ТИТУЛЬНЫЙ ЛИСТ

Разработать приложение для обеспечения пользователя информацией из БД в заданной предметной области.

***Требования:***

пользователь должен получать актуальную информацию по предметной области;

база данных должна содержать не менее 4 таблиц

1. Разработать приложение для работы с БД учета кассовых операций коммерческого банка.

ОГЛАВЛЕНИЕ

[ВЕДЕНИЕ 3](#_Toc201516573)

[РАЗДЕЛ 1. ОБЩИЕ ТЕОРИТЕЧСКИЕ СВЕДЕНЬЯ 4](#_Toc201516574)

[1.1 Информация о предприятии – базе учебной практики. 4](#_Toc201516575)

[1.2 Теоретические сведения 4](#_Toc201516576)

[1.2.1 REST API 4](#_Toc201516577)

[1.2.2 MongoDB 6](#_Toc201516578)

[1.2.3 Swagger 7](#_Toc201516579)

[1.2.4 NodeJS 8](#_Toc201516580)

[РАЗДЕЛ 2. ПРАКТИЧЕСКАЯ ЧАСТЬ 9](#_Toc201516581)

[2.1 Создание Базы Данных 9](#_Toc201516582)

[2.2. Создание таблиц и заполнение таблиц начальными полями. 10](#_Toc201516583)

[2.3 Проверка созданных таблиц и корректности полей в них 12](#_Toc201516584)

[2.4 Написание программы для работы с БД 16](#_Toc201516585)

[2.5 CRUD операции к таблицам БД и проверка работоспособности решения (Postman, MongoDB Compass) 22](#_Toc201516586)

[2.6 Создание собственных запросов к таблицам БД через MQL (MongoDB Query Language) 27](#_Toc201516587)

[ЗАКЛЮЧЕНИЕ 30](#_Toc201516588)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 31](#_Toc201516589)

# ВЕДЕНИЕ

Учебная практика по специальности технического профиля предусматривает закрепление и углубление знаний, полученных обучающимися в процессе теоретического обучения, приобретение ими необходимых умений практической работы по избранной специальности, овладение навыками профессиональной деятельности. Данная работа предполагает целенаправленное освоение всех категорий отрасли программирования и создания структурированных баз данных, редактирования и хранения любой необходимой информации, с незначительным физическим вмешательством, посредством создания программных модулей и дополнения.

Учебная практика по программному модулю ПМ.01 "Ознакомление с технологией разработки программного обеспечения. Создание пользовательского интерфейса" проходила в период с 16 марта 2020 г по 29 марта 2020 г. на базе Таврического колледжа (структурное подразделение) ФГАОУ ВО «Крымский федеральный университет им. В. И. Вернадского».

Заданием от базы практики стало создание сайта с информационным наполнением.

Целью данной работы является «Разработка пользовательского интерфейса на программный продукт в визуальной среде разработки программных продуктов».

Исходя из цели были поставленные следующие задачи:

* изучение языков и методов Web-разработки;
* создание макета сайта;
* реализация формы авторизации и регистрации пользователя.

# РАЗДЕЛ 1. ОБЩИЕ ТЕОРИТЕЧСКИЕ СВЕДЕНЬЯ

## Информация о предприятии – базе учебной практики.

## Теоретические сведения

### REST API

REST (Representational State Transfer) – архитектурный стиль для создания веб-сервисов. API (Application Programming Interface) – способ взаимодействия между клиентом и сервером.

Основные принципы:

* Stateless – сервер не хранит состояние клиента (каждый запрос независим);
* Ресурсы (URI) – данные доступны по уникальным URL (/users, /posts);
* Идемпотентность - свойство, которое говорит, что повторный идентичный запрос, сделанный один или несколько раз подряд, имеет один и тот же эффект не меняя состояние сервера. Корректно реализованные методы PUT, GET, DELETE являются идемпотентыми, но не метод POST;
* Версионирование API – посредством использования v1/v2/v3 stages;
* Документирование API – посредством использования инструментов OpenAPI,Swagger;
* Кеширование даных -–может быть реализовано на стороне клиента(localstorage,sessionstorage), так и на стороне сервера(redis,memcache).

