYPE NOT NULL, -- наименование поля, его тип и дополнительные квалификаторы

foreign\_key\_field INT NOT NULL REFERENCES another\_table(id) ON DELETE SET DEFAULT ON UPDATE CASCADE, -- установка внешнего ключа к другой таблице

-- При удалении внешнего ключа будет установлено значение по умолчанию

-- При обновлении внешнего произойдет каскадное обновление в этой таблице

-- ... другие поля

)

Для дальнейшей работы возьмем, к примеру, таблицу «products», содержащую четыре поля: id, name, price, unit\_id.

CREATE TABLE products(

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

price INT NOT NULL,

unit\_id INT NOT NULL REFERENCES units(id) ON DELETE SET DEFAULT ON UPDATE CASCADE

)

* **Добавление**:

INSERT INTO products(name, price, unit\_id) VALUES

(‘Чай Lipton Зеленый Чай 1,5л’, 145, 0)

Для добавления используется команда «INSERT INTO», затем пишется название таблицы (products), перечисляются затрагиваемые поля в скобках (поля, которые не указаны, получают значения по умолчанию, или null, если возможно, иначе – ошибка), служебное слово VALUES, и в скобках перечисляются значение в соответствии с полями.

* **Получение**:

SELECT products.id, products.name, products.price, units.name

FROM products

INNER JOIN units

ON units.id = products.unit\_id

ORDER BY products.name ASC

Команда **SELECT** используется для извлечения данных из таблиц products и units, связанных между собой с помощью внешнего ключа unit\_id.

SELECT products.id, products.name, products.price, units.name указывает, какие столбцы необходимо выбрать из таблиц products и units. В данном случае мы выбираем id, name и price из таблицы products и name из таблицы units.

FROM products INNER JOIN units ON units.id = products.unit\_id определяет, из каких таблиц извлекать данные и как они связаны друг с другом. Здесь мы используем оператор INNER JOIN, чтобы объединить строки из таблиц products и units, связанных между собой по столбцу unit\_id.

ORDER BY products.name ASC определяет порядок сортировки результатов. Здесь мы сортируем результаты по столбцу name таблицы products в алфавитном порядке (ASC - от меньшего к большему).

* **Обновление**:

UPDATE products SET price = 100 WHERE id = 1

D

Команда UPDATE используется для изменения данных в таблице **products**.

UPDATE products определяет таблицу, которую мы хотим изменить.

SET price = 100 WHERE id = 1 указывает новое значение столбца price для тех строк таблицы products, которые соответствуют условию id = 1. В данном случае значение столбца price будет изменено на 100.

Таким образом, эта команда UPDATE изменит значение столбца price на 100 для строки таблицы products, у которой значение столбца id равно 1.

* **Удаление**:

DELETE FROM products WHERE id = 1

Команда DELETE FROM используется для удаления записей из таблицы по условию WHERE.

В данном случае из таблицы products удалится запись, где id = 1.

* Прочие команды:

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| ALTER TABLE table\_name RENAME COLUMN old\_name TO new\_name; | ALTER TABLE используется для изменения структуры таблицы (например, для добавления новых столбцов, изменения типов данных или удаления столбцов). Переименование столбца. |
| ALTER TABLE my\_table ADD COLUMN new\_column INTEGER; | Добавление столбца |
| ALTER TABLE my\_table ALTER COLUMN column\_name TYPE new\_data\_type; | Изменение типа столбца |
| ALTER TABLE my\_table DROP COLUMN column\_name; | Удаление столбца |

## Самостоятельное задание

* 1. Перейдите на официальный сайт: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads;
  2. Выберите вашу платформу и последнюю версию (15.2 на момент написания);
  3. Запустите установщик по завершении скачивания;
  4. Нажимайте кнопку Next в установщике до страницы с компонентами;
  5. **Снимите** галочку со «Stack Builder»;
  6. Оставьте путь по умолчанию;
  7. Введите пароль «**qwerty**» для администратора сервера;
  8. Нажимайте далее, пока не начнется установка;
  9. После завершения установки нажмите «Finish».

Эксплуатировать PostgreSQL можно в двух режимах:

* Через командную строку (CLI);
* Через пользовательский интерфейс (pgAdmin 4).

В рамках этой методички будет использоваться интерфейс командной строки, так как пользовательский интерфейс программы «pgAdmin 4» требует определенных знаний и опыта, а также предлагает гораздо больше, чем требуется в лабораторных работах. Консольный ввод очень удобный и быстрый, позволяет не отвлекаться на освоения интерфейса и сконцентрироваться на SQL-запросах непосредственно.

Для использования psql в командной строке необходимо добавить путь к каталогу bin в переменную среды PATH.

Для добавления пути к каталогу bin PostgreSQL в переменную среды PATH на Windows, выполните следующие шаги:

* 1. Откройте панель управления и выберите "Система и безопасность" (System and Security), затем выберите "Система" (System).
  2. Нажмите на кнопку "Дополнительные параметры системы" (Advanced System Settings).
  3. В открывшемся окне выберите вкладку "Дополнительно" (Advanced), затем нажмите на кнопку "Переменные среды" (Environment Variables).
  4. В разделе "Системные переменные" (System Variables) найдите переменную "Path" и нажмите на кнопку "Изменить" (Edit).
  5. В окне "Изменить системную переменную" (Edit environment variable) нажмите на кнопку "Новый" (New) и введите путь к каталогу bin PostgreSQL (например, "C:\Program Files\PostgreSQL\15\bin").
  6. Нажмите на кнопку "ОК" во всех окнах для сохранения изменений.

В связи с особенностями работы командной строки Windows необходимо проверить настройки языка для корректного использования русских символов в psql. Откройте Панель управления → Часы и регион → Региональные стандарты → вкладка «Дополнительно» вверху окна → Изменить язык системы → убедитесь, что в выпадающем меню выбран язык «Русский (Россия)»

1. Нажмите «Ок»;
2. Откройте терминал и введите chcp. Если вывод не содержит 1251, то введите команду chcp 1251 для смены кодировки командной строки;
3. Теперь введите команду: psql -U postgres -h localhost
4. Пароль для пользователя «**postgres**» тот же, что вы вводили при установке: «**qwerty»**;
5. Строка приветствия должна стать такой: postgres=#
6. Введите команду с обязательным сохранением точки с запятой: CREATE USER student PASSWORD ‘qwerty’;
7. Теперь от роли суперпользователя необходимо создать базу данных и назначить владельца: CREATE DATABASE proxy\_db OWNER student;
8. Нажмите сочетание клавиш Ctrl + C, чтобы выйти из режима psql, и подключитесь к созданной базе данных: psql -U student -d proxy\_db
9. Введите пароль: qwerty;

Вы готовы к работе! Небольшая справка по командам в psql:

|  |  |
| --- | --- |
| **Команда** | **Описание** |
| \dt | Выводит список таблиц |
| \d table\_name | Показать информацию о таблице (столбцы, типы данных, ограничения и т.д.) |
| \du | Список всех пользователей |
| \q или Ctrl + C | Выход из psql |

В ходе самостоятельного задания необходимо создать базу данных и заполнить каждую таблицу минимум одной строкой.

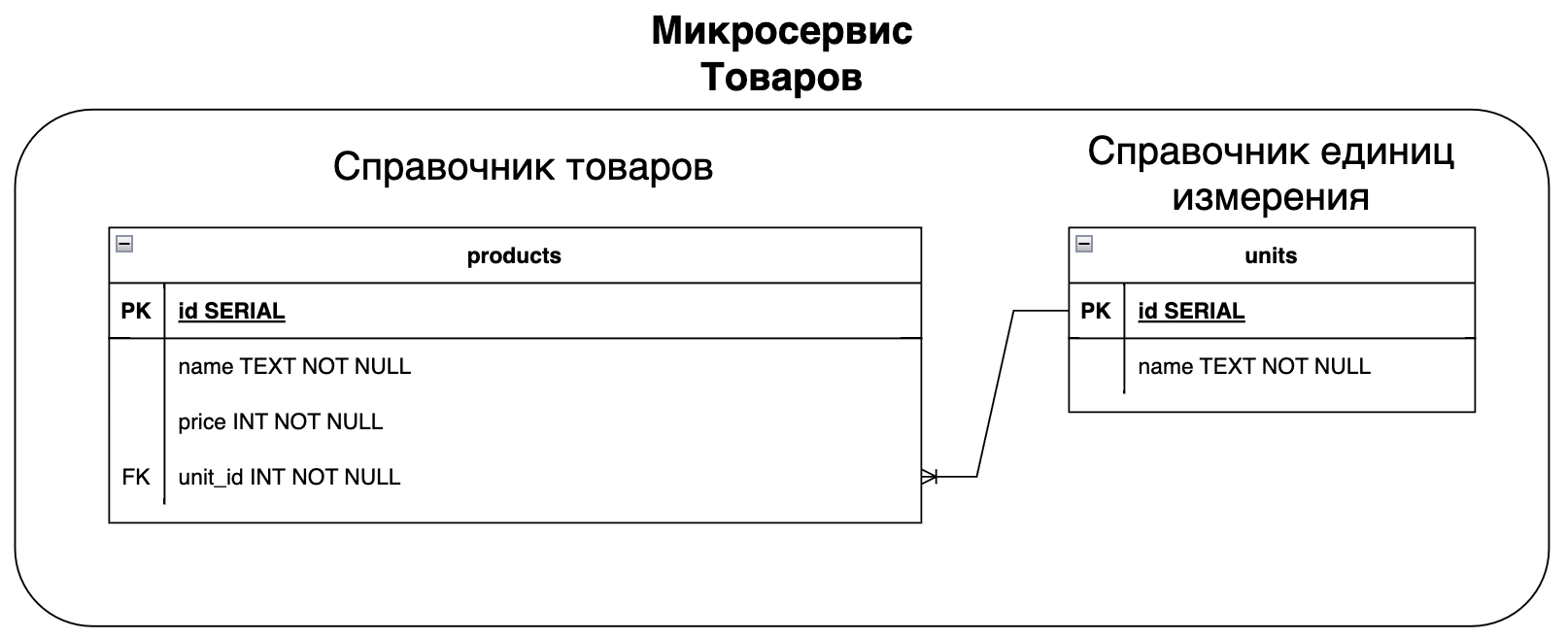
Логически схема поделена на шесть микросервисов со своими таблицами:

|  |  |
| --- | --- |
| **Микросервис** | **Таблицы** |
| Товары | products, units |
| Организации | organizations |
| Счета | accounts |
| Контрагенты | customers |
| Сотрудники | employees |
| Доверенности | proxies, proxy\_bodies |

Схема данных приложения выглядит следующим образом:



Рассмотрим и реализуем схему данных для микросервиса товаров:



Линия, соединяющая таблицы units и products отображает отношение   
**один-ко-многим**. Соответственно, «один» относится к units, а «многим» к products. Как понимать это отношение? У одной единицы измерения (например, «шт.») может быть много товаров или много товаров имеют единицу измерения «шт.»

Есть важное теоретическое правило, которому всегда нужно следовать: из PK **не может** выходить «много».

Еще одно важное практическое правило: таблица, к которой имеется REFERENCE, должна быть создана **первее**, чем таблица, из которой происходит REFERENCE.

Введенный вами SQL-запрос выполняется, когда в конце строки появляется символ точки с запятой «;». То есть, вы можете переносить запрос, нажимая Enter, и psql его запомнит.

* **Создание таблицы units:**

CREATE TABLE units (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL

);

* **Создание таблицы products:**

CREATE TABLE products (

id SERIAL PRIMARY KEY,

name TEXT NOT NULL,

price INT NOT NULL,

unit\_id INT NOT NULL DEFAULT 0 REFERENCES units(id) ON DELETE SET DEFAULT ON UPDATE CASCADE

);

* **Заполнение таблиц**

INSERT INTO units(id, name) VALUES (0,‘Не указано’);

INSERT INTO units(id, name) VALUES (1,‘шт.’);

INSERT INTO products(name, price, unit\_id) VALUES

(‘Чай Lipton Зеленый Чай 1,5л’, 145, 1)

Аналогично, создайте остальные таблицы. Создавая внешние ключи других таблиц, установите **запрет** на удаление записи, если на неё есть ссылка из создаваемой таблицы, например:

account\_id INT NOT NULL REFERENCES accounts(id) ON DELETE RESTRICT ON UPDATE CASCADE

Все запросы сохраните в файл **/sql/create\_tables.sql** в корневой папке проекта.

# Лабораторная работа № 2. Golang и Visual Studio Code.

## Цель

Установить среду разработки VSCode, язык программирования Go, изучить синтаксис и особенности работы.

## Установка Visual Studio Code

1. Откройте браузер и перейдите на официальный сайт VSCode (https://code.visualstudio.com/);
2. На главной странице сайта найдите и нажмите кнопку "Download" (скачать) в верхней части экрана;
3. Выберите подходящую версию для вашей операционной системы (Windows, macOS, Linux), и нажмите кнопку "Download" рядом с ней;
4. Когда загрузка завершится, найдите загруженный файл на вашем компьютере и запустите его;
5. В окне установки выберите язык, настройте дополнительные параметры (включая установку дополнительных компонентов, если это необходимо), и нажмите кнопку "Next" (далее);
6. Примите лицензионное соглашение и нажмите кнопку "Next";
7. Выберите папку, в которую будет установлен VSCode, и нажмите кнопку "Next";
8. На следующем экране вы можете выбрать, создать ли ярлык на рабочем столе или в меню "Пуск", а также другие параметры. Выберите нужные опции и нажмите кнопку "Install" (установить);
9. Дождитесь завершения установки, и нажмите кнопку "Finish" (завершить).

## Установка Golang

1. Перейдите на официальный сайт Golang: https://golang.org/;
2. Нажмите кнопку Download;
3. Скачайте установщик для вашей операционной системы и запустите его;
4. Следуйте инструкциям установщика и нажимайте кнопку Next, пока не начнется установка;
5. По умолчанию Go устанавливается в папку C:/Program Files;
6. Проекты на Go принято хранить по следующему пути: C:/Program Files/Go/src/github.com/<YOUR\_LOGIN>/<PROJECT\_NAME>. Перейдите в C:/Program Files/Go/src → создайте папку **github.com** → перейдите в папку github.com → создайте папку и назовите её **вашим** логином github (если у вас нет гитхаба, просто выберите любой ваш логин) → перейдите в эту папку → создайте папку hello-world;
7. Запустите Visual Studio Code от имени **администратора**. Нажмите в верхнем меню Файл-Открыть папку... и выберите папку hello-world;
8. Нажмите сочетание клавиш CTRL + SHIFT + ` (буква Ё) или кликните по Терминал → Создать терминал в верхнем меню для открытия **терминала** внутри VS Code;
9. В открывшемся терминале пропишите следующую команду: go mod init github.com/<YOUR\_LOGIN>/hello-world;
10. Через контекстное меню (правая кнопка мыши) создайте файл main.go в директории hello-world;
11. Если у вас не установлено расширение для языка Go, вам будет предложено его установить при первом открытии файла main.go. Нажмите "**Установить**";
12. После установки могут появиться другие уведомления о недостающих расширениях. Нажмите "**Install All**"; дождитесь окончания установки: All tools successfully installed. You are ready to Go. :)
13. Начните вводить следующий код в файл main.go:

package main

func main() {

fmt.Println("Hello, World!")

}

1. В процессе ввода кода в Visual Studio Code должны появиться подсказки для названий функций и импортов. Нажмите на "Println" в выпадающем списке, и в файл **автоматически** будет добавлена строка импорта:

import "fmt"

1. Включите **автосохранение** файлов: нажмите на шестерёнку в левом нижнем углу → выберите «Параметры» → во вкладке «Часто используемые», на пункте «Files: Auto Save» из выпадающего списка выберете «**afterDelay**».
2. Убедитесь, что вы находитесь в директории hello-world в терминале, и введите команду go run main.go;
3. Убедитесь, что в терминале выводится сообщение "**Hello, World!**"

## Синтаксис языка Go

Программа на Go начинается с **пакета** (**package**), который является основой для организации кода. Каждая программа должна содержать функцию main(), которая является **точкой входа** в программу. Внутри функции main() можно вызывать другие функции, создавать переменные, и т.д.

Пример структуры программы на Go:

package main

import (

"fmt"

"time"

)

func main() {

fmt.Println("Привет, мир!")

fmt.Println("Сегодня", time.Now().Format("02.01.2006"), "года.")

}

Очень важная деталь: в языке Go **нет** модификаторов области видимости **public** и **private** — вместо этого, чтобы сделать функцию, структуру или интерфейс **публичными**, они должны начинаться с большой буквы, а обращение к ним происходит через название пакета.

### Переменные и типы данных

// объявление переменной

var x int

x = 5

// объявление и инициализация переменной

var y int = 10

// короткое объявление переменной

z := 15

Go имеет следующие встроенные типы данных:

* bool - булев тип (true или false)
* int, int8, int16, int32, int64 - целочисленные типы со знаком
* uint, uint8, uint16, uint32, uint64 - целочисленные типы без знака
* float32, float64 - вещественные числа
* string - строковый тип
* byte - псевдоним для uint8
* rune - псевдоним для int32, используется для работы с символами Unicode

В Go доступны все стандартные арифметические операции:

// сложение

a + b

// вычитание

a - b

// умножение

a \* b

// деление

a / b

// остаток от деления

a % b

Массивы в Go имеют фиксированный размер и могут содержать элементы только одного типа данных. Объявление массива:

var arr [5]int

Для доступа к элементам массива используется индекс:

arr[0] = 1

arr[1] = 2

Добавление в массив происходит через функцию **append**:

arr = append(arr, 3)

### Циклы и условные конструкции

В Go доступны следующие типы циклов:

* **for** - цикл со счетчиком;
* **range** - цикл для итерации по элементам коллекции.

Пример использования цикла **for**:

for i := 0; i < 10; i++ {

if (i % 2 == 1) {

continue // переход к следующей итерации

}

// код цикла

}

for { // бесконечный цикл

// код цикла

if (true) {

break // выход из цикла

}

}

Пример использования цикла **range**:

arr := [3]int{1, 2, 3}

for index, value := range arr {

fmt.Println(arr[index], value) // Выведет одно и то же число

}

В Go есть две условные конструкции: **if** и **switch**.

Конструкция **if** позволяет выполнять блок кода, если условие истинно. Синтаксис **if** выглядит так:

if x > 0 {

fmt.Println("x больше нуля")

} else if x == 0 {

fmt.Println("x равен нулю")

} else {

fmt.Println("x меньше нуля")

}

Конструкция **switch** используется для выбора одной из нескольких альтернатив в зависимости от значения выражения. Синтаксис **switch** выглядит так:

switch x {

case 0:

fmt.Println("x равен нулю")

case 1:

fmt.Println("x равен одному")

default:

fmt.Println("x не равен нулю и не равен одному")

}

### Функции

Функции в Go - это ключевой элемент программы, который позволяет выполнять повторяющиеся операции с множеством входных параметров и возвращать результат.

Создание функции в Go начинается с ключевого слова **func**, за которым следует имя функции, список входных параметров и тип возвращаемого значения (если оно есть). Например:

func add(x int, y int) int {

return x + y

}

Функция *add* принимает два целочисленных аргумента *x* и *y*, складывает их и возвращает результат также в виде целого числа.

Можно также **объединять** несколько входных параметров с одним типом:

func add(x, y int) int {

return x + y

}

Также можно возвращать **несколько** значений из функции, просто указав их через запятую в списке возвращаемых значений. Например:

func swap(x, y string) (string, string) {

return y, x

}

В Go есть функции с **переменным** числом параметров. Такие функции объявляются с помощью **многоточия** ... перед типом параметра. Например, такая функция может выводить список аргументов на экран:

func printArgs(args ...string) {

// через \_ обозначается переменная, которая не будет

// использоваться в коде

for \_, arg := range args {

fmt.Println(arg)

}

}

Функции в Go могут быть определены в **любом** месте программы, в том числе внутри других функций, и могут быть переданы в качестве **аргументов** другим функциям и **возвращены** из функций в качестве значений.

Также в Go есть **анонимные** функции, которые могут быть определены и вызваны без имени. Они часто используются для создания замыканий (closures), которые позволяют сохранять состояние между вызовами функции.

func main() {

// Определяем анонимную функцию и вызываем ее сразу после определения

func() {

fmt.Println("Hello, anonymous function!")

}()

// Сохраняем анонимную функцию в переменной и вызываем ее позже

f := func() {

fmt.Println("Hello, saved anonymous function!")

}

f()

}

В Go есть также возможность использовать функции как значения и передавать их в качестве аргументов другим функциям или возвращать из функций в качестве значений. Например, функция *apply* принимает функцию и аргумент и вызывает эту функцию с переданным аргументом:

func apply(f func(int) int, x int) int {

return f(x)

}

func double(x int) int {

return x \* 2

}

func main() {

fmt.Println(apply(double,2)) // Выведет 4

}

### Указатели

**Указатель** (pointer) в языке программирования – это переменная, которая содержит **адрес** ячейки памяти, где хранится другая переменная. Указатели используются для передачи значений и изменения значений переменных **по ссылке**, а не по значению.

В Go указатели используются для работы с данными **напрямую** в памяти. Они позволяют создавать более **эффективный** код, а также обеспечивают возможность изменять значения переменных, которые были переданы в функцию, **не копируя** их.

Для получения адреса переменной в Go используется символ "&". Например, чтобы получить адрес переменной *x*, нужно написать &x.

Для объявления указателя используется символ "\*". Например, var p \*int. Это означает, что переменная *p* содержит адрес ячейки памяти, где хранится значение типа **int**.

Чтобы получить значение, на которое ссылается указатель, используется символ "\*". Например, чтобы получить значение, на которое ссылается указатель p, нужно написать \*p.

Вот пример применения указателей в Go:

package main

import "fmt"

func main() {

x := 10

fmt.Println("Значение x =", x)

p := &x

fmt.Println("Значение указателя p =", p)

fmt.Println("Значение, на которое ссылается указатель p =", \*p)

\*p = 20

fmt.Println("Значение x после изменения через указатель =", x)

}

В этом примере мы создали переменную *x* и присвоили ей значение 10. Затем мы создали указатель *p* и присвоили ему адрес переменной *x*. Мы напечатали значение переменной *x*, адрес переменной *x* и значение, на которое ссылается указатель *p*.

Затем мы изменили значение, на которое ссылается указатель *p*, на 20, и напечатали значение переменной *x*, чтобы показать, что значение изменилось.

Вывод программы будет следующим:

Значение x = 10

Значение указателя p = 0x1040a124

Значение, на которое ссылается указатель p = 10

Значение x после изменения через указатель = 20

Как видите, мы смогли изменить значение переменной x, используя указатель. Это очень полезно, когда нужно передать переменную в функцию и изменить ее значение внутри функции.

### Структуры, интерфейсы, ООП

**Структуры в Go**

Структуры в Go - это типы данных, которые позволяют **объединять** несколько разнородных полей в **единый** объект. В Go структуры могут иметь **методы**, что делает их похожими на классы в других языках программирования.

Пример объявления структуры в Go:

type Person struct {

name string

age int

}

В данном примере определяется структура *Person*, которая имеет два поля: *name* и *age*.

Структуры могут использоваться для передачи нескольких значений между функциями, а также для хранения данных внутри структуры.

**Интерфейсы в Go**

Интерфейсы в Go позволяют определить **набор методов**, которые должны быть реализованы для конкретного типа. Интерфейсы помогают абстрагироваться от конкретной реализации и сосредоточиться на функциональности.

Пример определения интерфейса в Go:

type Shape interface {

Area() float64

Perimeter() float64

}

В данном примере определяется интерфейс *Shape*, который определяет два метода: *Area* и *Perimeter*. Любой тип, который реализует эти два метода, будет считаться реализацией интерфейса *Shape*.

Вот, как это выглядит в коде:

type Square struct {

Side float64

}

func (s Square) Area() float64 {

return s.Side \* s.Side

}

func (s Square) Perimeter() float64 {

return s.Side \* 4

}

**Принципы ООП в Go**

Go **не является** чистым объектно-ориентированным языком, но многие принципы ООП могут быть применены в Go.

Например, в Go есть возможность определять методы на структурах, что позволяет реализовывать **инкапсуляцию**. Также в Go можно использовать интерфейсы для **полиморфизма** и абстрагирования.

Однако в Go **нет классов и наследования**, что делает его отличным от классических ООП-языков, таких как Java или C#. Вместо этого в Go используется **композиция** и **встраивание** (embedding), что позволяет создавать более гибкие и модульные программы.

## Отладка кода

**Отладка кода** – это процесс поиска и исправления ошибок в коде программы. Она позволяет разработчикам проверить код на наличие ошибок и устранить их. Чтобы запустить режим отладки, требуется создать файл в папке **/.vscode/launch.json**, в котором будут прописаны варианта запуска программ в проекте:

    {

        "version": "0.2.0",

        "configurations": [

            {

                "name": "Launch Product Front",

                "type": "go",

                "request": "launch",

                "mode":"auto",

                "program":".../go/src/github.com/joyzem/documents/services/account/frontend/cmd/"

            },

            {

                "name": "Launch Account Back",

                "type": "go",

                "request": "launch",

                "mode":"auto",

                "program":".../go/src/github.com/joyzem/documents/services/account/backend/cmd/"

            },

// ...

Запустить режим отладки можно с помощью клавиши F5.

Установите точки останова (breakpoints) в местах кода, где вы хотите остановить выполнение программы. Для этого нажмите на левой панели редактора на нужной строке кода и появится красный кружок.

Когда выполнение программы доходит до точки останова, выполнение приостанавливается. В панели отладки можно просмотреть текущие значения переменных, стек вызовов функций и другую информацию о состоянии программы.

Используйте кнопки в панели отладки (шаг вперед, шаг назад, продолжить выполнение, перезапустить и т.д.) для управления выполнением программы. Нажимайте кнопку "Шаг вперед" (или F10 на клавиатуре), чтобы выполнить следующую строку кода.

## Самостоятельное задание

**Для вариантов 1-13.**

Разработать приложение для работы с геометрическими фигурами.

Требуется описать интерфейс Shape со следующими методами:

* IsShapeRight() boolean
* CalculateArea() float
* CalculatePerimeter() int

В качестве фигуры, реализовывающей интерфейс, выберите из списка согласно своему варианту:

* 1. Кольцо
  2. Круг
  3. Овал
  4. Ромб
  5. Трапеция
  6. Квадрат
  7. Параллелограмм
  8. Прямоугольник
  9. Равнобедренный треугольник
  10. Прямоугольный треугольник
  11. Произвольный треугольник
  12. Равносторонний шестиугольник
  13. Полукруг

Пример работы приложения:

Пользователь запускает приложение. В терминале сообщение:

Здравствуйте!

Вам доступны следующие команды:

1 — Создать фигуру "Треугольник";

2 — Выйти.

При нажатии клавиши "2" приложение заканчивает работу.

При нажатии клавиши "1" терминал выдает следующее сообщение:

Задайте сторону a: <<Ввод пользователя>>

Задайте сторону b: <<Ввод пользователя>>

Задайте сторону c: <<Ввод пользователя>>

Если сумма двух сторон меньше третьей, то вывести сообщение, что данные некорректны, иначе:

1 — Подсчитать площадь; <<Примерный вывод: S = кв.корень(p \* (p - a) \* (p - b) \* (p - c)) = Ответ>>

2 — Подсчитать периметр; <<Примерный вывод: P = a + b + c = Ответ>>

3 — Проверить правильность фигуры; <<Правильная или Неправильная>>

4 — Построить новую фигуру;

5 — Выйти.

**Для вариантов 14-30.**

Есть функция *Compare*:

// Сравнивает два объекта, реализующих интерфейс Comparable и возвращает результат и отчет

// Результатом является 1, если первый объект "больше", 0, если равны, и -1, если первый объект "меньше"

// Отчет — Объединение методов Print() у двух экземпляров интерфейса Comparable

func Compare(first Comparable, second Comparable) (int, string)

Она сравнивает два объекта, реализующих интерфейс Comparable, который обладает двумя методами:

* Value() float — Значение для сравнения
* Formula() string — Формула, по которой высчитывается Value() с выводом значений всех полей структуры

Требуется:

* определить подходящее для сравнения свойство у объектов согласно варианту;
* написать реализацию функции Compare;
* написать структуры, реализующие интерфейс Comparable, согласно варианту и сравнить их:
  1. Треугольник и Квадрат
  2. Пицца и Шашлык
  3. Градусы Цельсий и Градусы Кельвин
  4. Дом и Квартира
  5. Фильм и Книга
  6. Рубли и Копейки
  7. Видеокарта и Процессор
  8. Машина и Мотоцикл
  9. Монитор и Телефон
  10. Гитарист и Вокалист
  11. Килограммы и Граммы
  12. Студент и Волонтер
  13. Футболист и Велосипедист
  14. Сантиметры и Метры
  15. Поезд и Самолет
  16. Бриллиант и Золото
  17. Отжимания и Подтягивания

# Лабораторная работа № 3. Создание Backend-микросервиса.

## Цель

Разработать backend-часть микросервиса товаров.

## Определение структуры микросервиса

* 1. Перейдите в C:\Program Files\Go\src\github.com\<YOUR\_LOGIN>;
  2. Создайте папку documents;
  3. Откройте эту папку в Visual Studio Code от имени администратора;
  4. Создайте терминал и введите команду: go mod init github.com/<YOUR\_LOGIN>/documents
  5. Создадим структуру папок и рассмотрим назначение каждой из них:

services // Здесь распологаются микросервисы

├───base // Общий код и утилиты

└───product // Каждый сервис в отдельной папке

├───backend // программа бэкенда

│ ├───cmd // входная точка в программу

│ ├───implementation // реализация интерфейсов

│ ├───repo // интерфейс репозитория

│ ├───service // интерфейс сервиса

│ └───transport // транспортный уровень

│ └───http // реализация http запросов

├───domain // entity

├───dto // data transfer objects

└───frontend // Программа фронтэнда

go.mod

Структура папок в проекте на Go может отличаться в зависимости от конкретных требований проекта и используемых фреймворков

* services – папка с микросервисами, каждый сервис должен находиться в отдельной папке;
* base – папка с общим кодом, утилитами, константами, функциями и т.д.;
* product – микросервис товаров, представленный в виде двух программ для backend’а и frontend’а;
* domain – папка с описанием сущностей, которые используются в сервисе;
* dto – папка с описанием структур для передачи JSON объектов;
* frontend – клиентская часть сервиса (в следующих лабораторных работах);
* backend – серверная часть микросервиса:
  + cmd – папка с входной точкой в программу, откуда запускается серверная часть
  + implementation – папка с реализацией интерфейса сервиса и репозиториев
  + repo – папка с описанием интерфейса репозитория (доступ к базе данных)
  + service – папка с описанием интерфейса сервиса
  + transport – папка с реализацией транспортного уровня (маршрутизатор)
    - http – папка с реализацией маршрутизатора

## Чистая архитектура

Данный подход к директориям и пакетам обусловлен использованием «Чистой архитектуры».

Главным принципом чистой архитектуры является разделение приложения на несколько слоев, каждый из которых отвечает за конкретную задачу, и уровень каждого слоя имеет свой набор абстракций и интерфейсов, что позволяет каждому слою быть независимым от других слоев и изменяться независимо. Чистая архитектура способствует уменьшению сложности проекта, что делает его более легким для разработки и поддержки.

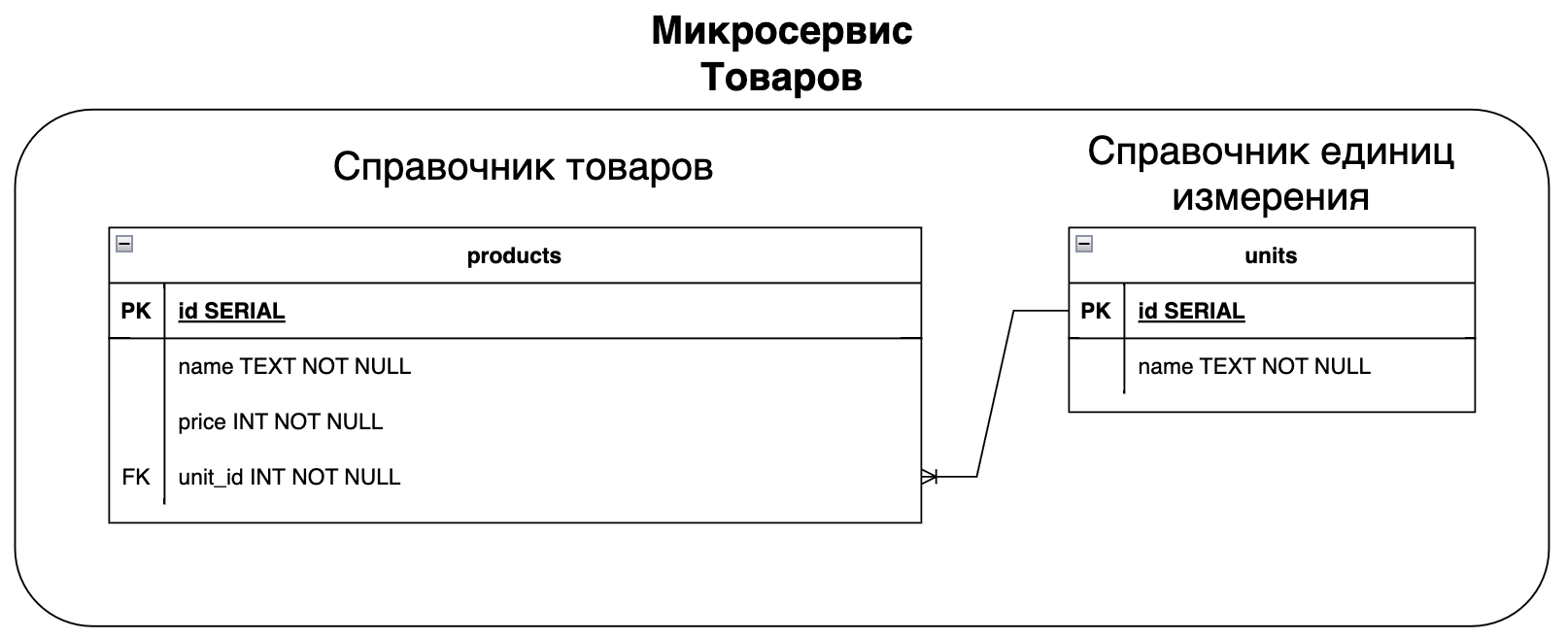
Важно также помнить, что чистая архитектура – это не стандарт или рамки, а концепция, которую можно адаптировать под конкретные нужды проекта. Это основа коммерческой разработки, без которой не обходится ни один проект.

Любая архитектура привносит избыточность. Эта избыточность нужна для единообразия, а единообразие нужно для того, чтобы код был очевидным.

## Поэтапное написание серверной части

### Изучение схемы данных

Первым делом необходимо изучить схему данных из предыдущей работы и в соответствии с ней реализовать сущности из пакета domain.



### Реализация domain слоя

В папке domain будут находиться сущности для сервиса. Создайте два файла: **product.go** и **unit.go**

Откройте файл **product.go** и напишите следующий код:

package domain

type Product struct {

Id int `json:"id"`

Name string `json:"name"`

Price int `json:"price"`

UnitId int `json:"unit\_id"`

}

Теги в обратных кавычках «`» позволяют указать наименование поля при его конвертации в JSON.

Аналогично, код для «**unit.go»**:

package domain

type Unit struct {

Id int `json:"id"`

Name string `json:"name"`

}

При нажатии Ctrl + S установленное расширение автоматически отформатирует файл.

### Интерфейс сервиса

Создайте в папке /services/product/backend/service файл **service.go**.

В дальнейших вставках кода заменяйте GitHub логин на свой.

Напишите этот код:

package service

import "github.com/joyzem/documents/services/product/domain"

// Service определяет интерфейс, который предоставляет методы для работы с товарами и единицами измерения.

// - CreateProduct создает новый товар с именем "name", ценой "price" и единицей измерения с id "unitId".

// - GetProducts возвращает список всех товаров.

// - UpdateProduct обновляет информацию о товаре.

// - DeleteProduct удаляет товар из базы данных по id.

// - CreateUnit создает новую единицу измерения с именем "name".

// - GetUnits возвращает список всех единиц измерения.

// - UpdateUnit обновляет информацию о единице измерения.

// - DeleteUnit удаляет единицу измерения из базы данных по id.

type ProductService interface {

CreateProduct(name string, price int, unitId int) (\*domain.Product, error)

GetProducts() ([]domain.Product, error)

UpdateProduct(domain.Product) (\*domain.Product, error)

DeleteProduct(int) error

ProductById(int) (\*domain.Product, error)

CreateUnit(string) (\*domain.Unit, error)

GetUnits() ([]domain.Unit, error)

UpdateUnit(domain.Unit) (\*domain.Unit, error)

DeleteUnit(int) error

UnitById(int) (\*domain.Unit, error)

}

Как было сказано ранее, функции могут возвращать несколько значений. Так, функция создания товара в результате вернет два значения: **ссылку** на товар и **ошибку**. Дело в том, что если никакой ошибки не произошло, то второе значение будет nil (то же, что и null). Рассмотрим на примере:

product, err := productService.CreateProduct(

productName,

productPrice,

unitId,

)

if err != nil {

// Произошла ошибка во время создания товара

fmt.Println("Ошибка: ", err.Error())

} else {

// Ошибки нет

}

В языке программирования Go **нет** оператора **try-catch**, так как его разработчики предпочли использовать другую концепцию для обработки ошибок, которая называется "**проверка ошибок**" (error checking).

В Go функции, которые могут вернуть ошибку, возвращают дополнительное значение типа **error**, содержащее описание ошибки (если таковая произошла) или **nil**, если ошибки не произошло. После вызова такой функции, программист может проверить, произошла ли ошибка, сравнивая значение error с nil. Если значение не равно nil, то это означает, что произошла ошибка, и ее нужно обработать.

Такой подход позволяет более явно и ясно **контролировать** ошибки и обрабатывать их на нужном уровне.

### Data transfer objects

**DTO (Data Transfer Object)** - это структуры, которые используются для **передачи** данных между разными компонентами системы.

В файлах этого пакета содержатся структуры для запросов и ответов, которые относятся к микросервису товаров. DTO используются для определения формата передаваемых данных и для облегчения процесса маршалинга (кодирования) и демаршалинга.

В тегах json указывается, как поля должны называться в **JSON**-формате. Тег "**omitempty**" говорит, что поле может быть опущено, если оно пустое. Если есть необходимость возвращать ошибки, следует определить поле "**Err**" типа string.

Создайте файл **/services/product/dto/product.go**

Напишите следующий код:

package dto

import (

"github.com/joyzem/documents/services/product/domain"

)

// Если запрос выполнен успешно, то придет ответ без поля “error”

// иначе — только с полем "error"

type (

CreateProductRequest struct {

Name string `json:"name"`

Price int `json:"price"`

UnitId int `json:"unit\_id"`

}

CreateProductResponse struct {

Product \*domain.Product `json:"product,omitempty"`

Err string `json:"error,omitempty"`

}

GetProductsRequest struct {

}

GetProductsResponse struct {

Products []domain.Product `json:"products,omitempty"`

Err string `json:"error,omitempty"`

}

UpdateProductRequest struct {

Product domain.Product `json:"product"`

}

UpdateProductResponse struct {

Product \*domain.Product `json:"product,omitempty"`

Err string `json:"error,omitempty"`

}

DeleteProductRequest struct {

Id int `json:"id"`

}

DeleteProductResponse struct {

Err string `json:"error,omitempty"`

}

ProductByIdRequest struct {

Id int `json:"id"`

}

ProductByIdResponse struct {

Product \*domain.Product `json:"product,omitempty"`

Err string `json:"error,omitempty"`

}

)

На каждую операцию CRUD (create, read, update, delete) есть структура для запроса и для ответа.

Аналогично для **/services/product/dto/unit.go**:

package dto

import "github.com/joyzem/documents/services/product/domain"

// Если запрос выполнен успешно, то придет ответ без поля "error"

// иначе — только с полем "error"

type (

CreateUnitRequest struct {

Unit string `json:"unit"`

}

CreateUnitResponse struct {

Unit \*domain.Unit `json:"unit,omitempty"`

Err string `json:"error,omitempty"`

}

GetUnitsRequest struct {

}

GetUnitsResponse struct {

Units []domain.Unit `json:"units,omitempty"`

Err string `json:"error,omitempty"`

}

UpdateUnitRequest struct {

Unit domain.Unit `json:"unit"`

}

UpdateUnitResponse struct {

Unit \*domain.Unit `json:"unit,omitempty"`

Err string `json:"error,omitempty"`

}

DeleteUnitRequest struct {

Id int `json:"id"`

}

DeleteUnitResponse struct {

Err string `json:"error,omitempty"`

}

UnitByIdRequest struct {

Id int `json:"id"`

}

UnitByIdResponse struct {

Unit \*domain.Unit `json:"unit,omitempty"`

Err string `json:"error,omitempty"`

}

)

### Репозитории

Создайте файл **/services/product/backend/repo/product.go**

package repo

import (

"github.com/joyzem/documents/services/product/domain"

)

type ProductRepo interface {

CreateProduct(domain.Product) (\*domain.Product, error)

GetProducts() ([]domain.Product, error)

UpdateProduct(domain.Product) (\*domain.Product, error)

DeleteProduct(int) error

ProductById(int) (\*domain.Product, error)

}

Аналогично **/services/product/backend/repo/unit.go**

package repo

import (

"github.com/joyzem/documents/services/product/domain"

)

type UnitRepo interface {

CreateUnit(domain.Unit) (\*domain.Unit, error)

GetUnits() ([]domain.Unit, error)

UpdateUnit(domain.Unit) (\*domain.Unit, error)

DeleteUnit(int) error

UnitById(int) (\*domain.Unit, error)

}

После написания интерфейсов необходимо написать реализацию.

Создайте файл **/services/product/backend/implementation  
/product-repo.go**

package implementation

import (

"database/sql"

"github.com/joyzem/documents/services/product/backend/repo"

"github.com/joyzem/documents/services/product/domain"

)

// Реализация репозитория товаров

type repository struct {

db \*sql.DB

}

// Возвращает репозиторий товаров

func NewProductRepo(db \*sql.DB) repo.ProductRepo {

return &repository{

db: db,

}

}

Теперь необходимо реализовать методы интерфейса **ProductRepo**:

**Создание товара**

// Добавляет новый товар

func (r \*repository) CreateProduct(p domain.Product) (\*domain.Product, error) {

// SQL-запрос для добавления товара с возвратом id

sql := `

INSERT INTO products (name, price, unit\_id)

VALUES ($1, $2, $3) RETURNING id

`

// Подготовка запроса

result, err := r.db.Prepare(sql)

if err != nil {

// Возврат nil и ошибки, если запрос не может быть выполнен

return nil, err

}

// Освобождение ресурсов после выполнения функции

defer result.Close()

// Выполнение запроса и сканирование результата (Id созданного товара)

var insertedId int

if err := result.QueryRow(p.Name, p.Price, p.UnitId).Scan(&insertedId); err != nil {

// объявление и проверка ошибки внутри блока if

return nil, err

}

// Возврат нового товара

return r.ProductById(insertedId)

}

**Получение товаров**

// GetProducts выполняет запрос в базу данных для получения списка всех товаров.

// Возвращает массив типа domain.Product и ошибку, если она произошла.

func (r \*repository) GetProducts() ([]domain.Product, error) {

// SQL-запрос для получения информации о всех товарах

sql := `

SELECT \* FROM products ORDER BY products.name ASC

`

// Выполнение запроса

rows, err := r.db.Query(sql)

if err != nil {

// Возврат пустого списка и ошибки, если запрос не может быть выполнен

return nil, err

}

// Освобождение ресурсов после выполнения функции

defer rows.Close()

// Создание пустого списка товаров

products := []domain.Product{}

// Перебор всех строк в ответе на запрос

for rows.Next() {

// Создание пустой структуры товара

product := domain.Product{}

// Заполнение структуры данными из текущей строки

if err := rows.Scan(&product.Id, &product.Name, &product.Price, &product.UnitId); err != nil {

return nil, err

}

// Добавление товара в список товаров

products = append(products, product)

}

// Возврат товаров

return products, nil

}

**Обновление**

// Обновляет информацию о товаре в базе данных.

// Принимает параметр p типа domain.Product, который содержит новые данные для товара.

func (r \*repository) UpdateProduct(p domain.Product) (\*domain.Product, error) {

// SQL-запрос для обновления цены, единицы измерения и имени товара

sql := `

UPDATE products SET price = $1, unit\_id = $2, name = $3 WHERE id = $4

`

// Выполнение запроса

rows, err := r.db.Query(sql, p.Price, p.UnitId, p.Name, p.Id)

if err != nil {

return nil, err

}

// Закрыть соединение, чтобы осуществить следующий запрос

rows.Close()

// Вернуть обновленный товар

return r.ProductById(p.Id)

}

**Удаление товара**

// DeleteProduct удаляет товар из базы данных по его ID.

// Id товара передается в качестве параметра.

func (r \*repository) DeleteProduct(id int) error {

sql := `

DELETE FROM products WHERE id = $1

`

// через \_ обозначается переменная, которая не будет

// использоваться в коде

\_, err := r.db.Exec(sql, id)

return err

}

**Получение по id**

func (r \*repository) ProductById(id int) (\*domain.Product, error) {

// SQL-запрос для получения данных товара

getProductSql := `

SELECT \* FROM products WHERE products.id = $1

`

// Инициализация переменной

product := domain.Product{}

// Выполнение запроса и сканирование результата (данные товара)

if err := r.db.QueryRow(getProductSql, id).Scan(

&product.Id,

&product.Name,

&product.Price,

&product.UnitId); err != nil {

return nil, err

}

// Возврат указателя на товар

return &product, nil

}

Реализация для UnitRepo: **/services/product/backend/implementation  
/unit-repo.go**

package implementation

import (

"database/sql"

"errors"

"sort"

"github.com/joyzem/documents/services/product/backend/repo"

"github.com/joyzem/documents/services/product/domain"

)

type unitRepository struct {

db \*sql.DB

}

func NewUnitRepository(db \*sql.DB) repo.UnitRepo {

return &unitRepository{

db: db,

}

}

func (r \*unitRepository) CreateUnit(unit domain.Unit) (\*domain.Unit, error) {

sql := `

INSERT INTO units (name) VALUES ($1) RETURNING id

`

result, err := r.db.Prepare(sql)

if err != nil {

return nil, err

}

defer result.Close()

var insertedId int

if err := result.QueryRow(unit.Name).Scan(&insertedId); err != nil {

return nil, err

}

return r.UnitById(insertedId)

}

func (r \*unitRepository) GetUnits() ([]domain.Unit, error) {

sql := `SELECT \* FROM units`

rows, err := r.db.Query(sql)

if err != nil {

return nil, err

}

defer rows.Close()

units := []domain.Unit{}

for rows.Next() {

unit := domain.Unit{}

rows.Scan(&unit.Id, &unit.Name)

units = append(units, unit)

}

// Сортировка по id

sort.Slice(units, func(i, j int) bool {

return units[i].Id < units[j].Id

})

return units, nil

}

func (r \*unitRepository) UpdateUnit(unit domain.Unit) (\*domain.Unit, error) {

sql := `

UPDATE units SET name = $1 WHERE id = $2

`

rows, err := r.db.Query(sql, unit.Name, unit.Id)

if err != nil {

return nil, err

}

rows.Close()

return r.UnitById(unit.Id)

}

func (r \*unitRepository) DeleteUnit(id int) error {

if id == 0 {

return errors.New("can't delete default value")

}

sql := `

DELETE FROM units WHERE id = $1

`

\_, err := r.db.Exec(sql, id)

return err

}

func (r \*unitRepository) UnitById(id int) (\*domain.Unit, error) {

sql := `

SELECT \* FROM units WHERE id = $1

`

unit := domain.Unit{}

if err := r.db.QueryRow(sql, id).Scan(&unit.Id, &unit.Name); err != nil {

return nil, err

}

return &unit, nil

}

### Реализация сервиса

Для удобного логирования возможных ошибок создайте директорию **/services/base**, а в ней файл **log.go**

Укажите пакет **base** и добавьте в этот файл функцию **LogError**, которая принимает на вход аргумент типа error, проверяет его на nil, и если аргумент не nil, то печатает его с помощью модуля “fmt”.

Создайте файл **/services/product/backend/implementation/service.go** и добавьте в него следующий код:

    package implementation

    import (

        "github.com/joyzem/documents/services/base"

        "github.com/joyzem/documents/services/product/backend/repo"

        "github.com/joyzem/documents/services/product/backend/service"

        "github.com/joyzem/documents/services/product/domain"

    )

    // Реализация сервиса

    type productService struct {

        productRepo repo.ProductRepo

        unitRepo    repo.UnitRepo

    }

    // Возвращает реализацию сервиса

    func NewService(productRepo repo.ProductRepo, unitRepo repo.UnitRepo) service.ProductService {

        return &productService{

            productRepo: productRepo,

            unitRepo:    unitRepo,

        }

    }

    // CreateProduct - создает новый товар с переданными параметрами (name, price, unitId)

    func (s \*productService) CreateProduct(name string, price int, unitId int) (\*domain.Product, error) {

        p := domain.Product{Name: name, Price: price, UnitId: unitId}

        createdProduct, err := s.productRepo.CreateProduct(p)

        base.LogError(err)

        return createdProduct, err

    }

    // GetProducts - возвращает все товары

    func (s \*productService) GetProducts() ([]domain.Product, error) {

        products, err := s.productRepo.GetProducts()

        base.LogError(err)

        return products, err

    }

    // UpdateProduct - обновляет информацию о товаре

    func (s \*productService) UpdateProduct(newProduct domain.Product) (\*domain.Product, error) {

        updatedProduct, err := s.productRepo.UpdateProduct(newProduct)

        base.LogError(err)

        return updatedProduct, err

    }

    // DeleteProduct - удаляет товар по id

    func (s \*productService) DeleteProduct(id int) error {

        err := s.productRepo.DeleteProduct(id)

        base.LogError(err)

        return err

    }

    // ProductById - возвращает товар по идентификатору

    func (s \*productService) ProductById(id int) (\*domain.Product, error) {

        product, err := s.productRepo.ProductById(id)

        base.LogError(err)

        return product, err

    }

    // CreateUnit - добавляет единицу измерения.

    func (s \*productService) CreateUnit(name string) (\*domain.Unit, error) {

        unit := domain.Unit{Name: name}

        createdUnit, err := s.unitRepo.CreateUnit(unit)

        base.LogError(err)

        return createdUnit, err

    }

    // GetUnits - возвращает список всех единиц измерения.

    func (s \*productService) GetUnits() ([]domain.Unit, error) {

        units, err := s.unitRepo.GetUnits()

        base.LogError(err)

        return units, err

    }

    // UpdateUnit - обновляет информацию об единице измерения.

    func (s \*productService) UpdateUnit(unit domain.Unit) (\*domain.Unit, error) {

        updatedUnit, err := s.unitRepo.UpdateUnit(unit)

        base.LogError(err)

        return updatedUnit, err

    }

    // DeleteUnit - удаляет единицу измерения.

    func (s \*productService) DeleteUnit(id int) error {

        err := s.unitRepo.DeleteUnit(id)

        base.LogError(err)

        return err

    }

    // UnitById - возвращает единицу измерения по идентификатору

    func (s \*productService) UnitById(id int) (\*domain.Unit, error) {

        unit, err := s.unitRepo.UnitById(id)

        base.LogError(err)

        return unit, err

    }

По сути, реализация сервиса обращается к требуемому репозиторию, получает данные и логирует ошибки, если те появляются. Зачем тогда добавлять этот слой? Бывает, что у приложения есть потребность в облачном и локальном хранении, проверке данных, различных бизнес-сценариях, расчетах, тогда реализация сервиса не будет заключаться в простом обращении к репозиторию – логика кода становится гораздо сложнее. Возвращаясь к теме чистой архитектуры, мы сохраняем разделение слоев, тем самым обеспечивая масштабируемость, читабельность и поддержку кода.