HTTP-методы – определяют действие:

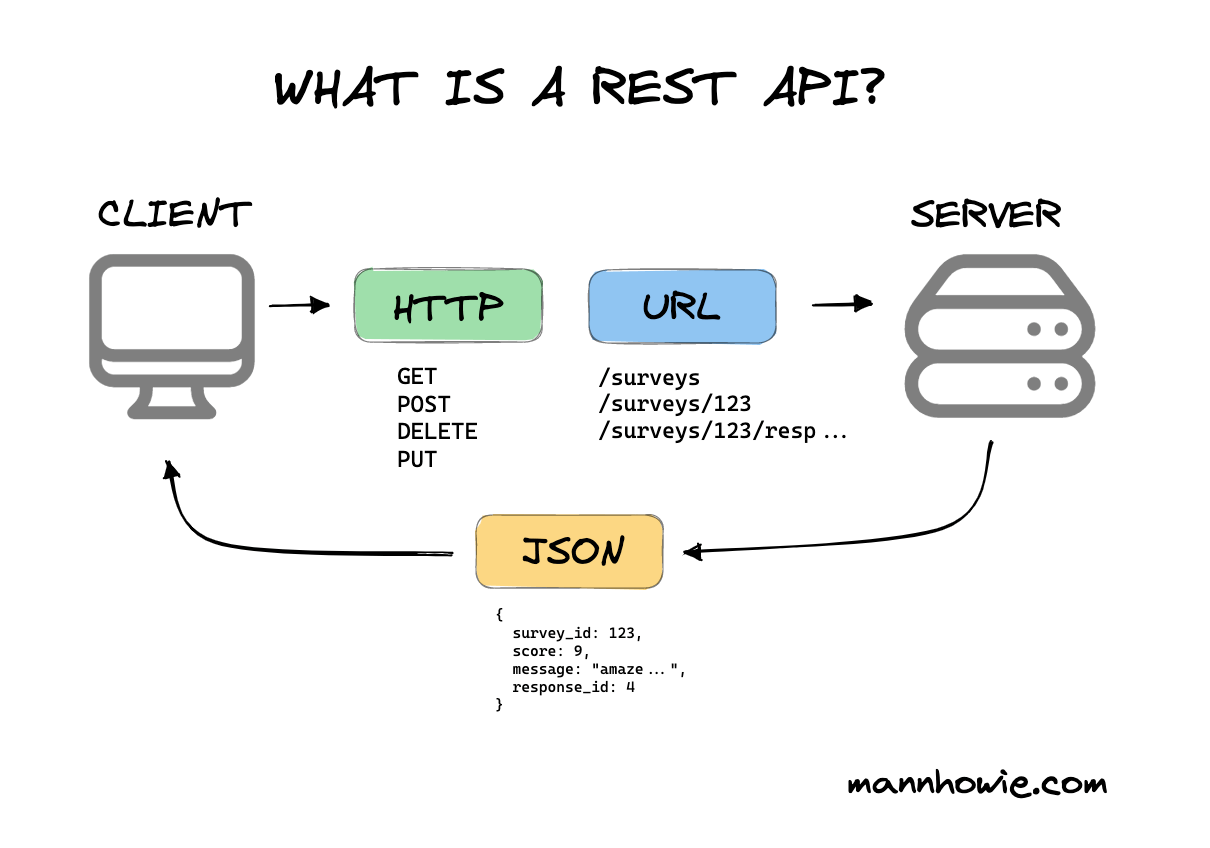
* GET – получить данные (например, список пользователей).
* POST – создать новый ресурс (регистрация пользователя).
* PUT / PATCH – обновить данные (изменить имя пользователя).
* DELETE – удалить ресурс (удалить пользователя).
* Форматы данных – обычно JSON (реже XML).

Рисунок 1 – базовая схема реализации клиент-серверного взаимодействия посредством REST

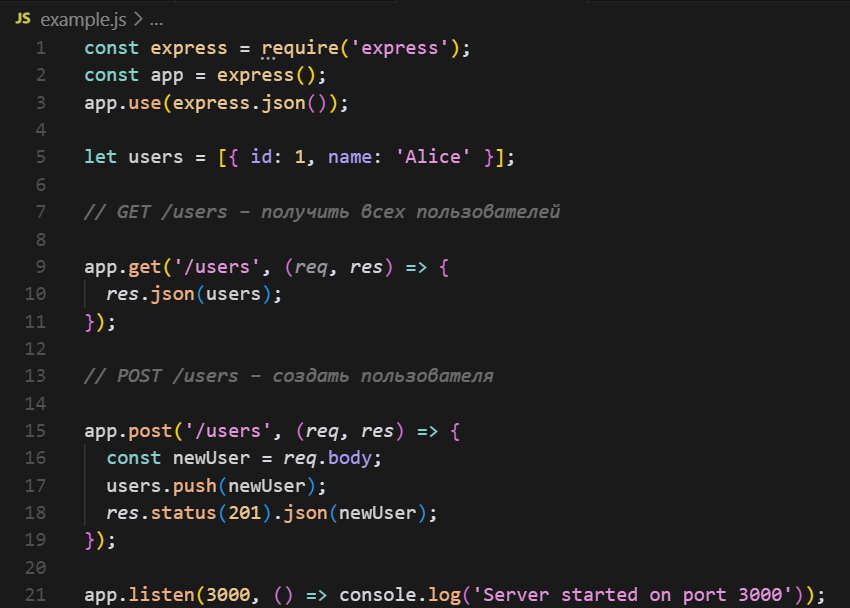
****

Рисунок 2 – пример минимального REST API

### MongoDB

MongoDB – представляет собой ведущую NoSQL-базу данных, которая кардинально отличается от традиционных реляционных систем. Её архитектура основана на документной модели, где данные хранятся в виде BSON-документов (бинарный JSON), что обеспечивает естественное отображение структур данных в современном программировании.

Ключевые особенности:

* Гибкая схема: В отличие от SQL-баз, MongoDB не требует строгого определения структуры таблиц заранее. Документы в одной коллекции могут иметь разную структуру, что особенно полезно при итеративной разработке;
* Масштабируемость: Поддержка горизонтального масштабирования через шардинг позволяет распределять данные по нескольким серверам. Репликация обеспечивает отказоустойчивость и высокую доступность;
* Богатый язык запросов: Поддерживает сложные запросы, включая поиск по полям, диапазонам, регулярным выражениям. Имеет мощные операторы агрегации для аналитики;
* Индексация: Поддерживает различные типы индексов (включая геопространственные и текстовые), что значительно ускоряет поиск.

В качестве СУБД рекомендуется использовать – веб-версию в браузере(Atlas), или MongoDB Compass, однако также исправно работает с многофункциональными СУБД по типу DBeaver.

### Swagger

OpenAPI (ранее Swagger) - это не просто инструмент документации, а целая экосистема для проектирования, разработки и сопровождения API. Спецификация OpenAPI стала отраслевым стандартом для REST API.

Направлен на описание и документирование моделей API сервисов в формате YAML/JSON, упрощения взаимодействия клиентской и серверной части ИС.

Возможности:

* Жизненный цикл API: Поддержка всего цикла от проектирования (Swagger Editor) до тестирования (Swagger UI) и мониторинга;
* Генерация кода: Возможность автоматической генерации клиентских и серверных заглушек на различных языках;
* Валидация: Встроенные механизмы проверки соответствия API его спецификации;
* Безопасность: Поддержка описания схем аутентификации (OAuth2, API Keys и др.);

### 1.2.4 NodeJS

Node.js – это серверная платформа для выполнения JavaScript вне браузера.

Основные моменты:

* Работает на движке V8 (Chrome).
* Асинхронная, событийно-ориентированная архитектура (non-blocking I/O).
* Используется для бэкенд-разработки, API, микросервисов, чат-ботов.
* Имеет огромную экосистему (npm – крупнейший пакетный менеджер).
* В качестве языка для интерпретации платформой использует JS/TS