### Эндпоинты

**Go-Kit** является набором библиотек, предназначенных для создания микросервисов на Go. Go-Kit реализует общие паттерны проектирования, такие как паттерн **Endpoint**, который позволяет определить общий интерфейс для микросервисов.

**Endpoint** – это абстракция, представляющая точку входа для взаимодействия с микросервисом. Он представляет собой **функцию**, которая принимает контекст запроса и входные данные и возвращает выходные данные и ошибку. Endpoint отвечает за обработку запроса и передачу его на следующий этап обработки.

Использование Go-Kit и паттерна Endpoint **упрощает** процесс разработки микросервисов, обеспечивает **гибкость** и **масштабируемость** приложения, а также **упрощает** поддержку кода.

Чтобы установить Go-Kit, откройте терминал и введите: go get github.com/go-kit/kit

Создайте файл **/services/product/backend/transport/endpoints.go**

package transport

import (

    "context"

    "github.com/go-kit/kit/endpoint"

    "github.com/joyzem/documents/services/product/backend/service"

    "github.com/joyzem/documents/services/product/dto"

)

// Структура, которая хранит в себе все эндпоинты, используемые в микросервисе.

type Endpoints struct {

    CreateProduct endpoint.Endpoint

    GetProducts   endpoint.Endpoint

    UpdateProduct endpoint.Endpoint

    DeleteProduct endpoint.Endpoint

    ProductById   endpoint.Endpoint

    CreateUnit    endpoint.Endpoint

    GetUnits      endpoint.Endpoint

    UpdateUnit    endpoint.Endpoint

    DeleteUnit    endpoint.Endpoint

    UnitById      endpoint.Endpoint

}

    // Возвращает набор эндпоинтов, используя сервис `s`.

    // Для создания каждого эндпоинта используется соответствующая

    // функция (например, `makeCreateProductEndpoint` для создания

    // эндпоинта `CreateProduct`)

    func MakeEndpoints(s service.ProductService) Endpoints {

        return Endpoints{

            CreateProduct: makeCreateProductEndpoint(s),

            GetProducts:   makeGetProductsEndpoint(s),

            UpdateProduct: makeUpdateProductEndpoint(s),

            DeleteProduct: makeDeleteProductEndpoint(s),

            ProductById:   makeProductByIdEndpoint(s),

            CreateUnit:    makeCreateUnitEndpoint(s),

            GetUnits:      makeGetUnitsEndpoint(s),

            UpdateUnit:    makeUpdateUnitEndpoint(s),

            DeleteUnit:    makeDeleteUnitEndpoint(s),

            UnitById:      makeUnitByIdEndpoint(s),

        }

    }

    // Возвращает endpoint, который создает новый продукт с помощью сервиса `s`

    func makeCreateProductEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(ctx context.Context, request interface{}) (interface{}, error) {

            // Приводим запрос к типу `dto.CreateProductRequest`

            req := request.(dto.CreateProductRequest)

            // Создаем новый продукт с помощью сервиса

            product, err := s.CreateProduct(req.Name, req.Price, req.UnitId)

            // Возвращаем ответ в виде `dto.CreateProductResponse` с полем `Product`

            return dto.CreateProductResponse{Product: product}, err

        }

    }

Так как интерфейс **endpoint.Endpoint** описывает одну функцию, мы и возвращаем анонимную функцию, реализовывающую интерфейс. На входе у функции **ctx** типа **context.Context** и **request** типа **interface{}**

Использование **ctx** позволяет эффективно управлять ресурсами, такими как сетевые подключения, блокировки и т.д., и предотвращает утечки ресурсов. Контексты также могут быть использованы для передачи данных между различными слоями приложения, такими как HTTP-обработчики, сервисы и репозитории. В рамках методички, чтобы не было усложнение кода, **ctx** не будет использоваться.

Аргумент **request** означает структуру, реализующую пустой интерфейс, то есть любую структуру.

Далее по коду происходит приведение **request** к типу **dto.CreateProductRequest** с помощью конструкции: req := request.(dto.CreateProductRequest)

Затем создаем продукт, используя сервис **ProductService**, и возвращаем результат в виде **dto.CreateProductResponse{Product: product}** и вторым значением **err**.

Если вы помните, в структуре ответа **dto.CreateProductResponse** было поле **Err**, куда нужно присваивать значение ошибки. На самом деле, это происходит, но в другом месте. Библиотека go-kit «под капотом» обрабатывает значения, которые возвращает **endpoint.Endpoint**, и, если **err != nil**, происходит вызов функции, которая в качестве ответа от сервера формирует сообщение об ошибке без других полей. Поэтому в **dto** пакете одним из тегов JSON был тег ‘omitempty’, который позволяет опускать поля, если они равны nil, соответственно, пользователю придет ответ с одним JSON полем **error**.

Остальные функции, возвращающие эндпоинты:

    func makeGetProductsEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            products, err := s.GetProducts()

            return dto.GetProductsResponse{Products: products}, err

        }

    }

    func makeUpdateProductEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.UpdateProductRequest)

            product, err := s.UpdateProduct(req.Product)

            return dto.UpdateProductResponse{Product: product}, err

        }

    }

    func makeDeleteProductEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.DeleteProductRequest)

            err := s.DeleteProduct(req.Id)

            return dto.DeleteProductResponse{}, err

        }

    }

    func makeProductByIdEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.ProductByIdRequest)

            product, err := s.ProductById(req.Id)

            return dto.ProductByIdResponse{Product: product}, err

        }

    }

    func makeDeleteUnitEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.DeleteUnitRequest)

            err := s.DeleteUnit(req.Id)

            return dto.DeleteUnitResponse{}, err

        }

    }

    func makeUpdateUnitEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.UpdateUnitRequest)

            unit, err := s.UpdateUnit(req.Unit)

            return dto.UpdateUnitResponse{Unit: unit}, err

        }

    }

    func makeGetUnitsEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(ctx context.Context, request interface{}) (interface{}, error) {

            units, err := s.GetUnits()

            return dto.GetUnitsResponse{

                Units: units,

            }, err

        }

    }

    func makeCreateUnitEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.CreateUnitRequest)

            unit, err := s.CreateUnit(req.Unit)

            return dto.CreateUnitResponse{Unit: unit}, err

        }

    }

    func makeUnitByIdEndpoint(s service.ProductService) endpoint.Endpoint {

        return func(\_ context.Context, request interface{}) (interface{}, error) {

            req := request.(dto.UnitByIdRequest)

            unit, err := s.UnitById(req.Id)

            return dto.UnitByIdResponse{Unit: unit}, err

        }

    }

### Маршрутизатор

Маршрутизатор (router) — это компонент, который определяет, какой обработчик (handler) будет вызываться для каждого запроса, поступающего на определенный маршрут. Для написания маршрутизатора в Go часто используются библиотеки, такие как Gorilla Mux или Chi. В данном примере используется библиотека Gorilla Mux.

* Для установки введите в терминале: go get github.com/gorilla/mux
* Создайте файл **/services/product/backend/transport/http/service.go**
* Введите в терминале go get github.com/go-kit/kit/transport/http

Перед тем, как начать писать код для маршрутизатора, необходимо написать функции, кодирующие ответы, ошибки и декодирующие запросы. Так как этот функционал общий для всех сервисов, его нужно реализовать в пакете **base**.

* Создайте файл **/services/base/responses-requests.go**;
* Добавьте в него следующий код:

    package base

    import (

        "context"

        "encoding/json"

        "fmt"

        "io/ioutil"

        "net/http"

        "reflect"

    )

    // Функция декодирует тело запроса и возвращает ошибку, если тело

    // запроса не соответствует структуре

    func DecodeBody(r \*http.Request, dest interface{}) error {

        // Ожидаемые поля

        expected := map[string]struct{}{}

        elem := reflect.ValueOf(dest).Elem()

        for i := 0; i < elem.NumField(); i++ {

            expected[string(elem.Type().Field(i).Tag.Get("json"))] = struct{}{}

        }

        // Создание копии тела запроса, так как r.Body может вызываться только один раз

        body, err := ioutil.ReadAll(r.Body)

        if err != nil {

            return err

        }

        // Декодировать тело запроса, чтобы проверить корректность

        var jsonKeys map[string]interface{}

        if err := json.Unmarshal(body, &jsonKeys); err != nil {

            return err

        }

        // Проверить все ожидаемые поля

        for key := range expected {

            if \_, ok := jsonKeys[key]; !ok {

                return fmt.Errorf("missing field: %s", key)

            }

        }

        // Проверить лишние поля

        for key := range jsonKeys {

            if \_, ok := expected[key]; !ok {

                return fmt.Errorf("additional field: %s", key)

            }

        }

        // Декодировать тело запроса в dest по адресу в памяти

        r.Body.Close()

        err = json.Unmarshal(body, &dest)

        return err

    }

    // Закодировать ответ, содержащий ошибку

    func EncodeErrorResponse(\_ context.Context, err error, w http.ResponseWriter) {

        w.Header().Set("Content-Type", "application/json; charset=utf-8")

        w.WriteHeader(http.StatusInternalServerError)

        json.NewEncoder(w).Encode(map[string]string{

            "error": err.Error(),

        })

    }

    // Закодировать ответ

    func EncodeResponse(\_ context.Context, w http.ResponseWriter, response interface{}) error {

        w.Header().Set("Content-Type", "application/json; charset=utf-8")

        return json.NewEncoder(w).Encode(response)

    }

Теперь можно писать реализацию транспортного уровня для **/services/product/backend/transport/http/service.go**. Внимательно изучите комментарии внутри кода:

    package http

    import (

        "context"

        "net/http"

        "strconv"

// Псевдоним, чтобы избежать конфликтов имен пакетов

        kithttp "github.com/go-kit/kit/transport/http"

        "github.com/gorilla/mux"

        "github.com/joyzem/documents/services/base"

        "github.com/joyzem/documents/services/product/backend/transport"

        "github.com/joyzem/documents/services/product/dto"

    )

    // NewService создает новый HTTP-обработчик, используя указанные входные точки transport.Endpoints и опции kithttp.ServerOption

    func NewService(

        svcEndpoints transport.Endpoints,

        options []kithttp.ServerOption,

    ) http.Handler {

        // Создается новый роутер mux

        router := mux.NewRouter()

        // Устанавливается кодировщик ошибок errorEncoder, который будет использоваться для кодирования ответов об ошибках

        errorEncoder := kithttp.ServerErrorEncoder(base.EncodeErrorResponse)

        // В список опций добавляется кодировщик ошибок

        options = append(options, errorEncoder)

        // Для следующих методов и путей создаются обработчики kithttp.NewServer, которые реализуют интерфейс http.Handler

        // Каждому методу (POST, GET, DELETE, PUT) и пути (/products, /units) создается обработчик функцией kithttp.NewServer,

        // которая принимает эндпоинт, функцию, декодирующую запрос, функцию, кодирующую ответ и опции

        // Одной из опций является кодирование ошибок. Если в процессе работы сервиса произойдет ошибка, то обработчик вернет

        // JSON ответ в виде {"error": "описание ошибки"}

        router.Methods("POST").Path("/products").Handler(

            kithttp.NewServer(

                svcEndpoints.CreateProduct, // Первый аргумент – endpoint.Endpoint

                decodeCreateProductRequest, // Второй – функция, декодирующая запрос

                base.EncodeResponse, // Функция, кодирующая ответ

                options..., // Опции, например, кодировщик ошибок

            ))

        router.Methods("GET").Path("/products").Handler(

            kithttp.NewServer(

                svcEndpoints.GetProducts,

                decodeGetProductsRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("PUT").Path("/products").Handler(

            kithttp.NewServer(

                svcEndpoints.UpdateProduct,

                decodeUpdateProductRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("DELETE").Path("/products").Handler(

            kithttp.NewServer(

                svcEndpoints.DeleteProduct,

                decodeDeleteProductRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("GET").Path("/products/{id}").Handler(

            kithttp.NewServer(

                svcEndpoints.ProductById,

                decodeProductByIdRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("POST").Path("/units").Handler(

            kithttp.NewServer(

                svcEndpoints.CreateUnit,

                decodeCreateUnitRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("GET").Path("/units").Handler(

            kithttp.NewServer(

                svcEndpoints.GetUnits,

                decodeGetUnitsRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("PUT").Path("/units").Handler(

            kithttp.NewServer(

                svcEndpoints.UpdateUnit,

                decodeUpdateUnitRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("DELETE").Path("/units").Handler(

            kithttp.NewServer(

                svcEndpoints.DeleteUnit,

                decodeDeleteUnitRequest,

                base.EncodeResponse,

                options...,

            ))

        router.Methods("GET").Path("/units/{id}").Handler(

            kithttp.NewServer(

                svcEndpoints.UnitById,

                decodeUnitByIdRequest,

                base.EncodeResponse,

                options...,

            ))

        return router

    }

    // Следующие функции декодируют запрос и возвращают требуемую структуру данных для дальнейшей работы в эндпоинте

    // Приходит http.Request r, и он декодируется с помощью функции base.DecodeBody(r, &req), которая

    // возвращает требуемую структуру или ошибку, если она произошла

    func decodeCreateProductRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.CreateProductRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeGetProductsRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        // У этого запроса нет тела, и декодировать нечего

        return dto.GetProductsRequest{}, nil

    }

    func decodeUpdateProductRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.UpdateProductRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeDeleteProductRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.DeleteProductRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeProductByIdRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        // Получение id с помощью методов библиотеки mux

        idStr := mux.Vars(r)["id"]

// Конвертация строки в int

        id, err := strconv.Atoi(idStr)

        if err != nil {

            return nil, err

        }

        req := dto.ProductByIdRequest{Id: id}

        return req, nil

    }

    func decodeCreateUnitRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.CreateUnitRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeGetUnitsRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.GetUnitsRequest

        return req, nil

    }

    func decodeUpdateUnitRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.UpdateUnitRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeDeleteUnitRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        var req dto.DeleteUnitRequest

        err := base.DecodeBody(r, &req)

        return req, err

    }

    func decodeUnitByIdRequest(\_ context.Context, r \*http.Request) (interface{}, error) {

        idStr := mux.Vars(r)["id"]

        id, err := strconv.Atoi(idStr)

        if err != nil {

            return nil, err

        }

        req := dto.UnitByIdRequest{Id: id}

        return req, nil

    }

## Запуск программы

Создайте файл **/services/product/backend/cmd/main.go**

Этот файл – входная точка в программу. Алгоритм работы программы следующий:

Подключение к базе данных;

Получение экземпляров репозиториев;

Создание сервиса;

Создание эндпоинтов;

Создание маршрутизатора;

Запуск сервера.

Так как к базе данных надо будет подключаться из всех сервисов, вынесем подключение в отдельную функцию из пакета base в файле **/services/base/database.go**:

Сперва необходимо установить библиотеку PostgreSQL для Go. Введите в терминале go get github.com/lib/pq;

    package base

    import (

        "database/sql"

        "fmt"

    \_ "github.com/lib/pq" // Драйвер PostgreSQL. Необходимо прописать вручную

    )

    // Функция подключается к базе данных, используя драйвер PostgreSQL

    func ConnectToDb() (\*sql.DB, error) {

        // Имя хоста и пользователя базы данных. Используется GetEnv, так как в Docker-контейнере имена переменных могут отличаться

        databaseHost := "localhost"

        databaseUser := "student"

        // Строка подключения

        connection := fmt.Sprintf("postgresql://%s:qwerty@%s:5432/proxy\_db?sslmode=disable", databaseUser, databaseHost)

        var db \*sql.DB

        var err error

        db, err = sql.Open("postgres", connection)

        if err != nil {

            return nil, err

        }

        return db, nil

    }

Код для **/services/product/backend/cmd/main.go**:

    package main

    import (

        "fmt"

        "net/http"

        "os"

        \_ "github.com/lib/pq"

        "github.com/joyzem/documents/services/base"

        "github.com/joyzem/documents/services/product/backend/implementation"

        "github.com/joyzem/documents/services/product/backend/transport"

        httptransport "github.com/joyzem/documents/services/product/backend/transport/http"

        kithttp "github.com/go-kit/kit/transport/http"

    )

    func main() {

        // Подключение к БД

        db, err := base.ConnectToDb()

        if err != nil {

            base.LogError(err)

        // Аварийный выход из программы

            os.Exit(-1)

        }

        defer db.Close()

        // Репозиторий товаров

        productRepo := implementation.NewProductRepo(db)

        // Репозиторий единиц измерения

        unitRepo := implementation.NewUnitRepository(db)

        // Создание сервиса

        svc := implementation.NewService(productRepo, unitRepo)

        // Создание эндпоинтов

        endpoints := transport.MakeEndpoints(svc)

        // Создание маршрутизатора

        h := httptransport.NewService(endpoints, []kithttp.ServerOption{})

        // Запуск сервиса по адресу http://localhost:7071

        fmt.Println("Listening on 7071...")

        if err := http.ListenAndServe(":7071", h); err != nil {

            base.LogError(err)

            os.Exit(-1)

        }

    }

* С помощью команды cd services/product/backend/cmd перейдите в папку с программой.
* Введите команду go run main.go;
* Если потребуется, дайте требуемое разрешение для брандмауэра;
* Если все сделано правильно, должна появиться надпись Listening on 7071…;
* Откройте браузер и введите адрес «http://localhost:7071/units»;
* Должен отобразиться список единиц измерения:

{

"units": [

{

"id": 0,

"name": "Не указано"

},

{

"id": 1,

"name": "шт."

},

{

"id": 2,

"name": "кг"

},

{

"id": 3,

"name": "уп"

},

{

"id": 4,

"name": "м"

}

]

}

## Тестирование

**Postman** – это инструмент для тестирования API, который позволяет отправлять HTTP-запросы к веб-серверам и анализировать ответы, полученные от серверов.

1. Установите Postman: скачайте и установите приложение Postman на свой компьютер. Для этого перейдите на сайт https://www.postman.com/downloads/ и загрузите версию приложения для вашей операционной системы;
2. Пропустите авторизацию, нажав на ссылку в левом нижнем углу приложения;
3. Создайте новый запрос: откройте Postman и нажмите на кнопку "New" (Новый) в левом верхнем углу экрана, затем на "HTTP Request". Затем выберите GET запрос возле поля с адресом;
4. Укажите URL-адрес: введите URL-адрес, на который вы хотите отправить запрос. Так как мы тестируем на локальной машине, URL будет начинаться с <http://localhost>. Номер порта, на котором работает приложение – 7071, поэтому BASE URL будет <http://localhost:7071>;
5. Чтобы получить список единиц измерения, добавьте к BASE URL адрес единиц измерения – "/units";
6. Выполните GET запрос;
7. Выберите POST запрос;
8. Добавьте тело запроса: если вам нужно отправить тело запроса, например, в POST-запросе, перейдите на вкладку "Body" (Тело) и добавьте нужные данные в соответствующем формате. Нажмите на переключатель "raw" и введите следующий JSON текст:

{

     "unit": "тест"

}

1. Отправьте запрос: после того, как вы настроили запрос, нажмите на кнопку "Send" (Отправить), чтобы отправить запрос на сервер;
2. Анализируйте ответ: после того, как сервер ответит на запрос, вы увидите ответ на странице Postman. Вы можете анализировать ответ и проверять, соответствуют ли данные, которые вы получили, вашим ожиданиям:

{

     "unit": {

         "id": <только что созданный id>,

         "name": "тест"

     }

}

1. Выберите PUT запрос;
2. Введите тело запроса:

{

     "unit": {

         "id": <только что созданный id>,

         "name": "релиз"

     }

}

1. Нажмите кнопку "Send";
2. Выберите DELETE запрос;
3. Введите тело запроса:

{

        "id": <только что созданный id>

}

1. Выполните запрос;
2. Если запрос выполнен без ошибки, то в ответе будут две фигурные скобки: "{}"

# Лабораторная работа № 4. Создание Frontend-микросервиса

## Цель

Разработать frontend-часть микросервиса товаров.

## Немного теории

Создание интерфейса сайта – это один из основных этапов веб-разработки, который включает в себя создание дизайна, структуры и элементов управления, позволяющих пользователям взаимодействовать с сайтом. Для создания интерфейса сайта используются языки разметки HTML и CSS, а также язык программирования JavaScript.

### HTML

**Определение документа**

Каждый HTML-документ должен начинаться с определения документа, которое выглядит так:

    <!DOCTYPE html>

    <html>

    <head>

    <title>Название страницы</title>

    </head>

    <body>

    ...

    </body>

    </html>

**Разметка текста**

HTML используется для создания структуры документа и разметки текста. Стандартные элементы, используемые для разметки текста, включают в себя заголовки, параграфы, списки и теги форматирования, такие как жирный и курсивный текст.

Примеры:

<!-- Комментарии -->

    <h1>**Заголовок первого уровня**</h1>

    <p>Параграф текста</p>

    <ul>

        <li>• Элемент списка 1</li>

        <li>• Элемент списка 2</li>

        <li>• Элемент списка 3</li>

    </ul>

    <em>*Текст курсивом*</em> <!—также <i></i> -->

    <strong>**Текст жирным шрифтом**</strong> <!—также <b></b> -->

**Разметка ссылок и изображений**

HTML также используется для создания ссылок и вставки изображений на страницу. Для создания ссылки используется тег <a>, а для вставки изображения – <img>.

    <a href="http://example.com">Ссылка на сайт</a>

    <img src="image.jpg" alt="Описание изображения">

**Разметка таблиц**

HTML используется для создания таблиц. Для этого используются теги <table>, <tr>, <th> и <td>.

Пример:

    <table>

        <tr> <!-- Ряд -->

            <th>Заголовок 1</th>

            <th>Заголовок 2</th>

        </tr>

        <tr>

            <td>Ячейка 1</td>

            <td>Ячейка 2</td>

        </tr>

    </table>

**Разметка форм**

HTML используется для создания форм, которые позволяют пользователям отправлять данные на сервер. Для этого используются теги <form>, <input> и <button>.

Пример:

    <form>

        <label for="name">Имя:</label>

        <input type="text" id="name" name="name"><br><br>

        <label for="email">Email:</label>

        <input type="email" id="email" name="email"><br><br>

        <button type="submit">Отправить</button>

    </form>

**Классы и идентификаторы (id)**

Классы и идентификаторы (id) являются атрибутами, которые можно присваивать HTML-элементам для описания их стиля и поведения на странице. Классы используются для группировки элементов с одинаковыми стилями, что позволяет легко изменять стиль всех элементов в группе, применяя стили к классу. Идентификаторы, с другой стороны, используются для определения уникальных элементов на странице, таких как навигационные меню или заголовки. Они могут быть использованы для навигации на странице или для целей JavaScript, когда необходимо ссылаться на конкретный элемент на странице.

### CSS

В рамках этого проекта будут использоваться готовые стили, чтобы уменьшить время разработки, однако для этого надо понимать основы CSS.

CSS (Cascading Style Sheets) - это язык стилей, который используется для оформления веб-страниц. Он позволяет определить стиль и внешний вид элементов HTML, таких как шрифты, цвета, расположение и размеры.

Чтобы использовать CSS в HTML, добавьте ссылку на стили в ваш HTML файл, используя тег <link>.

    <head>

        <title>Название страницы</title>

        <link rel="stylesheet" type="text/css" href="style.css">

    </head>

**Стили элементов**

Определить стиль для элементов HTML можно с помощью селекторов, которые указывают, какие элементы должны быть стилизованы. Например, чтобы изменить цвет текста для всех заголовков H1 на странице, используйте следующий код:

    h1 {

        color: blue;

    }

В CSS классы используются для определения одинакового стиля для группы элементов HTML. Классы позволяют определить стиль, который будет применяться ко всем элементам с заданным классом. Для определения класса в CSS используется символ точки перед именем класса, например:

    .my-class {

        color: blue;

      }

Здесь ".my-class" - это имя класса, которое вы можете использовать для любого элемента, чтобы применить к нему стиль, определенный в CSS.

Тег <div> в HTML является блочным элементом, который используется для создания контейнеров на странице. Вы можете использовать <div> для группировки элементов, которые нужно стилизовать вместе. Например:

    <div class="container">

        <h1>Заголовок страницы</h1>

        <p>Описание страницы</p>

    </div>

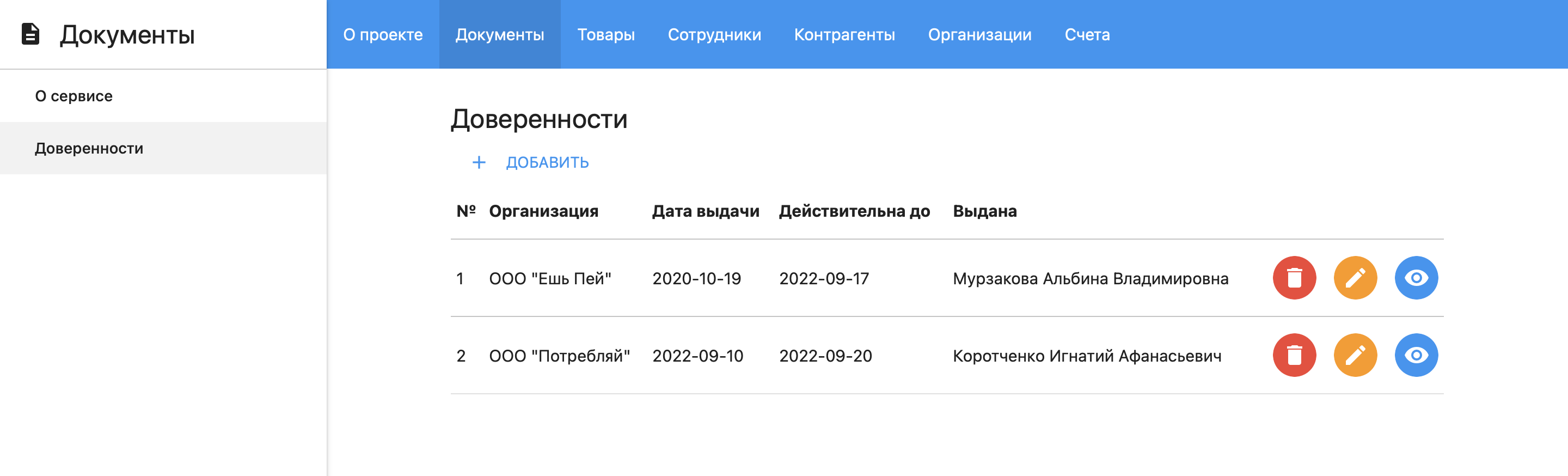
### Materialize CSS

Materialize CSS - это CSS-фреймворк, основанный на принципах дизайна Google Material Design, который обеспечивает множество готовых компонентов и стилей для создания привлекательных и удобных пользовательских интерфейсов. Вот несколько причин, почему стоит использовать Materialize CSS:

1. Он обеспечивает реализацию принципов дизайна Google Material Design, что делает ваш сайт или приложение более современным и удобным в использовании для пользователей.
2. Фреймворк предоставляет широкий набор готовых компонентов (например, кнопки, формы, карточки, навигационные меню и т.д.), что упрощает процесс создания пользовательского интерфейса и сокращает время разработки.
3. Materialize CSS основан на сетке Bootstrap, что обеспечивает простоту и гибкость при создании адаптивных и отзывчивых дизайнов.
4. Фреймворк также содержит множество встроенных стилей и классов, которые упрощают управление внешним видом вашего сайта или приложения.
5. Он имеет активное сообщество разработчиков и пользователей, которые постоянно работают над его улучшением и развитием, что делает его более надежным и актуальным в долгосрочной перспективе.