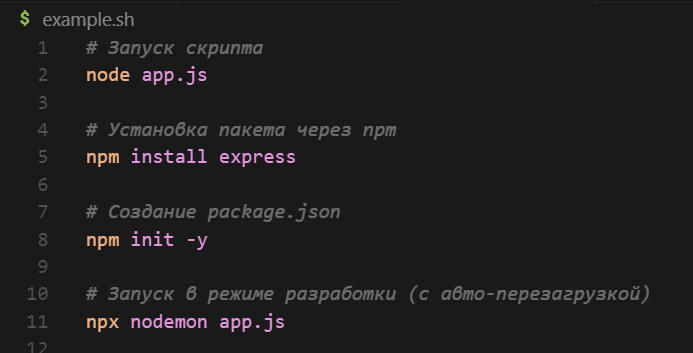


Рисунок 3 – основные команды для работы с Node JS



Рисунок 4 – пример простейшего сервера на Node JS

# РАЗДЕЛ 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2.1 Создание Базы Данных

Для реализации технического задания по практике, первым делом была создана База Данных, для удобства я использовал образ mongo:latest и запускал базу в контейнере, благодаря использования volume – таблицы и записи в базе будут сохранятся при остановке/перезапуске контейнера.

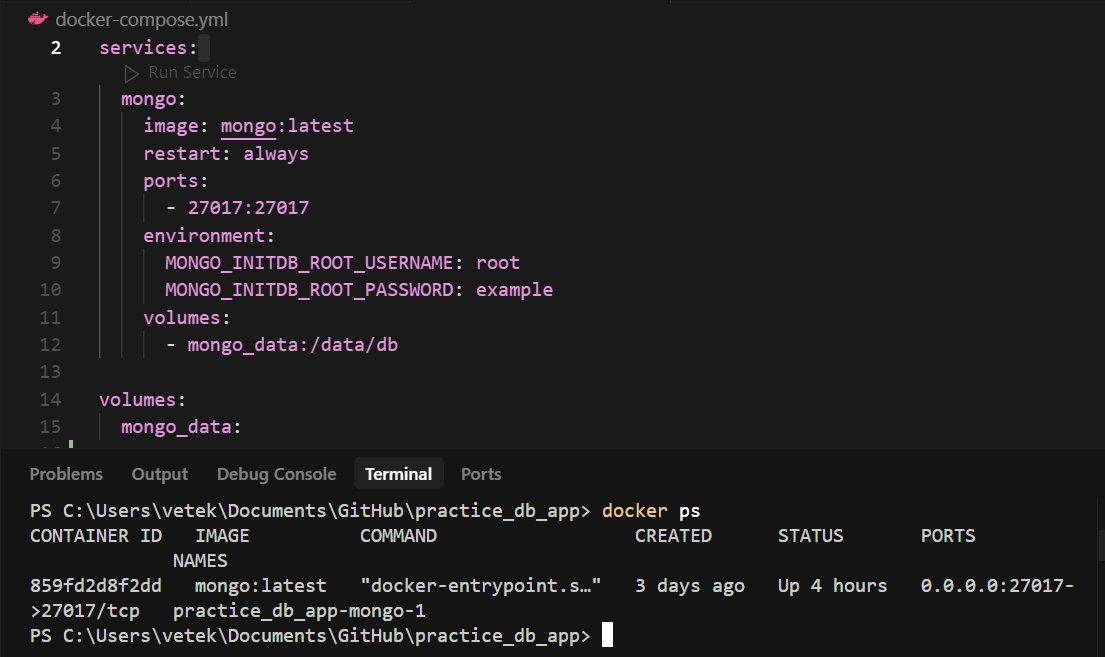
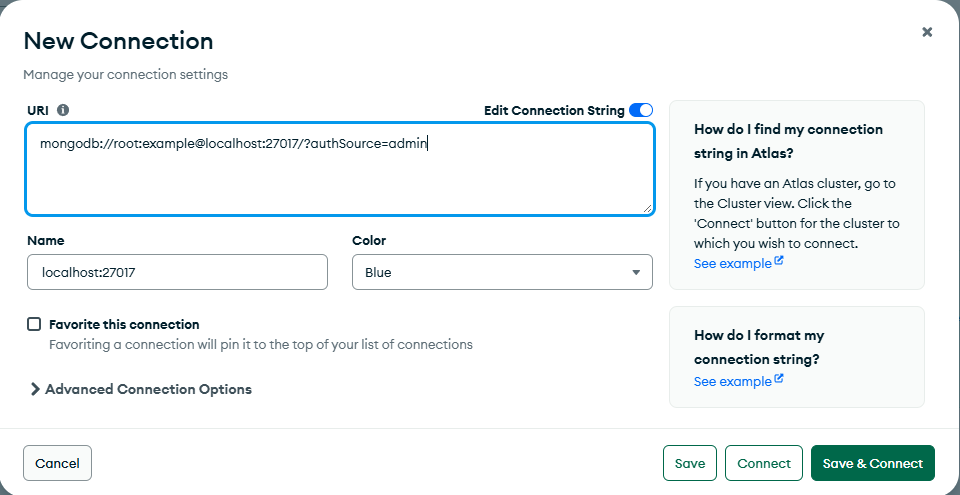


Рисунок 5 – docker-compose файл с конфигурацией БД mongoDB

После поднятия контейнера БД будет доступна на порту 21017 на всех сетевых интерфейсах системы.

Рисунок 6 – проверка подключения к БД через СУБД MongoDB Compass

### 2.2. Создание таблиц и заполнение таблиц начальными полями.

Для наполнения таблиц тестовыми значениями был написан и запущен скрипт – seed.js[код в приложении]

Основные этапы:

1. Подключение к MongoDB

* Использует строку подключения из .env (переменная MONGO\_URI).

1. Очистка старых данных

* Удаляет все записи из коллекций: клиентов, счетов, транзакций и кассиров.

1. Генерация кассиров

* Создаёт 5 случайных кассиров с помощью библиотеки faker.

1. Генерация клиентов и счетов

* Создаёт 20 клиентов с фейковыми паспортными данными.
* Для каждого клиента создаёт банковский счёт с валютой, балансом и статусом.

1. Генерация транзакций

* Создаёт 50 транзакций разных типов: пополнение (deposit), снятие (withdrawal), перевод (transfer).
* Для переводов выбирает разные счета отправителя и получателя.
* Каждая транзакция привязана к случайному кассиру.

1. Завершение работы

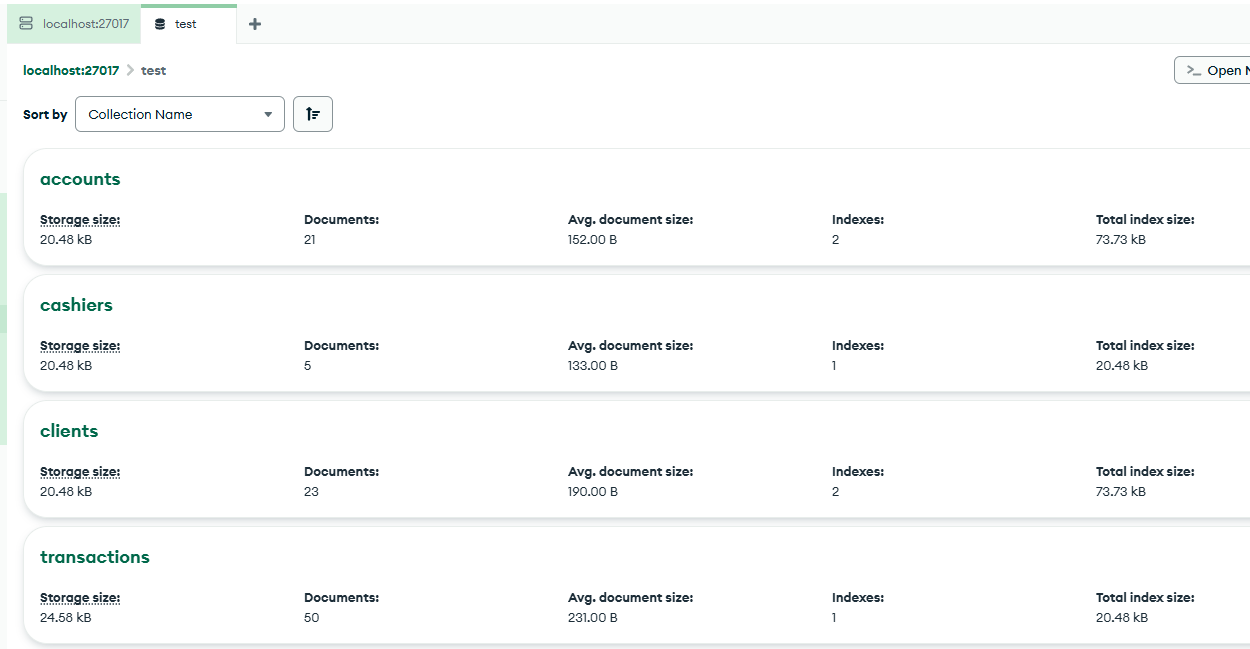
* После генерации данных скрипт отключается от базы.