Таким образом, использование Materialize CSS может значительно ускорить процесс создания пользовательского интерфейса.

Интерфейс следующей страницы построен с помощью готовых стилей Materialize CSS:



### JavaScript

JavaScript является одним из основных языков программирования для разработки фронтэнд-части веб-сайтов и приложений. Вот несколько причин, почему JavaScript используется во фронтэнд:

1. Он позволяет создавать интерактивные пользовательские интерфейсы, что обеспечивает более легкий и удобный доступ к контенту сайта и повышает уровень удобства использования.
2. JavaScript обеспечивает динамическое обновление страницы без необходимости перезагрузки, что позволяет создавать более гибкие и быстрые приложения.
3. Он позволяет создавать анимацию и визуализацию данных на странице, что обеспечивает более привлекательный дизайн.
4. JavaScript может использоваться для создания валидации форм и обработки данных, что позволяет создавать более безопасные и удобные формы для пользователей.
5. С его помощью можно реализовывать множество различных функций, таких как слайдеры, всплывающие окна, меню и т.д., что делает сайты и приложения более функциональными и удобными для пользователей.

Хотя JavaScript является стандартом для разработки фронтэнд-части веб-сайтов и приложений, в рамках данной методички он будет использоваться только для инициализации интерфейса. Для работы с данными будет использоваться библиотека templates.

### HTML/Templates

Пакет "html/template" является частью стандартной библиотеки Go и предоставляет мощный и гибкий механизм для создания HTML-шаблонов. Он позволяет создавать динамические HTML-страницы, которые могут автоматически заполняться данными из структур данных Go.

Пример использования "html/template" с использованием HTML-файла в качестве шаблона:

template.html

-------------------------------------------------------------------------

    <!DOCTYPE html>

    <html>

        <head>

            <title>{{.Title}}</title>

        </head>

        <body>

            <h1>{{.Heading}}</h1>

            <ul>

            {{range .Items}}

                <li>{{.}}</li>

            {{end}}

            </ul>

        </body>

    </html>

main.go

-------------------------------------------------------------------------

    package main

    import (

        "html/template"

        "os"

    )

    type Page struct {

        Title   string

        Heading string

        Items   []string

    }

    func main() {

        page := Page{

            Title:   "My Page",

            Heading: "Welcome to My Page!",

            Items: []string{

                "Item 1",

                "Item 2",

                "Item 3",

            },

        }

        tmpl, err := template.ParseFiles("template.html")

        if err != nil {

            panic(err)

        }

        err = tmpl.Execute(os.Stdout, page)

        if err != nil {

            panic(err)

        }

    }

В этом примере мы создаем структуру "Page", которая содержит заголовок, заголовок страницы и список элементов. Затем мы парсим HTML-шаблон из файла "template.html" с помощью функции "template.ParseFiles()". Далее мы заполняем шаблон данными и выводим результат в стандартный вывод с помощью функции "Execute()".

## Разработка Frontend-микросервиса

Этапы разработки frontend микросервиса:

1. Создать html-разметку и js-код для страниц:

* Определить, какие страницы должен содержать микросервис
* Создать html-разметку для каждой страницы с использованием соответствующих элементов
* Написать js-код для обработки пользовательских действий на страницах (например, клики на кнопки, отправка форм)

1. Создать функцию получения адреса бэка:

* Написать функцию для получения адреса бэкенда, например, в виде константы или функции.

1. Написать хэндлеры:

* Определить, какие запросы и ответы должен обрабатывать микросервис
* Написать хэндлеры для каждого запроса, используя функцию получения адреса бэка, и отправлять соответствующие запросы на бэкенд
* Обработать ответы от бэкенда и обновить html-разметку страницы в соответствии с полученными данными

1. Написать router:

* Определить, какие адреса должен обрабатывать микросервис
* Написать router для перенаправления запросов на соответствующие хэндлеры

1. Написать main.go:

* Создать экземпляр http-сервера
* Зарегистрировать router
* Запустить http-сервер

### Структура папок

Рассмотрим продолжение структуры папок для микросервиса товаров:

services

├───base

└───product

├───backend

├───domain

├───dto

└───frontend // frontend микросервис

├───cmd // входная точка в программу

├───handlers // обработчики эндпоинтов

├───router // маршрутизатор

├───static // статичные файлы

│ ├───html // html-шаблоны

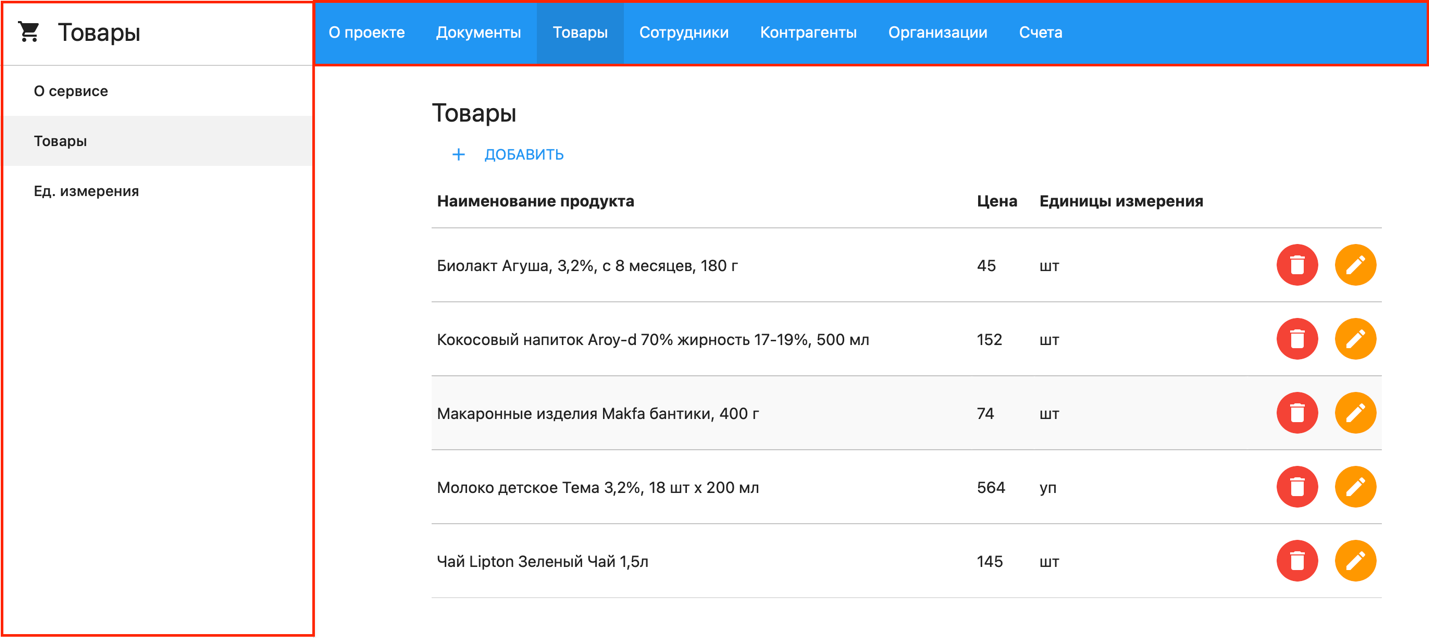
│ └───js // скрипты js

└───utils // утилиты для frontend

* services/product/frontend/cmd: содержит входную точку в программу фронтэнда.
* services/product/frontend/handlers: содержит обработчики для эндпоинтов фронтэнда. Эндпоинтом во frontend называется адрес страницы. Например, http://localhost**/product/units**. Обработчиком является функция, которую вызывает **маршрутизатор**
* services/product/frontend/router: содержит маршрутизатор фронтэнда. **Маршрутизатор** (или роутер) – это компонент, который используется для определения, какой код должен быть выполнен при обращении к определенному URL-адресу. Он отвечает за связывание URL–адресов с функциями, которые должны быть выполнены при обращении к этим URL-адресам. Обычно маршрутизатор является ключевым компонентом веб-приложения.
* services/product/frontend/static: содержит статические файлы, используемые фронтэндом, включая HTML-шаблоны и скрипты JS.
* services/product/frontend/utils: содержит утилиты, которые используются фронтэндом. Сюда входит, например, получение адреса бэкенда.

### Установка стилей

Интерфейс сайта представлен в виде двухуровневой навигации и окна для отображения текущей HTML-страницы.



Одним из важнейших компонентов сайта являются файлы фреймворка Materialize CSS. Скачать стили CSS и скрипты JS можно по ссылке: <https://github.com/Dogfalo/materialize/releases/download/1.0.0/materialize-v1.0.0.zip>.

В корне проекта создайте папки **/static/css** и **/static/js**. Распаковав скачанный архив, скопируйте файлы **/materialize-2/css/materialize.css** и **/materialize-2/js/materialize.min.js** и вставьте их в папки **/static/css** и **/static/js** соответственно. Также в папке **/static/css** создайте файл **styles.css** и добавьте в него следующий код:

    .margin-right {

        margin-right: 16px

    }

.has-fixed-sidenav {

        left: 300px

    }

    body.has-fixed-sidenav {

        padding-left: 300px

    }

    .container {

        width: 80%

    }

    .has-fixed-sidenav .navbar-fixed nav.navbar {

        width: calc(100% - 300px);

        left: 300px

    }

    body:not(.has-fixed-sidenav) nav a.sidenav-trigger {

        display: block

    }

    body:not(.has-fixed-sidenav) nav.navbar .brand-logo {

        position: absolute;

        left: 50%;

        transform: translateX(-50%)

    }

    .sidenav .logo-container {

        padding: 0 16px;

        height: 64px;

        line-height: 64px;

        font-size: 24px;

        border-bottom: 1px solid rgba(0, 0, 0, 0.2);

        box-sizing: content-box

    }

    .sidenav .logo-container i.material-icons {

        line-height: inherit

    }

### Панели навигации

Так как на всех экранах верхняя панель навигации одинакова, будет целесообразно создать отдельный скрипт, который можно подключать во все HTML–страницы. В таком случае не придется делать изменения во всех HTML­–файлах при добавлении новых пунктов навигации.

Создайте файл **/static/js/nav.js** и добавьте в него следующий код:

    // Классы, используемые в этом коде, взяты из библиотеки Materialize CSS

    // Создание элемента навигационной панели (navbar)

    const navbar = document.createElement('nav')

    // Добавление класса z-depth-0, который устанавливает размер тени, равный нулю

    navbar.classList.add('z-depth-0')

    // Создание обертки для навигационной панели (navWrapper)

    const navWrapper = document.createElement('div')

    // Добавление обертки для панели навигации, цвет - blue

    navWrapper.classList.add('nav-wrapper', 'blue')

    // Добавление внутреннего элемента в navbar

    navbar.appendChild(navWrapper)

    // Создание списка навигации (navList).  Элемент ul - неупорядоченный список

    const navList = document.createElement('ul')

    // Выравнивание по левому краю

    navList.classList.add('left')

    navWrapper.appendChild(navList)

    // Создание списка ссылок навигации (navLinks)

    const navLinks = [

        {

            label: 'Документы',

            href: '/documents/proxies',

        },

        {

            label: 'Товары',

            href: '/product/products',

        },

        {

            label: 'Сотрудники',

            href: '/employee/employees',

        },

        {

            label: 'Контрагенты',

            href: '/customer/customers',

        },

        {

            label: 'Организации',

            href: '/organization/organizations',

        },

        {

            label: 'Счета',

            href: '/account/accounts',

        },

    ];

    // Добавление ссылок в список навигации (navList)

    navLinks.forEach(link => {

        // Добавление элемента в неупорядоченный список navList

        const navItem = document.createElement('li')

        navItem.classList.add('nav-item')

        navList.appendChild(navItem)

        // Создание ссылки

        const navLink = document.createElement('a')

        // Текст ссылки

        navLink.textContent = link.label

        // Адрес ссылки

        navLink.href = link.href

        // Добавление ссылки в navItem

        navItem.appendChild(navLink)

    });

    // Создание обертки для навигационной панели (navbarContainer)

    const navbarContainer = document.createElement('div')

    navbarContainer.classList.add('navbar-fixed')

    navbarContainer.appendChild(navbar)

    // Добавление навигационной панели (navbar) на страницу

    document.querySelector('header').appendChild(navbarContainer)

    // Получение текущего пути

    const currentPath = window.location.pathname

    // Получение элементов списка навигации (navItems)

    const navItems = document.querySelectorAll('.nav-item');

// Подсветка текущего пункта навигации

for (let i = 0; i < navItems.length; i++) {

    item = navItems[i]

    let tabLink = item.querySelector('a').getAttribute('href')

    rootUrl = tabLink.slice(0, tabLink.slice(1, tabLink.length).indexOf("/"))

    if (currentPath.includes(rootUrl)) {

        item.classList.add('active')

    } else {

        item.classList.remove('active')

    }

}

Боковая панель навигации меняется в зависимости от выбранного пункта в верхней, однако на всех экранах принцип заложен тот же. Создадим функцию, которая принимает название лого, текст для лого и список элементов, и будем вызывать её при необходимости.

Создайте файл **/static/js/sidenav.js** и добавьте в него следующий код:

    // Функция создает боковую панель навигации

    // logoText - текст возле логотипа микросервиса

    // logoIconName - наименование логотипа из библиотеки Google Icons

    // sidebarLinks - список ссылок для боковой панели навигации

    function createSideNav(logoText, logoIconName, sidebarLinks) {

        // создание списка

        const sidebar = document.createElement('ul')

        // добавление класса sidenav из Materialize CSS

        sidebar.classList.add('sidenav')

        // назначение уникального идентификатора

        sidebar.id = "sidenav-left"

        sidebar.style = "transform: translateX(0px);"

        // Создание контейнера для логотипа боковой панели навигации

        const logoContainer = document.createElement('li')

        logoContainer.classList.add('logo-container')

        // Текст возле логотипа

        logoContainer.textContent = logoText

        // Создание логотипа

        const logoIcon = document.createElement('i')

        logoIcon.classList.add('material-icons', 'left')

        logoIcon.textContent = logoIconName

        // Добавление всех элементов в тег 'header'

        logoContainer.appendChild(logoIcon)

        sidebar.appendChild(logoContainer)

        document.querySelector('header').appendChild(sidebar)

        sidebarLinks.forEach(link => {

            const navItem = document.createElement('li')

            navItem.classList.add('bold')

            if (window.location.pathname == link.href) {

                navItem.classList.add('active')

            }

            const itemLink = document.createElement('a')

            itemLink.classList.add('waves-effect')

            itemLink.href = link.href

            itemLink.textContent = link.label

            navItem.appendChild(itemLink)

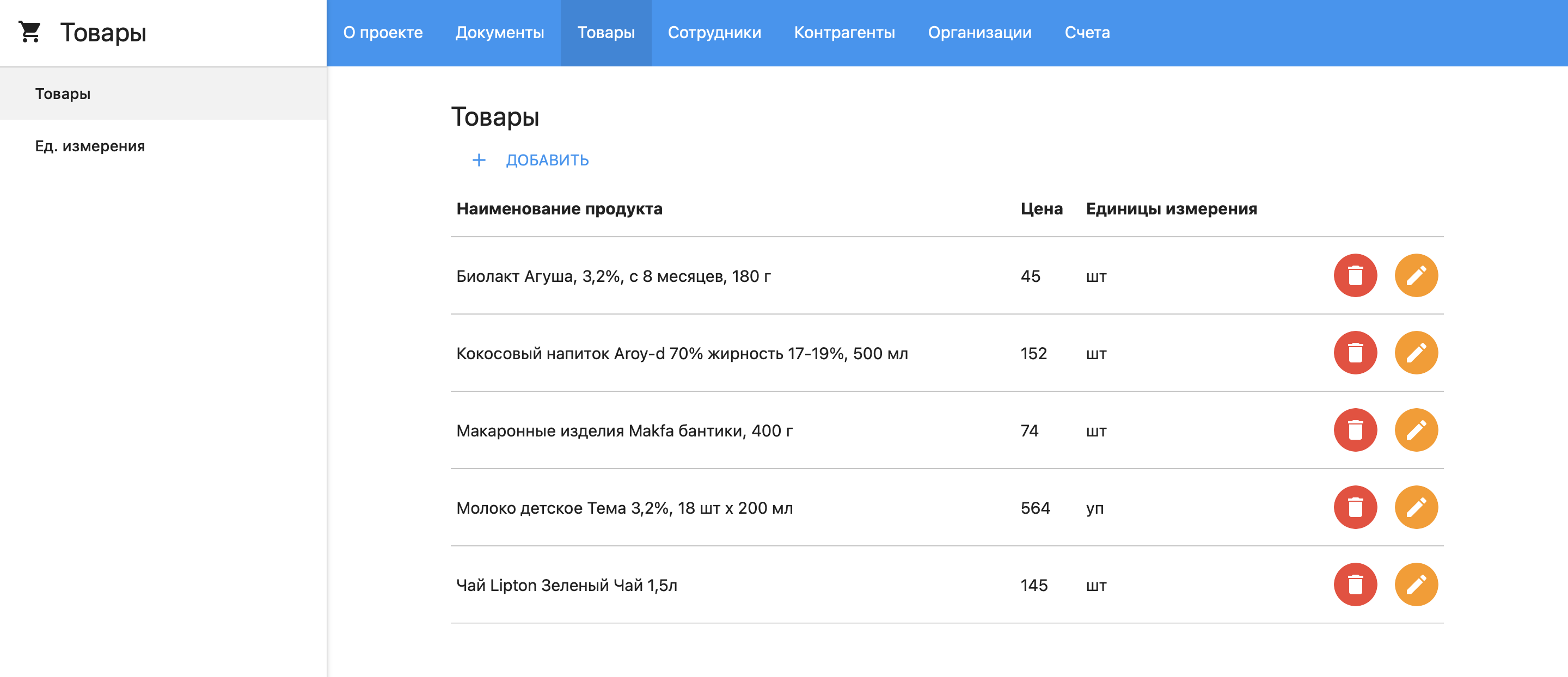
            sidebar.appendChild(navItem)

        })

    }

### Создание страниц для товаров

Конечный вид страницы товаров выглядит следующим образом:



Структуру можно описать следующим образом:

* Верхняя панель навигации;
* Боковая панель навигации;
* Контент страницы:
  + Заголовок «Товары»;
  + Кнопка «Добавить»;
  + Таблица с товарами:
    - Наименование продукта;
    - Цена;
    - Единицы измерения;
    - Кнопки для удаления и редактирования.

Внимательно изучите код для HTML-страницы товаров:

/services/product/frontend/static/html/products.html

-------------------------------------------------------------------------

  <!DOCTYPE html>

  <html lang="ru">

  <head>

    <meta charset="UTF-8">

    <link rel="stylesheet" type="text/css" href="/static/css/materialize.css">

    <link rel="stylesheet" type="text/css" href="/static/css/styles.css">

    <script src="/static/js/materialize.min.js"></script>

    <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

    <title>Товары</title>

  </head>

  <body class="has-fixed-sidenav">

    <header>

<!-- Здесь код добавляется с помощью JS скриптов -->

    </header>

    <main class="white">

<!-- Класс ‘container’ создает отступы по краям так, чтобы контент занимал 80% страницы -->

      <div class="container">

<!-- Класс ‘section’ создает отступы сверху и снизу -->

        <div class="section">

          <h5>Товары</h5>

<!-- Кнопка для перехода на страницы добавления товара -->

          <a class="waves-effect waves-blue btn-flat white blue-text" href="/product/products/create">

<!-- Иконка добавления из библиотеки Google Icons -->

<i class="material-icons left">add</i>

Добавить

</a>

<!-- Таблицы товаров. Класс ‘highlight’ подсвечивает элемент таблицы при наведении мышкой -->

          <table class="highlight">

<!-- Заголовки таблицы -->

            <thead>

              <th>Наименование продукта</th>

              <th>Цена</th>

              <th>Единицы измерения</th>

              <th></th>

            </thead>

<!-- Использование синтаксиса шаблонов. range берёт

структуру, переданную через библиотеку html/templates и

начинает пробег по переданному списку. Структура,

переданная в шаблон, содержит список, каждый элемент

которого содержит товар и единицу измерения -->

            {{range . }}

            <tr>

<!-- Сейчас шаблон находится в контексте элемента списка.

Через точку происходит обращение к полям структуры -->

              <td>{{.Product.Name}}</td>

              <td>{{.Product.Price}}</td>

              <td>{{.Unit.Name}}</td>

              <td>

<!-- Блок для кнопок -->

                <div class="valign-wrapper right">

<!-- Форма для отправки запроса на удаление товара -->

                  <form action="/product/products/delete" method="post" class="col">

<!-- Форма должна содержать в себе поля, которые

необходимо отправить на сервер. В данном случае

нет необходимости показывать id товара, однако

передать это поле надо, поэтому задаём тип “hidden”

и присваиваем значение через шаблон .Product.Id

-->

                    <input name="id" type="hidden" value="{{ .Product.Id }}">

<!-- Кнопка для отправки запроса формы. Обязательно

присваиваем тип “submit” -->

                    <button type="submit" class="btn-floating z-depth-0 waves-effect waves-light margin-right">

<!-- Иконка для удаления -->

                      <i class="material-icons red margin-right">delete</i>

                    </button>

                  </form>

<!-- Ссылка для перехода на страницу обновления

информации о товаре. Ссылка содержит id товара -->

                  <a href="/product/products/update/{{.Product.Id}}" class="col">

                    <button class="btn-floating waves-effect z-depth-0 waves-light ">

                      <i class="material-icons orange">edit</i>

                    </button>

                  </a>

                </div>

              </td>

            </tr>

<!-- Конец списка -->

            {{end}}

          </table>

        </div>

      </div>

    </main>

<!-- Подключение скриптов для создания панелей навигации -->

    <script type="text/javascript" src="/static/js/nav.js"></script>

    <script src="/static/js/sidenav.js"></script>

<!-- Инициализации боковой панели навигации для товаров -->

    <script src="/product/static/js/sidenav-product.js"></script>

  </body>

  </html>

Скрипт для инициализации боковой панели навигации выглядит следующим образом:

/services/product/frontend/static/js/sidenav-product.js

-------------------------------------------------------------------------

    var sidebarLinks = [

        {

            label: "Товары",

            href: "/product/products"

        },

        {

            label: "Ед. измерения",

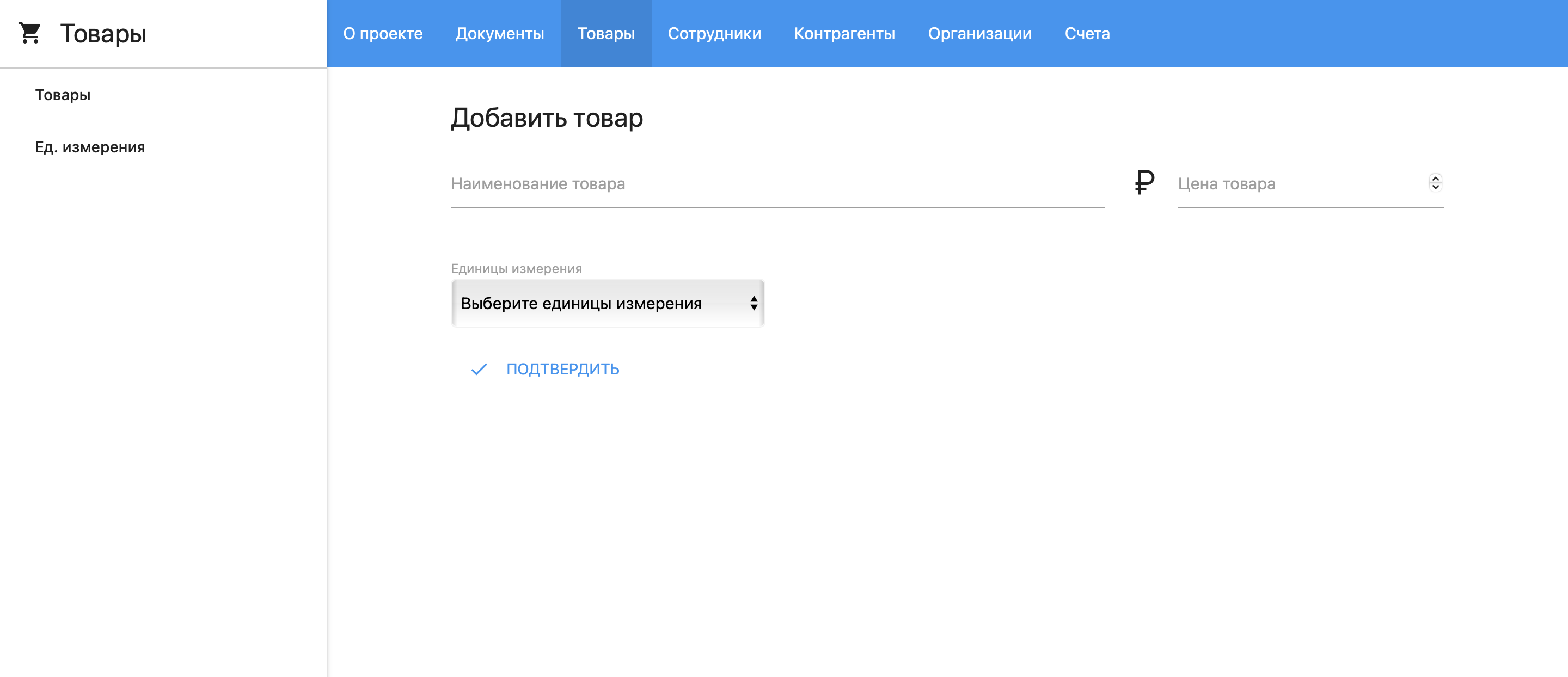
            href: "/product/units"

        }

    ]

    createSideNav("Товары", "shopping\_cart", sidebarLinks)

Страница для добавления товара выглядит следующим образом:



Два поля для ввода текста и один выпадающий список для выбора единицы измерения товара.

HTML-код представленной страницы:

/services/product/frontend/static/html/create-product.html

----------------------------------------------------------------------

    <!-- Определение типа документа и языка страницы -->

    <!DOCTYPE html>

    <html lang="ru">

    <head>

        <!-- Установка кодировки документа -->

        <meta charset="UTF-8">

        <!-- Подключение таблицы стилей Materialize -->

        <link rel="stylesheet" type="text/css" href="/static/css/materialize.css">

        <!-- Подключение скрипта Materialize -->

        <script src="/static/js/materialize.min.js"></script>

<link rel="stylesheet" type="text/css" href="/static/css/styles.css">

        <!-- Подключение шрифта Material Icons -->

        <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

        <!-- Заголовок страницы -->

        <title>Добавить товар</title>

    </head>

    <body class="has-fixed-sidenav">

        <!-- Создание шапки страницы -->

        <header>

        </header>

        <!-- Создание основного блока страницы -->

        <main class="white">

            <!-- Создание контейнера для размещения элементов страницы -->

            <div class="container">

                <!-- Создание секции страницы -->

                <div class="section">

                    <!-- Заголовок секции -->

                    <h5>Добавить товар</h5>

                    <!-- Форма для добавления товара -->

                    <form class="col s12" action="/product/products/create" method="post">

                        <!-- Ряд для элементов формы -->

                        <div class="row">

                            <!-- Поле для наименования товара -->

                            <div class="input-field col s8">

                                <input id="product\_name" name="name" type="text" class="validate">

                                <label for="product\_name">Наименование товара</label>

                            </div>

                            <!-- Поле для цены товара -->

                            <div class="input-field col s4">

                                <i class="material-icons prefix">currency\_ruble</i>

                                <input id="product\_price" name="price" type="number" class="validate" min="0"

                                    step="1">

                                <label for="product\_price">Цена товара</label>

                            </div>

                        </div>

                        <!-- Ряд для элементов формы -->

                        <div class="row">

                            <!-- Поле для выбора единиц измерения -->

                            <div class="col s4">

                                <label>Единицы измерения</label>

                                <select name="unit\_id" class="browser-default">

                                    <!-- Первый элемент выпадающего списка -->

                                    <option value=""disabled selected>Выберите единицы измерения</option>

                                    <!-- Генерация элементов выпадающего списка с помощью шаблонизатора -->

                                    {{ range . }}

                                    <option value="{{ .Id }}">{{ .Name }}</option>

                                    {{ end }}

                                </select>

                            </div>

                        </div>

                        <!-- Кнопка для отправки формы -->

                        <button class="waves-effect waves-blue btn-flat white blue-text" type="submit"><i

                                class="material-icons left">done</i>Подтвердить</button>

                    </form>

                </div>

            </div>

        </main>

        <!-- Подключение скрипта для работы навигации -->

        <script src="/static/js/nav.js"></script>

        <!-- Подключение скрипта для работы бокового меню -->

        <script src="/static/js/sidenav.js"></script>

        <!-- Подключение скрипта инициализации бокового меню-->

        <script src="/product/static/js/sidenav-product.js"></script>

    </body>

    </html>

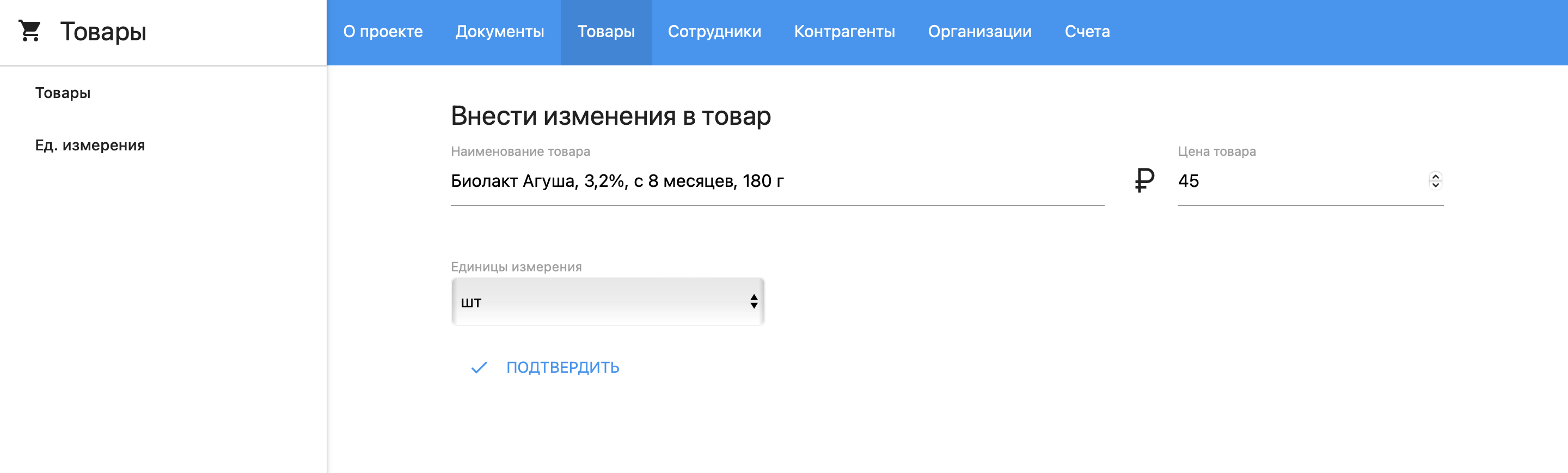
**Materialize CSS** предоставляет сеточную систему на основе классов   
 col и row . Эта система позволяет создавать адаптивные макеты, которые подстраиваются под размер экрана пользователя.

Класс row используется для создания строки, в которой размещаются элементы. Каждый элемент должен быть обернут в блок с классом col . Этот класс определяет ширину блока в зависимости от размера экрана. Например, класс col s12 означает, что элемент занимает 12 колонок на маленьких экранах (mobile), 12 колонок на средних экранах (tablet) и 12 колонок на больших экранах (desktop).

Класс s12 используется для определения ширины блока. Он имеет значения от s1 до s12, где s1 означает, что элемент занимает 1 колонку на маленьких экранах, а s12 – что элемент занимает все 12 колонок.

В коде выше класс col s8 определяет, что элемент занимает 8 колонок на маленьких экранах, а класс col s4 определяет, что другой элемент занимает 4 колонки на маленьких экранах.

Страница для внесения изменений в товар:



Код для страницы внесения изменений выглядит следующим образом:

/services/product/frontend/static/html/update-product.html

----------------------------------------------------------------------

    <!DOCTYPE html>

    <html lang="ru">

    <head>

        <meta charset="UTF-8">

        <link rel="stylesheet" type="text/css" href="/static/css/styles.css">

        <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

        <!-- Заголовок страницы -->

        <title>Обновить товар</title>

        <!-- Подключение стилей Materialize -->

        <link rel="stylesheet" type="text/css" href="/static/css/materialize.css">

        <!-- Подключение шрифтов Material Icons -->

        <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

    <body class="has-fixed-sidenav">

        <header>

        </header>

        <main class="white">

            <div class="container">

                <div class="section">

                    <!-- Заголовок формы -->

                    <h5>Внести изменения в товар</h5>

                    <!-- Форма отправки данных для обновления товара -->

                    <form class="col s12" action="/product/products/update" method="post">

                        <!-- Скрытое поле с ID товара -->

                        <input id="product\_id" name="id" type="hidden" value="{{ .Product.Id }}">

                        <div class="row">

                            <!-- Поле для ввода наименования товара -->

                            <div class="input-field col s8">

                                <!-- Чтобы отображать обратные кавычки, необходимо добавить тег html внутри шаблона-->

                                <input id="product\_name" name="name" type="text" class="validate"

                                    value="{{ .Product.Name | html }}">

                                <label for="product\_name">Наименование товара</label>

                            </div>

                            <!-- Поле для ввода цены товара -->

                            <div class="input-field col s4">

                                <i class="material-icons prefix">currency\_ruble</i>

                                <input id="product\_price" name="price" type="number" value="{{ .Product.Price }}"

                                    class="validate" min="0" step="1">

                                <label for="product\_price">Цена товара</label>

                            </div>

                        </div>

                        <div class="row">

                            <div class="col s4">

                                <!-- Выбор единицы измерения товара -->

                                <label>Единицы измерения</label>

                                <!-- Так как после вызова range .Units шаблонизатор перейдет в контекст

                                    единиц измерения, необходимо заранее запомнить id единицы измерения товара,

                                    чтобы подставить её в выпадающий список

                                -->

                                {{ $productUnitId := .Product.UnitId }}

                                <select name="unit\_id" class="browser-default">

                                    {{ range .Units }}

                                    <!-- Сравнение id текущей единицы измерения с productUnitId-->

                                    {{ if eq .Id $productUnitId }}

                                    <!-- Выбранная единица измерения -->

                                    <option value="{{ .Id }}" selected>{{ .Name | html}}</option>

                                    {{ else }}

                                    <option value="{{ .Id }}">{{ .Name | html }}</option>

                                    {{ end }}

                                    {{ end }}

                                </select>

                            </div>

                        </div>

                        <!-- Кнопка подтверждения изменений -->

                        <button class="waves-effect waves-blue btn-flat white blue-text" type="submit">

                            <i class="material-icons left">done</i>Подтвердить

                        </button>

                    </form>

                </div>

            </div>

        </main>

        <!-- Подключение скриптов Materialize -->

        <script src="/static/js/materialize.min.js"></script>

        <!-- Подключение скриптов навигации -->

        <script type="text/javascript" src="/static/js/nav.js"></script>

        <script src="/static/js/sidenav.js"></script>

        <script src="/product/static/js/sidenav-product.js"></script>

    </body>

    </html>

### Создание обработчиков

Обработчик (handler) в Go – это функция, которая обрабатывает запрос HTTP. В общем случае, обработчик должен иметь следующую сигнатуру:

    func handler(w http.ResponseWriter, r \*http.Request) {

        // обработка запроса

    }

Где w - объект http.ResponseWriter , который позволяет записывать ответ на запрос, а r - объект http.Request , который содержит информацию о запросе (метод, URL, заголовки, параметры и т.д.).

Обработчик может использоваться для любой задачи, связанной с запросами HTTP, таких как извлечение данных из запроса, обработка ошибок, проверка прав доступа, сохранение данных в базе данных и т.д. Обработчики могут быть созданы как функции внутри приложения, либо как методы структур.

Для получения, создания, удаления и изменения товара требуется создать следующие обработчики:

    // Обработчик страницы всех товаров

    func ProductsHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

    // Обработчик удаления товара

    func DeleteProductHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

    // Обработчик получения страницы добавления товара

    func CreateProductGetHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

    // Обработчик запроса на добавление товара

    func CreateProductPostHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

    // Обработчик страницы обновления товара

    func UpdateProductGetHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

    // Обработчик обновления товара

    func UpdateProductPostHandler(w http.ResponseWriter, r \*http.Request) {

        // ...

    }

Перед тем, как начать писать код, необходимо установить библиотеку для отправки запросов на бэкенд. Введите в терминал: go get github.com/levigross/grequests

Создайте файл **/services/product/frontend/handlers/product.go** и пропишите пакет **handlers**

**Получение товаров**

    // Обработчик запроса на страницу товаров

    func ProductsHandler(w http.ResponseWriter, r \*http.Request) {

        // Получение адреса сервера backend

        // Так как адрес сервиса товаров будет часто использоваться в обработчиках,

        // будет целесообразно вынести его в отдельную функцию.

        // Это также необходимо и по той причине, что при создании Docker-контейнера

        // адрес бэкенд сервера изменится

        productsUrl := fmt.Sprintf("%s/products", utils.GetBackendAddress())

        // Выполнение GET-запроса на сервер backend

        resp, err := grequests.Get(productsUrl, nil)

        if err != nil {

            // Обработка ошибки в случае неудачного запроса

            http.Error(w, err.Error(), http.StatusInternalServerError)

            return

        }

        var products dto.GetProductsResponse

        // Декодирование ответа от сервера backend в структуру GetProductsResponse

        if err := resp.JSON(&products); err != nil {

            http.Error(w, err.Error(), http.StatusInternalServerError)

            return

        }

        if products.Err != "" {

            // Обработка ошибки, переданной в ответе сервера backend

            http.Error(w, products.Err, http.StatusInternalServerError)

            return

        }

        // Получение списка единиц измерения из backend

        // Вынесен в отдельную функцию для повторных вызовов

        units, err := utils.GetUnitsFromBackend()

        if err != nil {

            http.Error(w, err.Error(), http.StatusInternalServerError)

            return

        }

        if units.Err != "" {

            http.Error(w, units.Err, http.StatusInternalServerError)

            return

        }

        // Создание списка товаров с указанием соответствующих единиц измерения

        type ProductItemTemplate struct {

            Product domain.Product

            Unit    domain.Unit

        }

        templateItems := []ProductItemTemplate{}

        for \_, product := range products.Products {

            templateItem := ProductItemTemplate{}

            templateItem.Product = product

            var productUnit domain.Unit

            for \_, unit := range units.Units {

                if unit.Id == product.UnitId {

                    productUnit = unit

                    break

                }

            }

            templateItem.Unit = productUnit

            templateItems = append(templateItems, templateItem)

        }

        // Загрузка шаблона страницы товаров и его рендеринг с передачей списка товаров в качестве аргумента

        productPage, \_ := template.ParseFiles("../static/html/products.html")

        productPage.Execute(w, templateItems)

    }

**Вспомогательные функции**

/services/product/frontend/utils/backend.go

----------------------------------------------------------------------

package utils

    import (

        "fmt"

        "github.com/joyzem/documents/services/base"

        "github.com/joyzem/documents/services/product/dto"

        "github.com/levigross/grequests"

    )

    const (

        BACKEND\_ADDRESS = "PRODUCT\_BACKEND\_ADDRESS"

    )

    // Получение адреса бэкенда

    func GetBackendAddress() string {

        // получение адреса из переменной окружения или использование http://localhost:7071

// Функция base.GetEnv принимает два аргумента (ключ и значение по умолчанию)

// и берет значение из переменной окружения для Docker-контейнера через ключ

// или же возвращает значение по умолчанию

// Напишите реализацию, которая возвращает значению по умолчанию (второй аргумент)

        address := base.GetEnv(BACKEND\_ADDRESS, "http://localhost:7071")

        return address

    }

    // Получение единиц измерения с бэкенда

    func GetUnitsFromBackend() (\*dto.GetUnitsResponse, error) {

        url := fmt.Sprintf("%s/units", GetBackendAddress())

        // отправка запроса и получение ответа

        resp, err := grequests.Get(url, nil)

        if err != nil {

            return nil, err

        }

        // парсинг ответа

        var data dto.GetUnitsResponse

        if err := resp.JSON(&data); err != nil {

            return nil, err

        }

        return &data, nil

    }

**Удаление товара**

    // Удаление товара

    func DeleteProductHandler(w http.ResponseWriter, r \*http.Request) {

        // Получаем значение id товара, который нужно удалить из формы

        id, \_ := strconv.Atoi(r.FormValue("id"))

        // Составляем URL-адрес, чтобы отправить запрос на удаление товара на бэкенд

        productsUrl := fmt.Sprintf("%s/products", utils.GetBackendAddress())

        // Отправляем запрос на удаление товара на бэкенд с помощью библиотеки grequests

        // Вторым аргументом передаем объект grequests.RequestOptions, содержащий JSON структуру

        resp, \_ := grequests.Delete(productsUrl, &grequests.RequestOptions{

            JSON: dto.DeleteProductRequest{Id: id},

        })

        // Получаем ответ в формате JSON и декодируем его в структуру deleteResponse

        var deleteResponse dto.DeleteProductResponse

        resp.JSON(&deleteResponse)

        // Если при удалении товара произошла ошибка, возвращаем ошибку в ответе HTTP

        if deleteResponse.Err != "" {

            http.Error(w, deleteResponse.Err, http.StatusInternalServerError)

            return

        }

        // Если товар успешно удален, перенаправляем пользователя на страницу со списком товаров

        http.Redirect(w, r, "/product/products", http.StatusSeeOther)

    }

**Страница добавления товара**

    // Обработчик страницы добавления товара

    func CreateProductGetHandler(w http.ResponseWriter, r \*http.Request) {

        // получить единицы измерения

        units, \_ := utils.GetUnitsFromBackend()

        if units.Err != "" {

            http.Error(w, units.Err, http.StatusInternalServerError)

            return

        }

        // шаблон добавления товара

        createProductPage, \_ := template.ParseFiles("../static/html/create-product.html")

        createProductPage.Execute(w, units.Units)

    }

**Добавление товара**

    // Обработчик запроса на добавление товара

    func CreateProductPostHandler(w http.ResponseWriter, r \*http.Request) {

        // Получаем параметры товара из формы

        productName := r.FormValue("name")                             // наименование товара

        productPrice, priceError := strconv.Atoi(r.FormValue("price")) // цена товара

        unitId, unitErr := strconv.Atoi(r.FormValue("unit\_id"))        // id ед.изм.

        // Проверяем корректность параметров товара

        if productName == "" || priceError != nil || unitErr != nil {

            http.Error(w, "fields validation error", http.StatusUnprocessableEntity)

            return

        }

        // Создаем запрос на добавление товара в бэкенд

        request := dto.CreateProductRequest{

            Name:   productName,

            Price:  productPrice,

            UnitId: unitId,

        }

        productUrl := fmt.Sprintf("%s/products", utils.GetBackendAddress())

        resp, \_ := grequests.Post(productUrl, &grequests.RequestOptions{

            JSON: request,

        })

        // Обрабатываем ответ от бэкенда

        var data dto.CreateProductResponse

        resp.JSON(&data)

        if data.Err != "" {

            http.Error(w, data.Err, http.StatusInternalServerError)

        } else {

            http.Redirect(w, r, "/product/products", http.StatusSeeOther)

        }

    }

**Страница обновления товара**

    // Обработчик страницы обновления товара

    func UpdateProductGetHandler(w http.ResponseWriter, r \*http.Request) {

        // парсинг id из адреса страницы

        id, \_ := strconv.Atoi(mux.Vars(r)["id"])

        // получение товара с бэка

        url := fmt.Sprintf("%s/products/%d", utils.GetBackendAddress(), id)

        resp, \_ := grequests.Get(url, &grequests.RequestOptions{

            JSON: dto.ProductByIdRequest{Id: id},

        })

        var product dto.ProductByIdResponse

        resp.JSON(&product)

        // товар не найден

        if product.Err != "" {

            http.Error(w, product.Err, http.StatusBadRequest)

            return

        }

        // получение единиц измерения с бэка

        unitsResp, \_ := utils.GetUnitsFromBackend()

        if unitsResp.Err != "" {

            http.Error(w, unitsResp.Err, http.StatusInternalServerError)

            return

        }

        // структура данных для шаблона

        type UpdateProductTemplate struct {

            Product \*domain.Product

            Units   []domain.Unit

        }

        data := UpdateProductTemplate{

            Product: product.Product,

            Units:   unitsResp.Units,

        }

        // шаблон страницы

        updateProductPage, \_ := template.ParseFiles("../static/html/update-product.html")

        updateProductPage.Execute(w, data)

    }

**Обновление товара**

    // Обработчик запроса на обновление товара

    func UpdateProductPostHandler(w http.ResponseWriter, r \*http.Request) {

        productId, \_ := strconv.Atoi(r.FormValue("id"))                // получаем id товара, который нужно обновить

        productName := r.FormValue("name")                             // получаем имя товара из формы

        productPrice, priceError := strconv.Atoi(r.FormValue("price")) // получаем цену товара из формы

        unitId, unitErr := strconv.Atoi(r.FormValue("unit\_id"))        // получаем id единицы измерения из формы

        // валидация формы

        if len(productName) == 0 || priceError != nil || unitErr != nil { // проверяем, что поля заполнены корректно

            http.Error(w, "fields\_validation\_error", http.StatusUnprocessableEntity) // возвращаем ошибку

            return

        }

        product := domain.Product{ // создаем новый объект товара для обновления

            Id:     productId,

            Name:   productName,

            Price:  productPrice,

            UnitId: unitId,

        }

        productUrl := fmt.Sprintf("%s/products", utils.GetBackendAddress()) // формируем URL для запроса к API

        request := dto.UpdateProductRequest{ // создаем объект запроса на обновление товара

            Product: product,

        }

        resp, \_ := grequests.Put(productUrl, &grequests.RequestOptions{ // отправляем PUT запрос на API

            JSON: request,

        })

        var updateResponse dto.UpdateProductResponse // создаем объект ответа на запрос обновления

        resp.JSON(&updateResponse)                   // получаем ответ от API и парсим его в объект ответа на запрос обновления

        if updateResponse.Err != "" {                // если в ответе есть ошибка, то возвращаем ее

            http.Error(w, updateResponse.Err, http.StatusInternalServerError)

            return

        }

        http.Redirect(w, r, "/product/products", http.StatusSeeOther) // перенаправляем пользователя на страницу товаров

    }

### Создание маршрутизатора

Библиотека **"github.com/rs/cors"** предназначена для обработки CORS (Cross-Origin Resource Sharing) веб-запросов. CORS - это механизм, который позволяет веб-страницам получать доступ к ресурсам на других доменах, которые были запрещены из соображений безопасности.

Введите в терминале: go get github.com/rs/cors

Создайте файл **/services/product/frontend/handler/router.go**

    package router

    import (

        "net/http"

        "github.com/gorilla/mux"

        "github.com/joyzem/documents/services/product/frontend/handlers"

        "github.com/rs/cors"

    )

    func GetRouter() http.Handler {

        // создание маршрутизатора

        router := mux.NewRouter()

        // обработка GET запроса. Выдача страницы всех товаров

        router.HandleFunc("/product/products", handlers.ProductsHandler)

        // обработка POST запроса. Удаление товара

        router.HandleFunc("/product/products/delete", handlers.DeleteProductHandler).Methods(http.MethodPost)

        // обработка GET запроса. Страница добавления товара

        router.HandleFunc("/product/products/create", handlers.CreateProductGetHandler).Methods(http.MethodGet)

        // обработка POST запроса. Добавление товара

        router.HandleFunc("/product/products/create", handlers.CreateProductPostHandler).Methods(http.MethodPost)

        // обработка GET запроса. Страница изменения товара

        router.HandleFunc("/product/products/update/{id}", handlers.UpdateProductGetHandler)

        // обработка POST запроса. Изменение товара

        router.HandleFunc("/product/products/update", handlers.UpdateProductPostHandler).Methods(http.MethodPost)

        // обработка обращения к глобальным статическим ресурсам: js, css. Используется в html

        router.PathPrefix("/static/").Handler(http.StripPrefix("/static/", http.FileServer(http.Dir("../../../../static"))))

        // обработка обращения к локальным статическим ресурсам

        router.PathPrefix("/product/static/").Handler(http.StripPrefix("/product/static/", http.FileServer(http.Dir("../static"))))

        // механизм, позволяющий отправлять запросы на другой домен (адрес)

        c := cors.New(cors.Options{

            AllowedOrigins: []string{"\*"},

            AllowedMethods: []string{http.MethodGet, http.MethodPost, http.MethodPut, http.MethodDelete},

        })

        // обертка над маршрутизатором

        handler := c.Handler(router)

        return handler

    }

### Запуск фронтэнда

Создайте файл **/services/product/frontend/cmd/main.go** и добавьте в него следующий код:

    func main() {

        // получение маршрутизатора

        handler := router.GetRouter()

        // запуск сервера

        fmt.Println("Listening on 8081...")

        if err := http.ListenAndServe(":8081", handler); err != nil {

            base.LogError(err)

            os.Exit(-1)

        }

    }

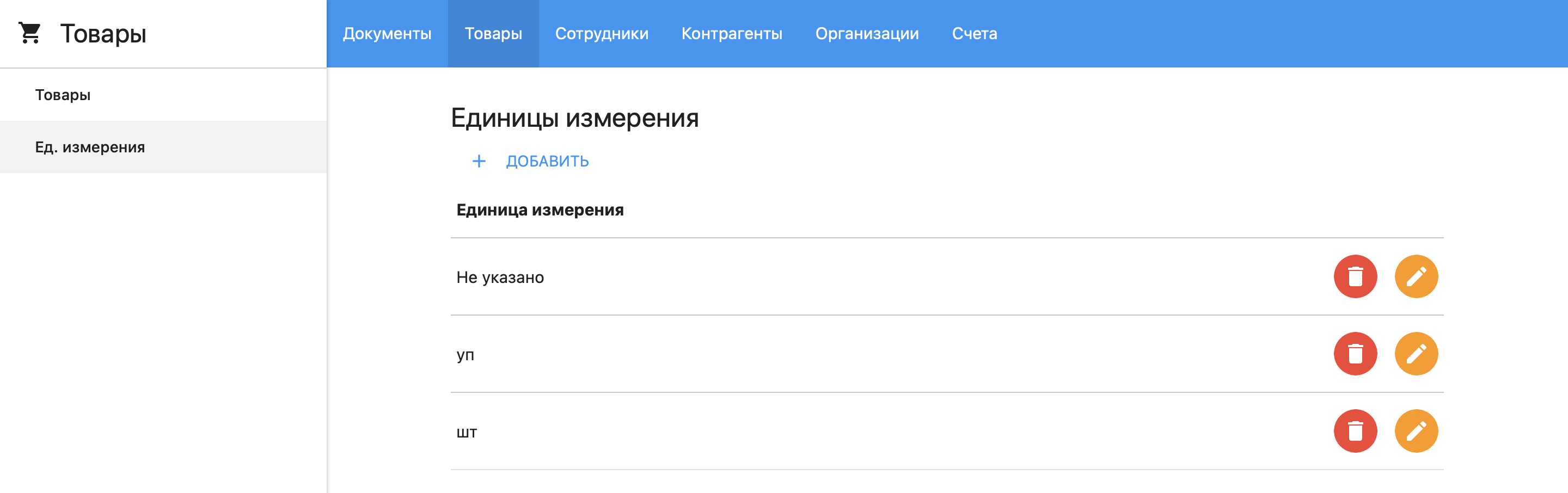
Откройте терминал и разделите его (Ctrl + Shift + 5). В левом окне запустите бэкенд сервер в папке **/services/product/backend/cmd**. В правом окне запустите фронтэнд сервер в папке **/services/product/frontend/cmd**.

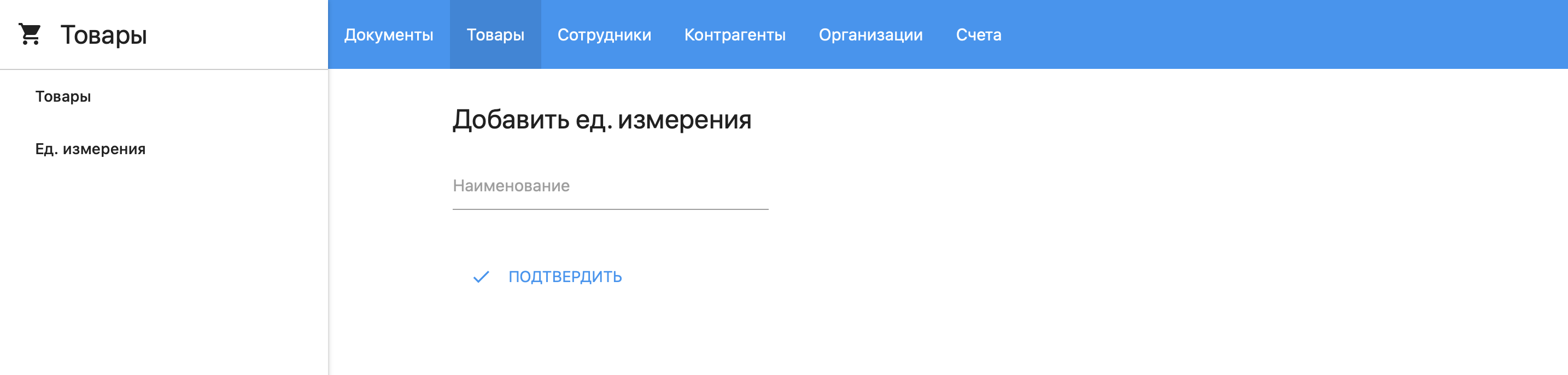
В оба терминала введите команду go run main.go.

Перейдите по адресу **http://localhost:8081/product/products**.

## Самостоятельное задание

Разработайте оставшуюся часть фронтэнда для единиц измерения:





В конечном варианте структура папки **/services/product/frontend**  выглядит следующим образом:

frontend

├── cmd

│ └── main.go

├── handlers

│ ├── product.go

│ └── unit.go

├── router

│ └── router.go

├── static

│ ├── html

│ │ ├── create-product.html

│ │ ├── create-unit.html

│ │ ├── index.html

│ │ ├── products.html

│ │ ├── units.html

│ │ ├── update-product.html

│ │ └── update-unit.html

│ ├── js

│ │ └── sidenav-product.js

└── utils

└── backend.go

# Лабораторная работа № 5. Объединение микросервисов в Docker.

## Цель

Объединить микросервисы в один проект с помощью Docker.

## Что такое Docker

**Docker** – это программное обеспечение для создания, управления и запуска приложений в изолированных контейнерах.

Контейнеры Docker позволяют запускать приложения в изолированной среде, которая включает в себя все необходимые зависимости и библиотеки, необходимые для работы приложения. Это облегчает развертывание приложений на разных платформах и упрощает процесс управления приложениями и их зависимостями.

Docker также облегчает процесс разработки и тестирования приложений. Разработчики могут создавать контейнеры для своих приложений и легко тестировать их на разных платформах, без необходимости устанавливать зависимости на каждом устройстве отдельно.

Еще одно преимущество Docker заключается в том, что он позволяет эффективно использовать ресурсы серверов. Контейнеры Docker могут запускаться и останавливаться быстро, а также масштабироваться в зависимости от потребностей, что позволяет более эффективно использовать ресурсы сервера.

Docker является очень полезным инструментом для разработки, тестирования и развертывания приложений, который позволяет упростить и ускорить процесс разработки, улучшить переносимость и масштабируемость приложений, а также повысить эффективность использования ресурсов сервера.

## Установка Docker

1. Перейдите на официальный сайт Docker (https://www.docker.com/), выберите "Get Started" в верхнем меню.
2. Нажмите "Download" и выберите нужную версию Docker для своей операционной системы.
3. Запустите установщик, следуя инструкциям на экране.
4. Оставьте настройки по умолчанию
5. Дождитесь завершения установки Docker.
6. Перезагрузите компьютер, чтобы применить изменения.
7. Откройте Docker Desktop и нажмите "Accept" на появившемся окне, чтобы принять лицензионное соглашение.
8. Если вам предложат обновить WSL (Windows Subsystem for Linux), откройте терминал и выполните команду "wsl --update", чтобы обновить WSL.
9. Дождитесь завершения установки обновлений и перезагрузки Docker Desktop.
10. Обучение можно пропустить, если вы уже знакомы с Docker и его основами.

После выполнения всех указанных действий Docker будет установлен и готов к использованию. Вы можете создавать, запускать и управлять контейнерами с помощью интерфейса Docker Desktop или через терминал с использованием команд Docker CLI.

## Dockerfile

Dockerfile - это текстовый файл, который используется для создания Docker-образов. Файл Dockerfile содержит инструкции для сборки образа и описывает, какие пакеты, файлы, конфигурации и другие зависимости нужно включить в образ.

Вот основы синтаксиса Dockerfile:

* FROM - определяет базовый образ, который будет использоваться для создания образа.
* RUN - выполняет команды внутри контейнера, чтобы установить пакеты и зависимости.
* COPY – копирует файлы из хоста в контейнер.
* ADD – копирует файлы из хоста в контейнер, но также может загружать файлы из Интернета и разархивировать архивы.
* WORKDIR – устанавливает рабочую директорию для всех последующих команд в Dockerfile. Та папка, откуда будут выполняться команды.
* EXPOSE – определяет порты, которые будут использоваться контейнером.
* CMD – определяет команду, которая будет запущена при запуске контейнера.

Пример Dockerfile’а для программы бэкенда товаров:

/services/product/backend/cmd/Dockerfile

----------------------------------------------------------------------

    # Использование golang:alpine в качестве билдера.

    # golang:alpine означает линукс с установленным golang

    # builder – это алиас, и он используется, чтобы скопировать   
 # бинарный файл в другой образ без установленного golang

    # это необходимо для того, чтобы образ не занимал много места

    # 300 мб с установленным golang либо 15 мб без golang

    FROM golang:alpine as builder

    # Директория, путь, в котором находится командная строка

    WORKDIR /app/services/product/backend/cmd

    # Копирование исходного кода в контейнер

    # Путь начинается с корня проекта, так как сборка происходит

    # с помощью docker-compose.yml файла, который находится в корне проекта

    # Очень важно соблюдать последовательность папок, так как в

# коде часто используются относительные пути, в том числе и в import командах

    # Копирование backend части

    COPY /services/product/backend /app/services/product/backend

    # Копирование пакетов, используемых в backend

    COPY /services/product/domain /app/services/product/domain

    COPY /services/product/dto /app/services/product/dto

    # Копирование базовых утилит для всех сервисов

    COPY /services/base /app/services/base

    # Копирование файлов модуля

    COPY /go.mod /app

    COPY /go.sum /app

    # Установка библиотек "RUN go mod download" пропущена, так как

    # вместо этого используется механизм "Volume" — общее хранилище между контейнерами

    # В docker-compose.yml файле прописано, что содержимое директории "vendor" копируется

    # в /go/pkg/mod — директории исходного кода для библиотек

    # Это позволяет не скачивать библиотеки каждый раз

    # Компиляция проекта

    RUN go build -o main .

    FROM alpine

    # Использование чистого Alpine (дистрибутив Linux). Характеризуется легковесностью

    # Это новый образ, который возьмет только исходный код

    WORKDIR /app/services/product/backend/cmd

    # Скопировать из builder исходный код в текущую рабочую директорию — из /app/services/product/backend/cmd/main в .

    COPY --from=builder /app/services/product/backend/cmd/main .

    # Установка порта

    EXPOSE 7071

    # Запуск бинарного файла

    CMD ["./main"]

## Использование docker-compose

**Docker Compose** – это инструмент, который позволяет запускать многоконтейнерные приложения в Docker-среде с помощью файла конфигурации в формате YAML. Вместо того чтобы запускать каждый контейнер отдельно, можно использовать **docker-compose.yml** файл, чтобы определить и управлять несколькими контейнерами вместе.

Инструкция по запуску Docker Compose:

1. Создайте файл **docker-compose.yml** в корневой папке вашего проекта;
2. В файле **docker-compose.yml** определите сервисы, которые вы хотите запустить. Каждый сервис должен содержать имя, образ, который будет использоваться, и другие параметры, такие как порты и переменные окружения;

  # Это версия Docker Compose, используемая для этого файла конфигурации.

  version: '3'

  # Секция, которая содержит определения сервисов для запуска

  services:

    # Имя сервиса

    product-back:

      # Docker-образ, который будет использоваться для запуска этого сервиса.

      image: documents:product-back

      # Имя контейнера, который будет создан при запуске этого сервиса.

      container\_name: product-back

      # Опция, которая указывает, что Docker должен построить образ на основе Dockerfile.

      build:

        # Опция указывает на папку, в которой находится docker-compose файл.

        context: .

        # ./services/product/backend/cmd/Dockerfile - указывает на путь к Dockerfile для product-back.

        dockerfile: ./services/product/backend/cmd/Dockerfile

      # Указывает на переменные окружения, которые будут использоваться в контейнере.

      environment:

        - DATABASE\_HOST=database

        - DATABASE\_USER=user

      # Указывает на папки, которые будут смонтированы () в контейнере.

      volumes:

        # Папка с модулями из этого проекта будет смонтирована в /go/pkg/mod

        # Позволяет не устанавливать модули из интернета при билде докерфайла

        - ./vendor:/go/pkg/mod

      # Какие порты открыты в сервисе

      ports:

        - "7071:7071"

      # Сервисы, от которых зависит данный сервис

      depends\_on:

        - database

    # Сервис базы данных

    database:

      # Образ базы данных берется из официального репозитория

      image: postgres:15.1

      # Установка окружения для базы данных

      environment:

        POSTGRES\_USER: student

        POSTGRES\_PASSWORD: qwerty

        POSTGRES\_DB: proxy\_db

      ports:

        - "5432:5432"

      # Инициализация базы данных

      volumes:

        - ./sql/create\_tables.sql:/docker-entrypoint-initdb.d/create\_tables.sql

Чтобы вынести все используемые библиотеку в папку **vendor**, введите в терминале go mod vendor

1. Необходимо внести изменения в функции base.ConnectToDb() и base.GetEnv()

/services/base/env.go

----------------------------------------------------------------------

    package base

    import (

        "os"

    )

    // Функция получения переменной из окружения с возможностью возврата значения по умолчанию

    // Требуется для легкого переключения между запуском сервиса через компьютер пользователя или Docker-контейнер

    func GetEnv(key string, defaultValue string) string {

        if value := os.Getenv(key); value == "" {

            return defaultValue

        } else {

            return value

        }

    }

    // Функция подключается к базе данных, используя драйвер PostgreSQL

    func ConnectToDb() (\*sql.DB, error) {

        // Имя хоста и пользователя базы данных. Используется GetEnv, так как в Docker-контейнере имена переменных могут отличаться

        databaseHost := GetEnv("DATABASE\_HOST", "localhost")

        databaseUser := GetEnv("DATABASE\_USER", "student")

        // Строка подключения

        connection := fmt.Sprintf("postgresql://%s:qwerty@%s:5432/proxy\_db?sslmode=disable", databaseUser, databaseHost)

        var db \*sql.DB

        // Проверка на доступность подключения к бд

        // В самый первый запуск база данных инициализирует себя, поэтому подключение к ней приведет ошибку, так как она еще не готова

        // Этот цикл дает пять секунд на подключение к бд, иначе выдает ошибку и заканчивает работу программы

        connectionAttempt := 0

        attemptsLimit := 5

        for ; connectionAttempt < attemptsLimit; connectionAttempt++ {

            var err error

            db, err = sql.Open("postgres", connection)

            if err != nil {

                LogError(err)

                os.Exit(-1)

            }

            if err := db.Ping(); err != nil {

                LogError(errors.New(

                    "database is not responding; attempts: " + strconv.Itoa(connectionAttempt+1) + "/" + strconv.Itoa(attemptsLimit)))

                time.Sleep(1 \* time.Second)

            } else {

                break

            }

        }

        if connectionAttempt == 5 {

            return nil, errors.New("failed to connect to database")

        }

        return db, nil

    }

1. Перейдите в корневую папку проекта и запустите команду docker-compose build для сборки всех сервисов. Процесс может занять продолжительное время в связи со скачиванием образа базы данных;
2. Введите команду docker-compose up по завершении сборки. В консоли будут отображаться логи запущенных сервисов;
3. Проверьте работу backend сервиса товаров, протестировав API в Postman;
4. Если вы хотите остановить сервисы, нажмите Ctrl + C;
5. Если вы хотите удалить текущие контейнеры, введите docker-compose down
6. Создайте файл **/services/product/frontend/cmd/Dockerfile**

    FROM golang:alpine as builder

    # Директория, путь, в котором находится командная строка

    WORKDIR /app/services/product/frontend/cmd

    # Копирование frontend части

    COPY /services/product/frontend /app/services/product/frontend

    # Копирование пакетов, используемых во frontend

    COPY /services/product/domain /app/services/product/domain

    COPY /services/product/dto /app/services/product/dto

    # Копирование базовых утилит для всех сервисов

    COPY /services/base /app/services/base

    # Копирование файлов модуля

    COPY /go.mod /app

    COPY /go.sum /app

    # Компиляция проекта

    RUN go build -o main .

    # Использование чистого Alpine (дистрибутив Linux).

    FROM alpine

    # Переход в рабочую директорию /app

    WORKDIR /app

    # Скопировать из builder исходный код в текущую рабочую директорию — из /app в .

    COPY --from=builder /app .

    # Переход в директорию с бинарным файлов

    WORKDIR /app/services/product/frontend/cmd

    # Установка порта

    EXPOSE 8081

    # Запуск приложения

    CMD ["./main"]

Добавьте описание frontend части микросервиса товаров в **docker-compose.yml** файл:

  product-front:

    image: documents:product-front

    container\_name: product-front

    build:

      context: .

      dockerfile: ./services/product/frontend/cmd/Dockerfile

    environment:

      - PRODUCT\_BACKEND\_ADDRESS=http://product-back:7071

    volumes:

      - ./static:/app/static

      - ./vendor:/go/pkg/mod

    ports:

      - "8081:8081"

1. Соберите и запустите docker-compose проект.
2. Проверьте работу контейнера, введя в браузере http://localhost:8081/product/products

## Самостоятельное задание

Необходимо реализовать микросервисы организаций, счетов, контрагентов и сотрудников в соответствии с диаграммой БД из лабораторной работы № 1. Микросервисы реализуются аналогично микросервису товаров с такой же структурой папок и последовательностью действий.

Схема распределения портов, а также адреса для описанных микросервисов представлены ниже:



При попытке переключиться на другую вкладку на верхней панели навигации у вас появится ошибка 404. Это связано с тем, что в JS коде для создания панели навигации использованы относительные ссылки, начинающиеся со знака «/». Эти относительные ссылки открывают страницу по тому же порту, на котором работает текущая страница, а на этом порту в программе не заложена обработка иных маршрутов.

Чтобы исправить эту проблему, необходимо запустить сервер, работающий на порту «:80» (на практике это означает, что не придется указывать номер порта после localhost), который будет выступать в роли прокси-сервера.

Прокси-сервер – это сервер, который действует как посредник между клиентом и другим сервером. Когда клиент отправляет запрос на доступ к ресурсу, такому как веб-сайт, прокси-сервер принимает этот запрос и перенаправляет его к запрашиваемому серверу. Затем запрашиваемый сервер отвечает на запрос прокси-сервера, который в свою очередь передает ответ обратно клиенту.

«Proxy» в переводе с английского означает «доверенность», поэтому это слово используется и в БД, и в рамках кода. Пусть прокси-сервер будет располагаться в директории **/services/home/**.

Создайте аналогичную структуру:

services

  ├── base

  └── home

      ├── cmd

      │   ├── Dockerfile

      │   └── main.go

      └── router

        └── router.go

Код, реализующий прокси-сервер, представлен в файле **router.go**

    package router

    import (

        "net/http"

        "net/http/httputil"

        "net/url"

        "github.com/gorilla/mux"

        "github.com/joyzem/documents/services/base"

    )

    // Перенаправляет с маршрута "path" на тот же маршрут, но на другом адресе "address"

    // Так, если относительная ссылка "/account" будет вызвана внутри этого сервера,

    // то он перенаправит запрос на "http://localhost:8083/account"

    func handleWithProxy(router \*mux.Router, path string, address string) {

        router.PathPrefix(path).HandlerFunc(func(w http.ResponseWriter, r \*http.Request) {

            url, \_ := url.Parse(address)

            proxy := httputil.NewSingleHostReverseProxy(url)

            proxy.ServeHTTP(w, r)

        })

    }

    func GetRouter() http.Handler {

        router := mux.NewRouter()

        // Получение адреса товаров из переменной окружения или же значение по умолчанию

        productAddress := base.GetEnv("PRODUCT\_ADDRESS", "http://localhost:8081")

        handleWithProxy(router, "/product", productAddress)

        organizationAddress := base.GetEnv("ORGANIZATION\_ADDRESS", "http://localhost:8082")

        handleWithProxy(router, "/organization", organizationAddress)

        accountAddress := base.GetEnv("ACCOUNT\_ADDRESS", "http://localhost:8083")

        handleWithProxy(router, "/account", accountAddress)

        employeeAddress := base.GetEnv("EMPLOYEE\_ADDRESS", "http://localhost:8084")

        handleWithProxy(router, "/employee", employeeAddress)

        customerAddress := base.GetEnv("CUSTOMER\_ADDRESS", "http://localhost:8085")

        handleWithProxy(router, "/customer", customerAddress)

// Прокси для документов

        proxyAddress := base.GetEnv("DOCUMENTS\_ADDRESS", "http://localhost:8086")

        handleWithProxy(router, "/documents", proxyAddress)

// Стили и скрипты

    router.PathPrefix("/static/").Handler(http.StripPrefix("/static/", http.FileServer(http.Dir("../../../static"))))

        return router

    }

Файл **/services/home/cmd/main.go**:

    package main

    import (

        "fmt"

        "net/http"

        "os"

        "github.com/joyzem/documents/home/router"

        "github.com/joyzem/documents/services/base"

    )

    func main() {

        hanlder := router.GetRouter()

        fmt.Println("Listening on 80...")

        if err := http.ListenAndServe(":80", hanlder); err != nil {

            base.LogError(err)

            os.Exit(-1)

        }

    }

Dockerfile:

    FROM golang:alpine as builder

    WORKDIR /app/services/home/cmd

    COPY /services/home /app/services/home

    COPY /services/base /app/services/base

    COPY go.mod /app

    COPY go.sum /app

    RUN go build -o main .

    FROM alpine

    WORKDIR /app

    COPY --from=builder /app .

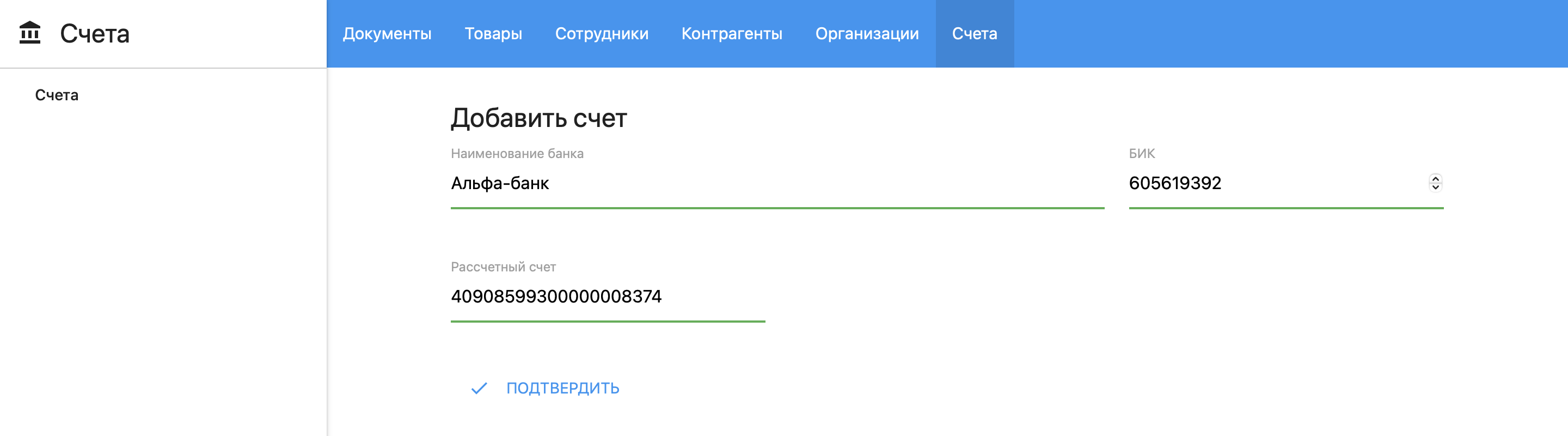
    WORKDIR /app/services/home/cmd

    EXPOSE 80

    CMD ["./main"]

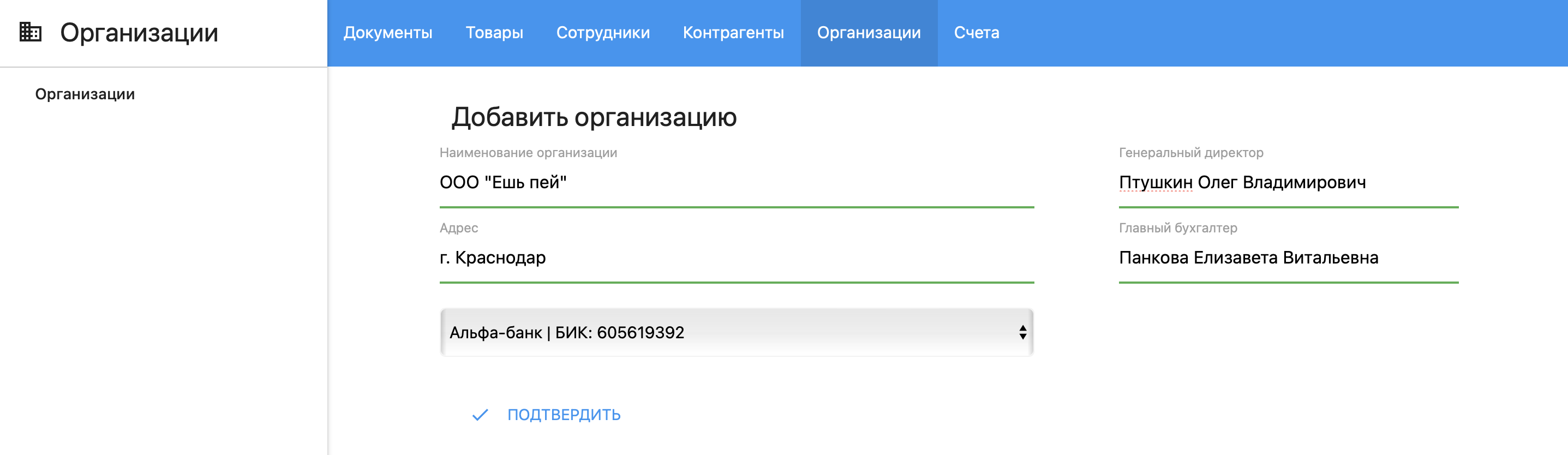
Реализуйте frontend согласно макетам:

**Микросервис счетов:**



БИК – 9 цифр

**Микросервис организаций:**



Frontend-часть микросервиса организаций обращается к микросервису счетов. В чистой архитектуры не стоит импортировать модули из других микросервисов – каждый микросервис создаёт свою модель на основе возвращаемого DTO, однако в учебном проекте этим можно пренебречь.

/services/organization/frontend/handlers/handlers.go

----------------------------------------------------------------------

    import (

// ...

        accountsDomain "github.com/joyzem/documents/services/account/domain"

        accountsDto "github.com/joyzem/documents/services/account/dto"

    )

// ...

    func CreateOrganizationHandler(w http.ResponseWriter, r \*http.Request) {

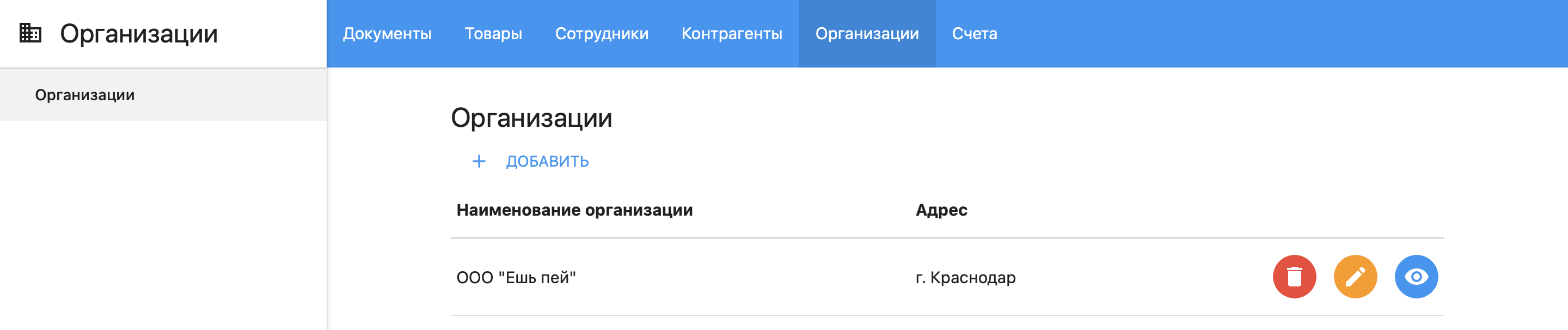
        accountsUrl := fmt.Sprintf("%s/accounts", utils.GetAccountsAddress())

        resp, \_ := grequests.Get(accountsUrl, nil)

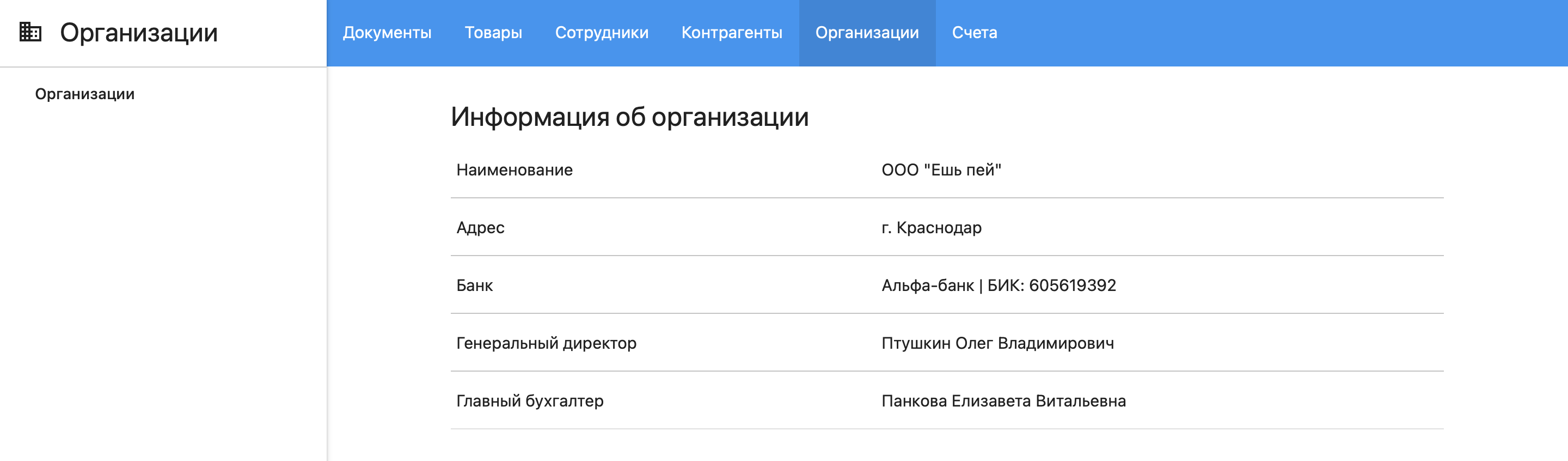
        var data accountsDto.GetAccountsResponse

// ...

Рядом с кнопками редактировать и удалить должна быть кнопка для просмотра детальной информации:



Иконка для просмотра детальной информации называется «visibility»



Адрес данной страницы: **http://localhost/organization/organizations  
/detail/{id}**

HTML-код представляет собой просто таблицу:

<table class="highlight">

    <tr>

             <td>Наименование</td>

             <td>{{ .Name }}</td>

        </tr>

        <tr>

             <td>Адрес</td>

             <td>{{ .Address }}</td>

        </tr>

        <tr>

             <td>Банк</td>

             <td>{{ .BankName }} | БИК: {{ .BankIdentityNumber }}</td>

        </tr>

        <tr>

             <td>Генеральный директор</td>

             <td>{{ .Chief}}</td>

        </tr>

        <tr>

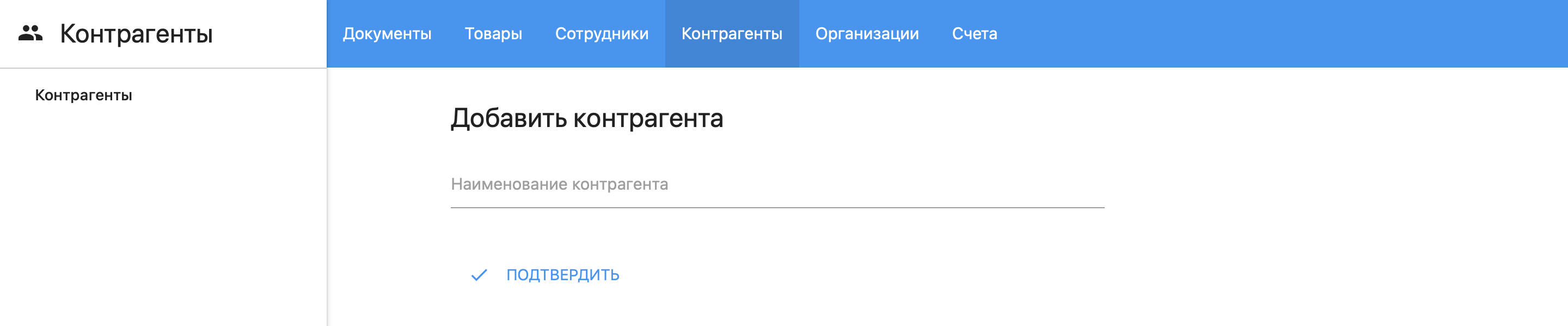
        <td>Главный бухгалтер</td>

        <td>{{ .FinancialChief }}</td>

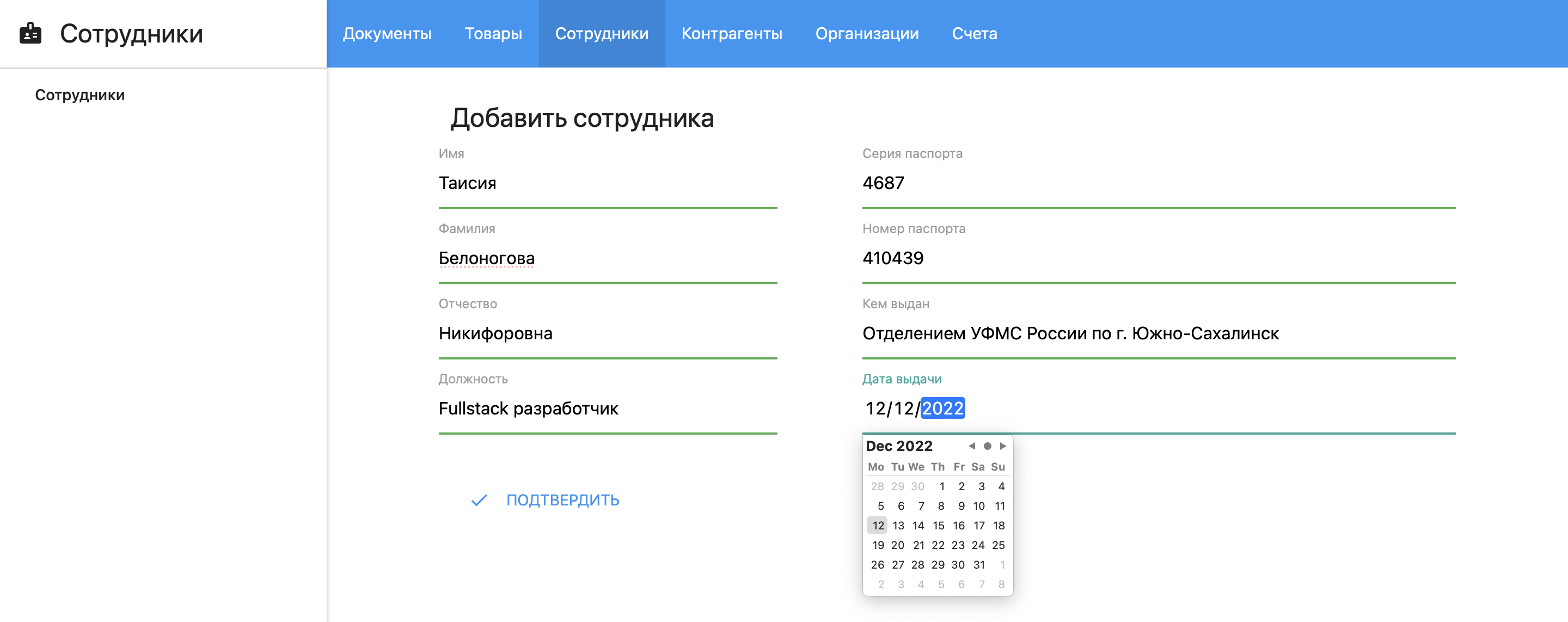
        </tr>

    </table>

**Микросервис контрагентов:**



**Микросервис сотрудников:**



Ввод даты осуществляется с помощью стандартного <input type="date">.

При создании структуры Employee для даты используйте тип string. При получении объекта в **implementation/repo** база данных выдает дату в формате "2022-12-12T00:00:00Z". Чтобы получить дату в корректном формате, используйте следующий код:

    parsedTime, err := base.ParseTime(employee.PassportDateOfIssue)

    employee.PassportDateOfIssue = parsedTime

/services/base/time.go

----------------------------------------------------------------------

    func ParseTime(source string) (string, error) {

        parsedTime, err := time.Parse(time.RFC3339, source)

        if err == nil {

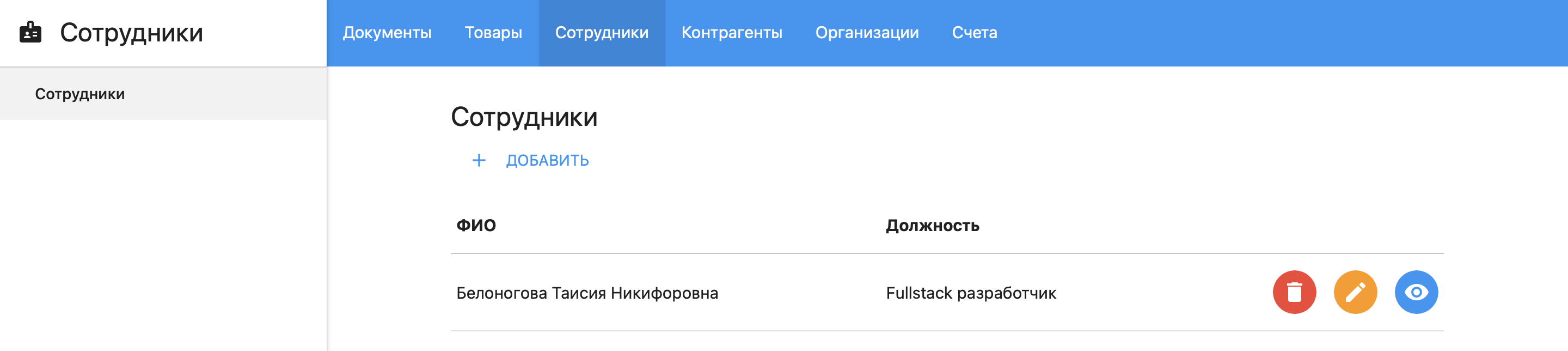
            return parsedTime.Format("2006-01-02"), nil

        }

        return "", err

    }

Аналогично организациям, список сотрудников также содержит лишь краткую информацию.



# Лабораторная работа № 6. Микросервис доверенностей.

## Цель

Добавить в систему документ «Доверенность».

## Отношение «Многие ко многим»

Отношение «многие ко многим» (many-to-many) в базах данных – это отношение, при котором каждый элемент одной таблицы может быть связан с несколькими элементами другой таблицы, и наоборот. В таком отношении каждая запись в таблице **A** может соответствовать нескольким записям в таблице **B**, и наоборот, каждая запись в таблице **B** может соответствовать нескольким записям в таблице **A**. Для реализации отношения «многие ко многим» необходимо использовать промежуточную таблицу, которая будет связывать записи из двух основных таблиц.

Например, в учебном пособии по базам данных можно рассмотреть отношение «многие ко многим» между таблицами "Студенты" и "Курсы". Каждый студент может записаться на несколько курсов, а каждый курс может быть выбран несколькими студентами. Для связи этих таблиц необходимо создать промежуточную таблицу "Запись на курс", которая будет содержать в себе данные о том, какой студент записался на какой курс.

Примером отношения «многие ко многим» в документе "Доверенность" может служить список товаров, перечисленных в «теле» доверенности. Один и тот же товар может быть указан в нескольких доверенностях, а каждая доверенность может содержать несколько товаров. Для связи этих данных необходимо создать промежуточную таблицу "Товары в доверенности", которая будет содержать информацию о том, какой товар указан в какой доверенности.

В лабораторной работе № 1 Вы реализовали отношение «многие ко многим» через таблицу «proxy\_bodies» в следующем фрагменте:



## Создание backend-части

Создание микросервиса доверенностей не сильно отличается по принципу, но есть некоторые аспекты, на которые необходимо обратить внимание.

### Domain-cлой

В этом слое представлено три структуры:

Структура ProxyBodyItem содержит информацию о каждом товаре, представленном в теле доверенности.

/services/proxy/domain/proxy-body.go

----------------------------------------------------------------------

    // Тело доверенности. Каждая строка с товаром представлена этой структурой

    type ProxyBodyItem struct {

        Id            int `json:"id"`

        ProductId     int `json:"product\_id"`

        ProxyId       int `json:"proxy\_id"`

        ProductAmount int `json:"product\_amount"`

    }

Структура ProxyHeader содержит информацию о шапке доверенности, такую как номер, id организации, контрагента, сотрудника, дату выдачи и дату истечения срока действия.

/services/proxy/domain/proxy-header.go

----------------------------------------------------------------------

    type ProxyHeader struct {

        Id             int    `json:"id"`

        OrganizationId int    `json:"organization\_id"`

        CustomerId     int    `json:"customer\_id"`

        EmployeeId     int    `json:"employee\_id"`

        DateOfIssue    string `json:"date\_of\_issue"`

        IsValidUntil   string `json:"is\_valid\_until"`

    }

Структура Proxy содержит шапку доверенности и список товаров, представленных в теле доверенности.

/services/proxy/domain/proxy.go

----------------------------------------------------------------------

    type Proxy struct {

        ProxyHeader    ProxyHeader     `json:"proxy\_header"`

        ProxyBodyItems []ProxyBodyItem `json:"proxy\_body\_items"`

    }

### Интерфейс сервиса.

Сервис предоставляет пользователям следующий функционал:

* Создание шапки доверенности
* Добавление строки тела доверенности
* Получение доверенностей
* Получение доверенности по id
* Обновление шапки доверенности
* Удаление доверенности
* Удаление строки тела доверенности

    type ProxyService interface {

        CreateProxyHeader(organizationId int, customerId int, employeeId int, dateOfIssue string, isValidUntil string) (\*domain.ProxyHeader, error)

        CreateProxyBodyItem(productId int, proxyId int, productAmount int) (\*domain.Proxy, error)

        GetProxies() ([]domain.ProxyHeader, error)

        ProxyById(id int) (\*domain.Proxy, error)

        UpdateProxyHeader(domain.ProxyHeader) (\*domain.Proxy, error)

        DeleteProxy(id int) error

        DeleteProxyBodyItem(itemId int) error

    }

### Транспортный уровень и репозиторий

Маршруты backend-части представлены в таблице:

|  |  |  |
| --- | --- | --- |
| Функция | Метод | Маршрут |
| Создание шапки доверенности | POST | /proxy |
| Получение доверенностей | GET | /proxy |
| Получение доверенности по id | GET | /proxy/{id:[0-9]+} |
| Обновление шапки доверенности | PUT | /proxy |
| Добавление строки тела доверенности | POST | /proxy/body |
| Удаление строки тела доверенности | DELETE | /proxy/body |

Процесс реализации репозитория практически ничем не отличается от предыдущих микросервисов. Единственная функция, которая будет рассмотрена – получение доверенности по id. После запроса на получение шапки доверенности выполняется запрос на получение тела доверенности.

    func (r \*proxyRepo) ProxyById(id int) (\*domain.Proxy, error) {

        sql := `

            SELECT \* FROM proxies WHERE id = $1

        `

        proxy := domain.Proxy{}

        if err := r.db.QueryRow(sql, id).Scan(

            &proxy.ProxyHeader.Id,

            &proxy.ProxyHeader.OrganizationId,

            &proxy.ProxyHeader.CustomerId,

            &proxy.ProxyHeader.EmployeeId,

            &proxy.ProxyHeader.DateOfIssue,

            &proxy.ProxyHeader.IsValidUntil,

        ); err != nil {

            return nil, err

        }

        proxy.ProxyHeader.DateOfIssue, \_ = base.ParseTime(proxy.ProxyHeader.DateOfIssue)

        proxy.ProxyHeader.IsValidUntil, \_ = base.ParseTime(proxy.ProxyHeader.IsValidUntil)

        getBodySql := `SELECT \* FROM proxy\_bodies WHERE proxy\_id = $1`

        rows, err := r.db.Query(getBodySql, id)

        if err != nil {

            return nil, err

        }

        for rows.Next() {

            bodyItem := domain.ProxyBodyItem{}

            if err := rows.Scan(

                &bodyItem.Id,

                &bodyItem.ProductId,

                &bodyItem.ProxyId,

                &bodyItem.ProductAmount,

            ); err != nil {

                return nil, err

            }

            proxy.ProxyBodyItems = append(proxy.ProxyBodyItems, bodyItem)

        }

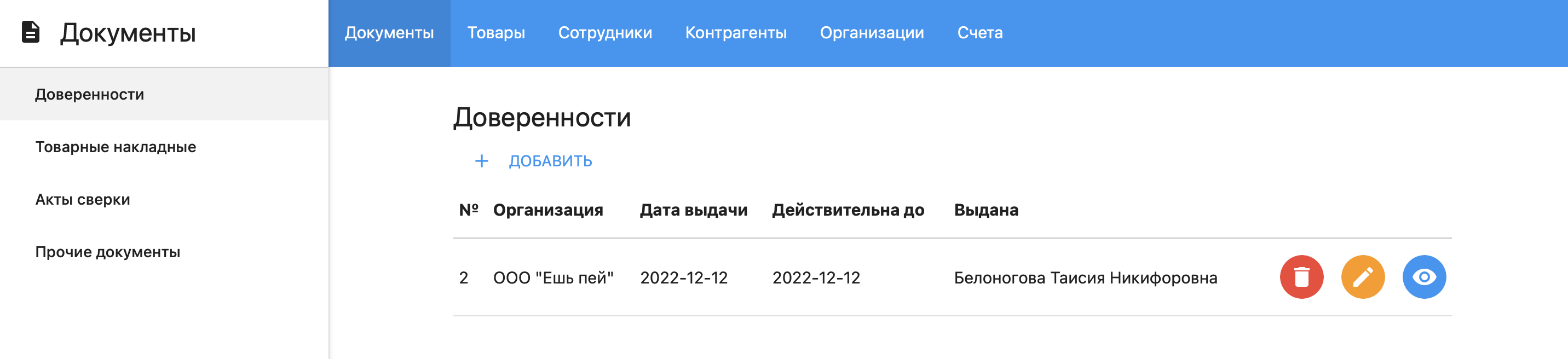
        return &proxy, nil

    }

Оставшаяся backend-часть реализуется аналогично другим микросервисам. Порт, на котором работает backend-часть: 7076.

## Создание frontend-части

Рассмотрим структуру вкладки «Документы»:



В боковом меню навигации будут перечислены имеющиеся в системе документы. По нажатию на боковой элемент откроется страница с перечислением краткой информации по документам. Каждый документ можно отредактировать, удалить или открыть в режиме просмотра. Сверху находится кнопка «Добавить».

Для документа «Доверенность» используются следующие маршруты:

/services/proxy/frontend/router/router.go

----------------------------------------------------------------------

    router.HandleFunc("/documents/proxies", handlers.ProxiesHandler)

    router.HandleFunc("/documents/proxies/create", handlers.CreateProxyGetHandler).Methods(http.MethodGet)

    router.HandleFunc("/documents/proxies/create", handlers.CreateProxyPosthandler).Methods(http.MethodPost)

    router.HandleFunc("/documents/proxies/update/{id:[0-9]+}", handlers.UpdateProxyGetHandler).Methods(http.MethodGet)

    router.HandleFunc("/documents/proxies/update", handlers.UpdateProxyPostHandler).Methods(http.MethodPost)

    router.HandleFunc("/documents/proxies/delete", handlers.DeleteProxyHandler).Methods(http.MethodPost)

    router.HandleFunc("/documents/proxies/details/{id:[0-9]+}", handlers.ProxyDetailsHandler).Methods(http.MethodGet)

    router.HandleFunc("/documents/proxies/{id:[0-9]+}/body/create", handlers.CreateProxyBodyGetHandler).Methods(http.MethodGet)

    router.HandleFunc("/documents/proxies/body/create", handlers.CreateProxyBodyPostHandler).Methods(http.MethodPost)

    router.HandleFunc("/documents/proxies/body/delete", handlers.DeleteProxyBodyHandler).Methods(http.MethodPost)

    router.PathPrefix("/static/").Handler(http.StripPrefix("/static/", http.FileServer(http.Dir("../../../../static"))))

    router.PathPrefix("/documents/static/").Handler(http.StripPrefix("/documents/static/", http.FileServer(http.Dir("../../../documents/static"))))

Боковое меню должно быть одинаковым для всех документов, поэтому его инициализация лежит в файле **/services/documents/static/js/sidenav-documents.js**.

Особенность микросервиса доверенностей состоит в том, что он обращается за данными практически ко всем микросервисам. В этом и состоит его сложность. Обработчик каждого запроса занимает много строк кода, так как приходится брать данные из всех остальных микросервисов и шаблонизировать их для страниц.

### Определение структур шаблонов

**Список всех доверенностей**



Адрес: /documents/proxies

Описание: список всех доверенностей. Требуется название организации, дата выдачи и дата «действительна до», а также, кому выдана доверенность. Следующая структура передается в виде списка:

type proxyTemplate struct {

    Id           int

    Organization string

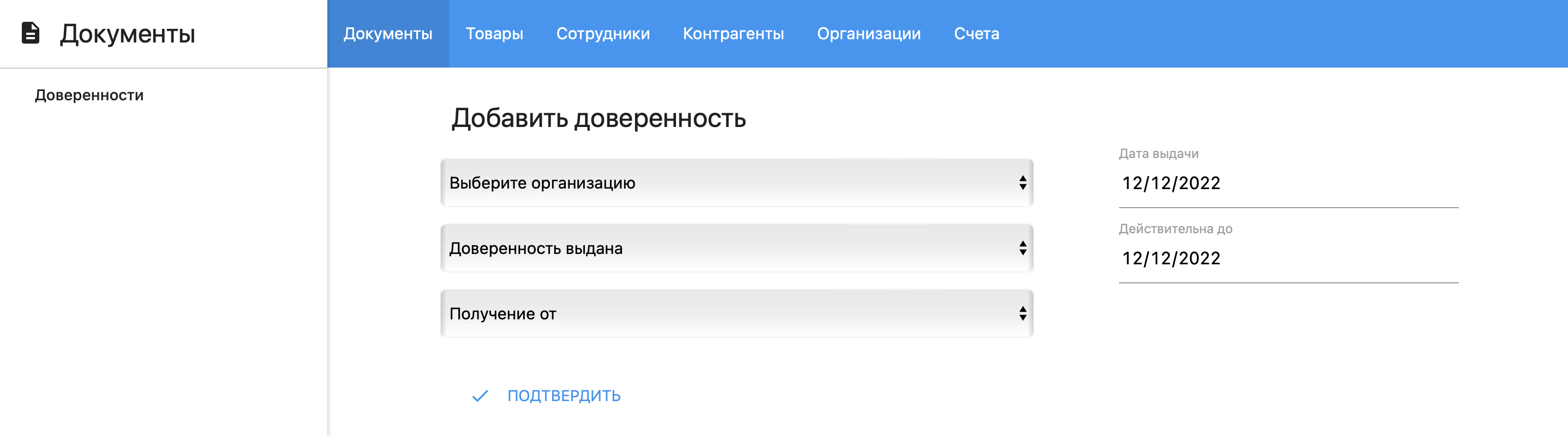
    DateOfIssue  string

    IsValidUntil string

    Employee     string

}

**Создание доверенности**



Адрес: /documents/proxies/create

Описание: экран создания шапки доверенности. Требуется выбрать организацию, сотрудника, которому выдали доверенность, и контрагента – «получение от». Также для отправки формы требуется ввести дату выдачи и «действительна до».

type proxyTemplate struct {

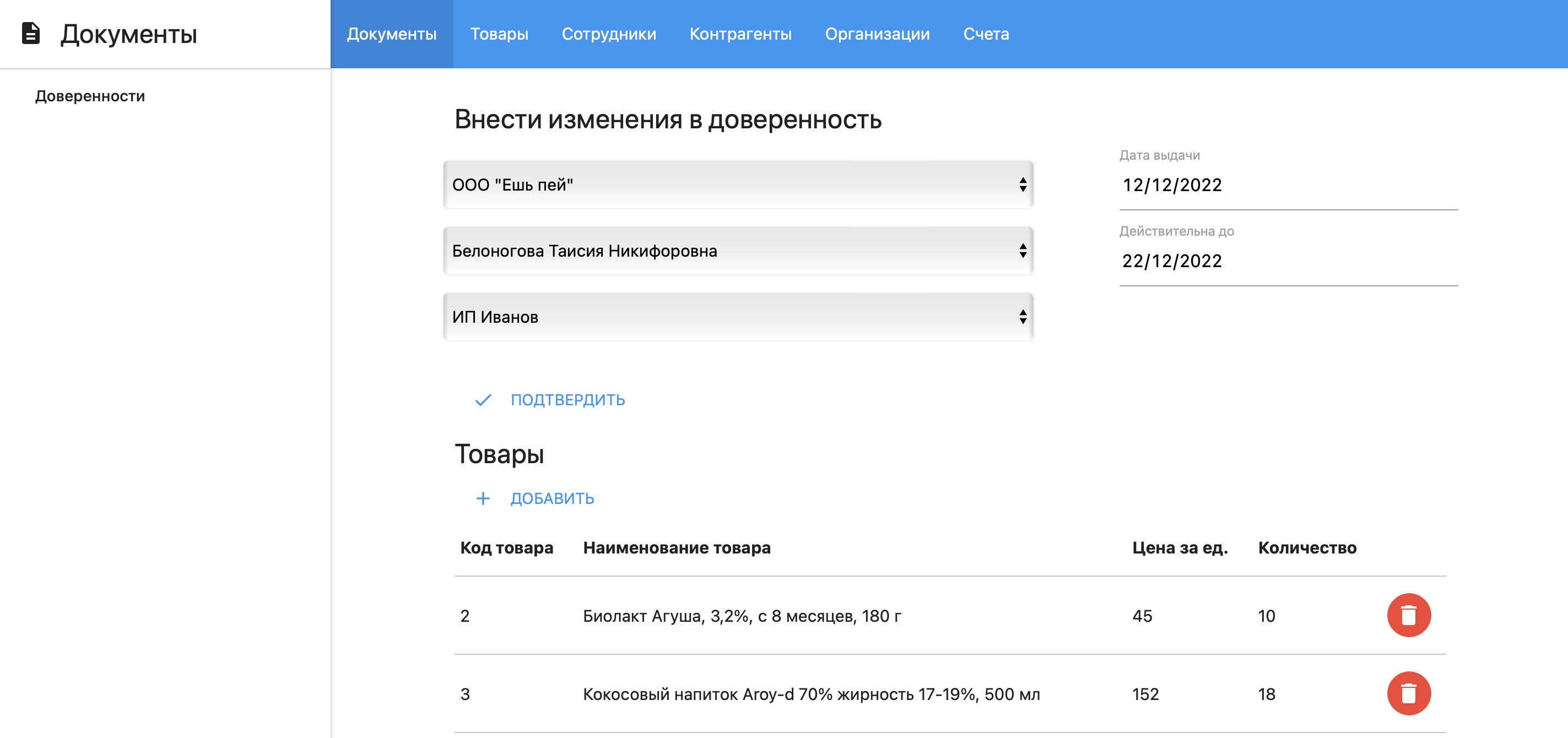
Organizations []organizationDomain.Organization

Employees []employeeDomain.Employee

Customers []customerDomain.Customer

}

**Внесение изменений в доверенность**

****

Адрес: /documents/proxies/update/{id:[0-9]}

Описание: на этом экране требуются данные о текущей доверенности (наименование организации, товары и т.д.), а также информация об организациях, сотрудниках и контрагентах (как на прошлом экране).

    type proxyBodyTemplate struct {

        Id            int

        Name          string

        Price         int

        ProductAmount int

    }

    type updateProxyTemplate struct {

        Proxy          domain.Proxy

        Organizations  []organizationDomain.Organization

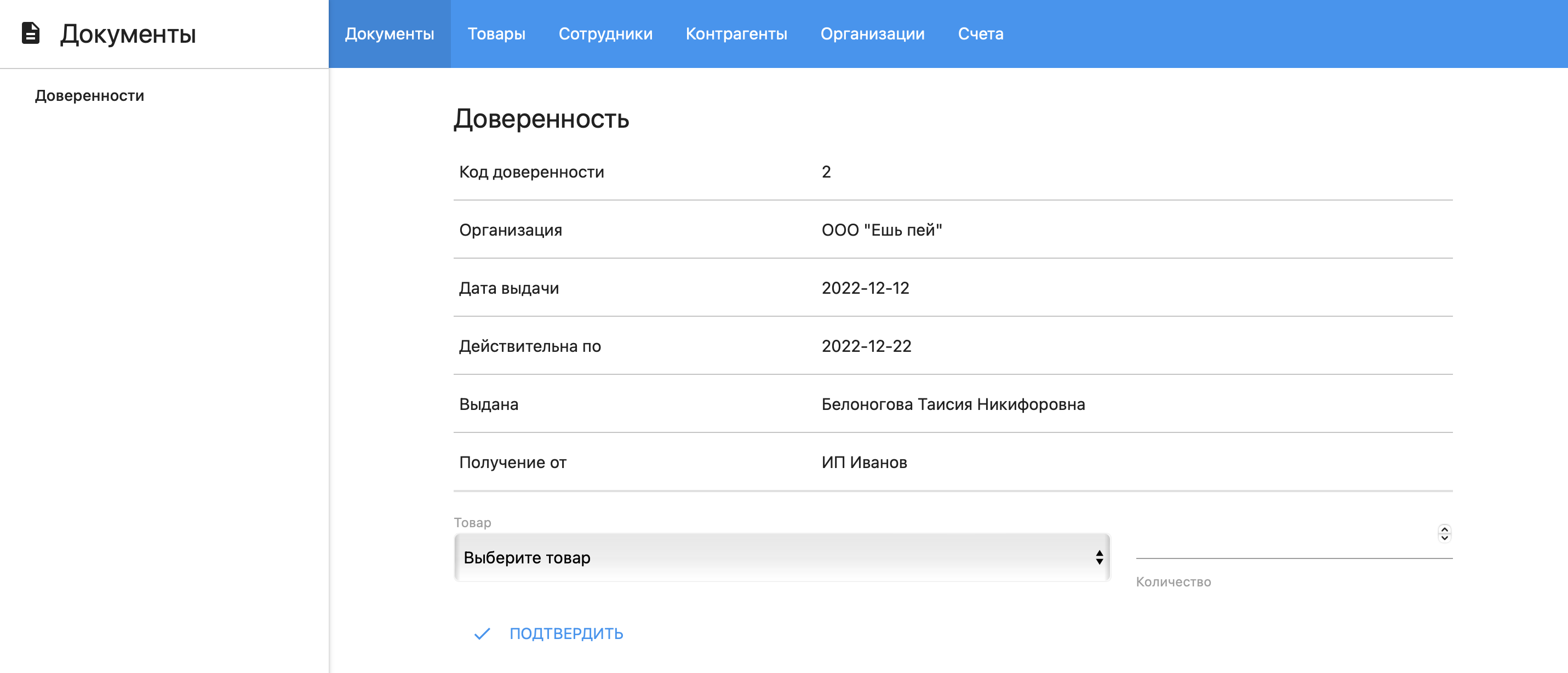
        Employees      []employeeDomain.Employee

        Customers      []customerDomain.Customer

        ProxyBodyItems []proxyBodyTemplate

    }

**Добавление товара в тело доверенности**



Адрес: /documents/proxies/2/body/create

Описание: на экране отображена краткая информация по доверенности. Под описанием располагается выпадающее меню с товарами, а рядом – количество.

    type proxyDetail struct {

        Id           int

        Organization string

        Employee     string

        Customer     string

        DateOfIssue  string

        IsValidUntil string

    }

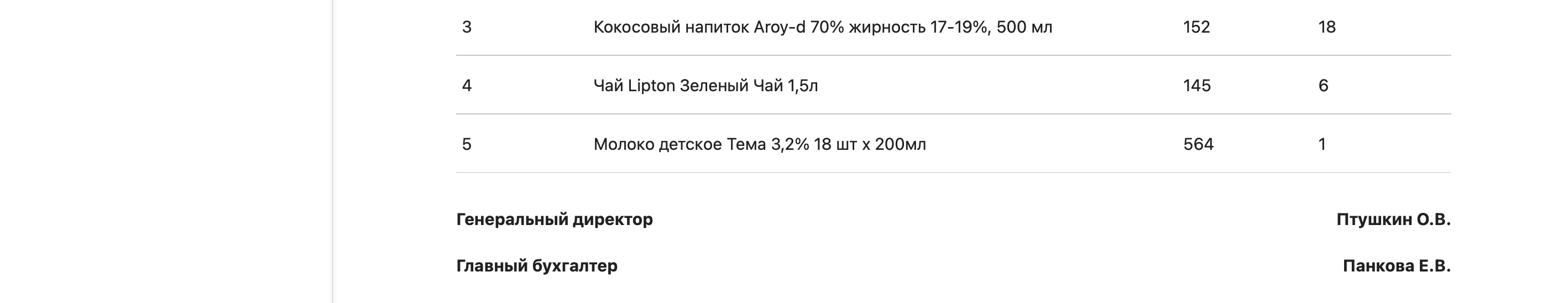
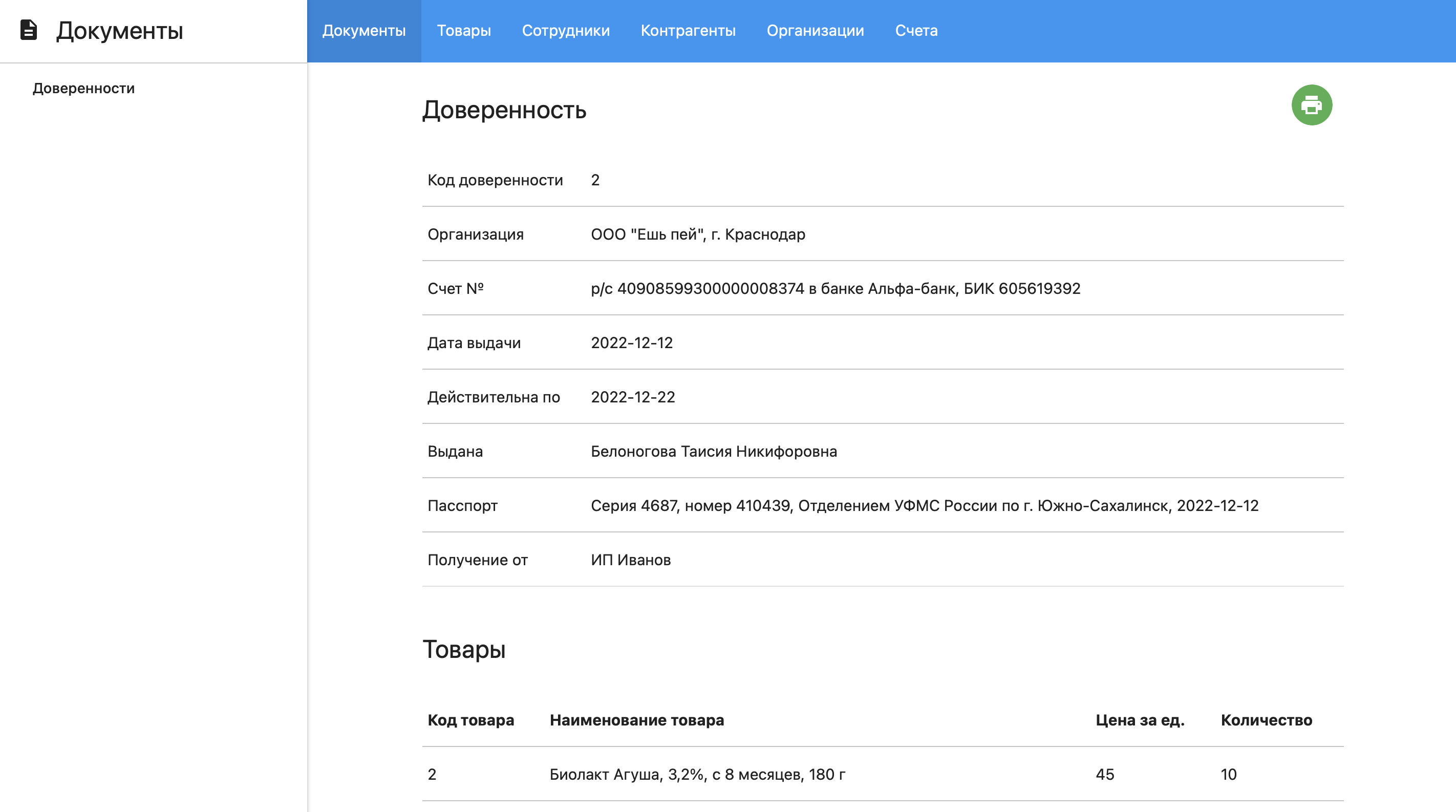
    type createProxyBodyTemplate struct {

        Proxy    proxyDetail

        Products []productsDomain.Product

    }

**Детальная информация по доверенности**



Адрес: /documents/proxies/details/{id:[0-9]+}

Описание: отображает подробную отформатированную информацию по всей доверенности.

    type proxyBodyTemplate struct {

        ProductId     int

        Name          string

        Price         int

        ProductAmount int

    }

    type proxyDetailTemplate struct {

        ProxyId             int

        Organization        string

        Address             string

        Account             string

        BankName            string

        BankIdentityNumber  string

        Employee            string

        PassportSeries      string

        PassportNumber      string

        PassportIssuedBy    string

        PassportDateOfIssue string

        Customer            string

        Chief               string

        FinancialChief      string

        DateOfIssue         string

        IsValidUntil        string

        ProxyBodyItems      []proxyBodyTemplate

    }

### Написание HTML-страниц

В основном, все страницы повторяют макеты из других микросервисов. Единственное, что здесь добавилось ­– кнопка печати доверенности. Реализовать это очень просто. В первую очередь, необходимо написать JavaScript функцию:

/static/js/print.js

-------------------------------------------------------------------------

    function printElementById(id) {

        var printContents = document.getElementById(id).innerHTML;

        var originalContents = document.body.innerHTML;

        document.body.innerHTML = printContents;

        window.print();

        document.body.innerHTML = originalContents;

    }

Добавить в файл **styles.css** следующие стили:

    @media print {

        .print-only {

            display: block;

        }

        .not-print {

            display: none;

        }

        body.has-fixed-sidenav {

            left: 0 !important;

            padding-left: 0 !important;

        }

    }

Затем в разметке добавить тегу main id "print-area" и класс "print-only". Кнопке печати добавить класс "not-print"

    <body class="has-fixed-sidenav">

        <header>

        </header>

        <main id="print-area" class="white print-only">

            <div class="container">

                <div class="section">

                    <div class="row valign-wrapper">

                        <h5>Доверенность</h5>

                        <div class="col s12 right-align not-print">

                            <button class="btn-floating z-depth-0 waves-effect waves-light"

                                onclick="printElementById('print-area')">

                                <i class="material-icons green">print</i>

                            </button>

                        </div>

                    </div>

                    <table class="highlight">

                        <tr>

                            <td>Код доверенности</td>

                            <td>{{ .ProxyId }}</td>

                        </tr>

...

### Написание вспомогательных функций

Создайте файл **/services/proxy/frontend/utils/backend.go** и добавьте в него функции получения адресов согласно схеме из пятой лабораторной работы. Назовите функции в формате GetProxiesAddress. Эти функции вызывают base.GetEnv(PROXIES\_ENV, "http://localhost:7076")

    const (

        PROXIES\_ENV       = "PROXY\_BACKEND\_ADDRESS"

        EMPLOYEES\_ENV     = "EMPLOYEE\_BACKEND\_ADDRESS"

        PRODUCTS\_ENV      = "PRODUCT\_BACKEND\_ADDRESS"

        ORGANIZATIONS\_ENV = "ORGANIZATION\_BACKEND\_ADDRESS"

        CUSTOMERS\_ENV     = "CUSTOMER\_BACKEND\_ADDRESS"

        ACCOUNTS\_ENV      = "ACCOUNT\_BACKEND\_ADDRESS"

    )

Получение доверенностей, товаров, организаций, сотрудников и контрагентов происходит в нескольких местах, поэтому реализуйте следующие функции в том же пакете utils:

    func GetProxyById(id int) (\*dto.ProxyByIdResponse, error)

func GetProducts() (\*productDto.GetProductsResponse, error)

func GetProductById(id int) (\*productDto.ProductByIdResponse, error)

func GetOrganizationById(id int) (\*organizationDto.OrganizationByIdResponse, error)

    func GetOrganizations() ([]organizationDto.GetOrganizationsResponse, error)

    func GetCustomers() (customerDto.GetCustomersResponse, error)

    func GetCustomerById(id int) (customerDto.CustomerByIdResponse, error)

    func GetEmployeeById(id int) (\*employeeDto.EmployeeByIdResponse, error)

    func GetEmployees() (employeeDto.GetEmployeesResponse, error)

Это значительно повысит читабельность кода и уменьшит количество ошибок.

### Написание обработчиков

Обработчики запросов кажутся большими и сложными, но их алгоритм достаточно прост, и они повторяют друг друга во многом. Для уменьшения шаблонного кода проверка ошибок была опущена. Рассмотрим два самых больших обработчика в этом микросервисе.

**Страница обновления доверенности**

    func UpdateProxyGetHandler(w http.ResponseWriter, r \*http.Request) {

        // Получение доверенности

        id, \_ := strconv.Atoi(mux.Vars(r)["id"])

        proxy, \_ := utils.GetProxyById(id)

        if proxy.Err != "" {

            http.Error(w, proxy.Err, http.StatusInternalServerError)

            return

        }

        // Создание шаблонов для HTML-страницы

        type proxyBodyTemplate struct {

            Id            int

            Name          string

            Price         int

            ProductAmount int

        }

        type updateProxyTemplate struct {

            Proxy          domain.Proxy

            Organizations  []organizationDomain.Organization

            Employees      []employeeDomain.Employee

            Customers      []customerDomain.Customer

            ProxyBodyItems []proxyBodyTemplate

        }

        templateData := updateProxyTemplate{

            Proxy: \*proxy.Proxy,

        }

        // Цикл по товарам из доверенности, чтобы

        // заполнить тело доверенности

        proxyBodyItems := []proxyBodyTemplate{}

        for \_, bodyItem := range proxy.Proxy.ProxyBodyItems {

            // Получение товара

            product, \_ := utils.GetProductById(bodyItem.ProductId)

            if product.Err != "" {

                base.LogError(errors.New(product.Err))

            } else {

                // Добавление в список товаров

                proxyBodyItems = append(proxyBodyItems, proxyBodyTemplate{

                    Id:            product.Product.Id,

                    Name:          product.Product.Name,

                    Price:         product.Product.Price,

                    ProductAmount: bodyItem.ProductAmount,

                })

            }

        }

        // Инициализация тела доверенности

        templateData.ProxyBodyItems = proxyBodyItems

        // Получение организаций и инициализация в шаблоне

        organizations, \_ := utils.GetOrganizations()

        if organizations.Err != "" {

            base.LogError(errors.New(organizations.Err))

        } else {

            templateData.Organizations = organizations.Organizations

        }

        // Инициализация аналогично организациям

        employees, \_ := utils.GetEmployees()

        if employees.Err != "" {

            base.LogError(errors.New(employees.Err))

        } else {

            templateData.Employees = employees.Employees

        }

        // Инициализация контрагентов

        customers, \_ := utils.GetCustomers()

        if customers.Err != "" {

            base.LogError(errors.New(customers.Err))

        } else {

            templateData.Customers = customers.Customers

        }

        // Выполнение шаблона

        tmpl, \_ := template.ParseFiles("../static/html/update-proxy.html")

        tmpl.Execute(w, templateData)

    }

**Просмотр доверенности**

    func ProxyDetailsHandler(w http.ResponseWriter, r \*http.Request) {

        // Получение доверенности

        id, \_ := strconv.Atoi(mux.Vars(r)["id"])

        proxy, \_ := utils.GetProxyById(id)

        if proxy.Err != "" {

            http.Error(w, proxy.Err, http.StatusInternalServerError)

            return

        }

        // Создание структур для шаблона

        type proxyBodyTemplate struct {

            ProductId     int

            Name          string

            Price         int

            ProductAmount int

        }

        type proxyDetailTemplate struct {

            ProxyId             int

            Organization        string

            Address             string

            Account             string

            BankName            string

            BankIdentityNumber  string

            Employee            string

            PassportSeries      string

            PassportNumber      string

            PassportIssuedBy    string

            PassportDateOfIssue string

            Customer            string

            Chief               string

            FinancialChief      string

            DateOfIssue         string

            IsValidUntil        string

            ProxyBodyItems      []proxyBodyTemplate

        }

        // Инициализация данных по доверенности

        templateData := proxyDetailTemplate{

            ProxyId:      proxy.Proxy.ProxyHeader.Id,

            DateOfIssue:  proxy.Proxy.ProxyHeader.DateOfIssue,

            IsValidUntil: proxy.Proxy.ProxyHeader.IsValidUntil,

        }

        // Заполнение списка товаров

        proxyBodyItems := []proxyBodyTemplate{}

        for \_, bodyItem := range proxy.Proxy.ProxyBodyItems {

            product, \_ := utils.GetProductById(bodyItem.ProductId)

            if product.Err != "" {

                base.LogError(errors.New(product.Err))

            } else {

                proxyBodyItems = append(proxyBodyItems, proxyBodyTemplate{

                    ProductId:     product.Product.Id,

                    Name:          product.Product.Name,

                    Price:         product.Product.Price,

                    ProductAmount: bodyItem.ProductAmount,

                })

            }

        }

        templateData.ProxyBodyItems = proxyBodyItems

        // Получение данных об организации

        organization, \_ := utils.GetOrganizationById(proxy.Proxy.ProxyHeader.OrganizationId)

        if organization.Err != "" {

            base.LogError(errors.New(organization.Err))

            templateData.Organization = fmt.Sprintf("Ошибка: %s", organization.Err)

        } else {

            templateData.Organization = organization.Organization.Name

            templateData.Address = organization.Organization.Address

            // Получение инициалов имени – Фамилия И.О.

            templateData.Chief = utils.NameInitials(organization.Organization.Chief)

            templateData.FinancialChief = utils.NameInitials(organization.Organization.FinancialChief)

        }

        // Получение данных о счёте

        if organization.Organization != nil {

            accountUrl := fmt.Sprintf("%s/accounts/%d", utils.GetAccountsAddress(), organization.Organization.AccountId)

            accountResp, \_ := grequests.Get(accountUrl, &grequests.RequestOptions{

                JSON: accountDto.AccountByIdRequest{

                    Id: organization.Organization.AccountId,

                }})

            var account accountDto.AccountByIdResponse

            accountResp.JSON(&account)

            if account.Err != "" {

                templateData.Account = fmt.Sprintf("Ошибка: %s", account.Err)

            } else {

                templateData.Account = account.Account.Account

                templateData.BankName = account.Account.BankName

                templateData.BankIdentityNumber = account.Account.BankIdentityNumber

            }

        }

        // Данные о сотруднике, которому выдали доверенность

        employee, \_ := utils.GetEmployeeById(proxy.Proxy.ProxyHeader.EmployeeId)

        if employee.Err != "" {

            base.LogError(errors.New(employee.Err))

            templateData.Employee = fmt.Sprintf("Ошибка: %s", employee.Err)

        } else {

            templateData.Employee = utils.Fullname(

                employee.Employee.LastName,

                employee.Employee.FirstName,

                employee.Employee.MiddleName,

            )

            templateData.PassportSeries = employee.Employee.PassportSeries

            templateData.PassportNumber = employee.Employee.PassportNumber

            templateData.PassportDateOfIssue = employee.Employee.PassportDateOfIssue

            templateData.PassportIssuedBy = employee.Employee.PassportIssuedBy

        }

        // Данные о контрагенте

        customer, \_ := utils.GetCustomerById(proxy.Proxy.ProxyHeader.CustomerId)

        if customer.Err != "" {

            base.LogError(errors.New(customer.Err))

            templateData.Customer = fmt.Sprintf("Ошибка: %s", customer.Err)

        } else {

            templateData.Customer = customer.Customer.Name

        }

        // Выполнение шаблона

        tmpl, \_ := template.ParseFiles("../static/html/proxy-details.html")

        tmpl.Execute(w, templateData)

    }

## Самостоятельное задание

Завершите написание микросервиса доверенностей и добавьте его в **docker-compose.yml**.

Добавьте в систему ещё один документ по указанию преподавателя.

# Заключение

В рамках данной работы была разработана система учета документов. В результате были получены навыки работы с Golang, Docker, HTML, Materialize CSS, JavaScript, SQL, PostgreSQL, Postman, HTTP, REST API и микросервисной архитектурой, что может быть хорошим началом для дальнейшего профессионального развития в области программной инженерии и разработки программного обеспечения.

Fullstack разработчик – это специалист, который может работать со всеми компонентами веб-приложения, включая серверную и клиентскую части, базы данных, системы контроля версий и другие. Работа в этой области может быть очень увлекательной и высокооплачиваемой, но требует постоянного обучения и совершенствования навыков.

Для разработки на Golang, JavaScript, PostgreSQL, HTML/CSS и микросервисной архитектуре существует множество полезных материалов, включая официальные сайты, курсы на платформах обучения, документации и другие ресурсы. Постоянное обучение и самообразование – ключевой фактор в совершенствовании своих навыков и достижении успеха в разработке программного обеспечения.