Рисунок 7 – проверка что БД test и 4 таблицы в ней успешно создались через MongoDB Compass

### 2.3 Проверка созданных таблиц и корректности полей в них

Было выделено 4 основные таблицы в качестве сущностей БД:

* accounts – сущность банковского счета пользователя в ИС банка;
* clients – сущность пользователя(клиента) в ИС банка;
* transactions – сущность каждой операций по аккаунту пользователя;
* cashiers – сущность кассира(оператора), обрабатывающего транзакцию в ИС банка.

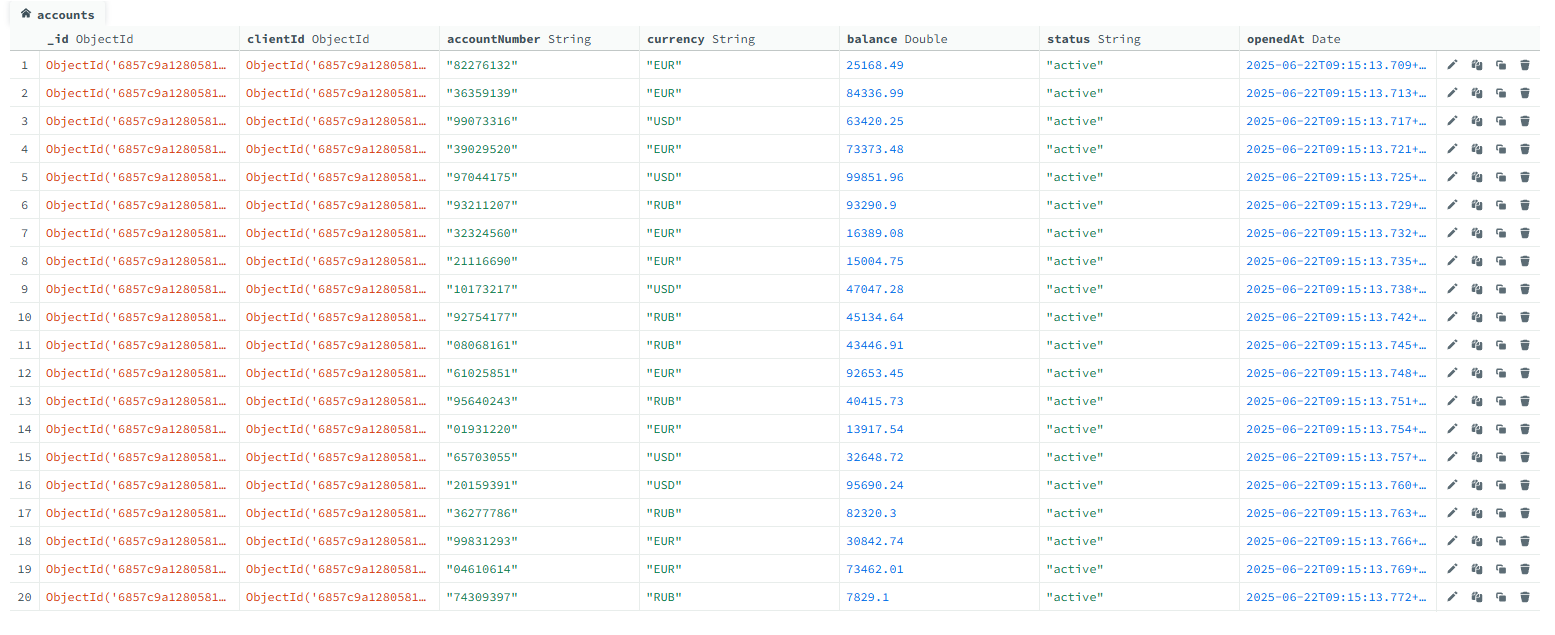
Содержит 20 начальных записей, таблица выглядит следующим образом:

Рисунок 8 – таблица accounts

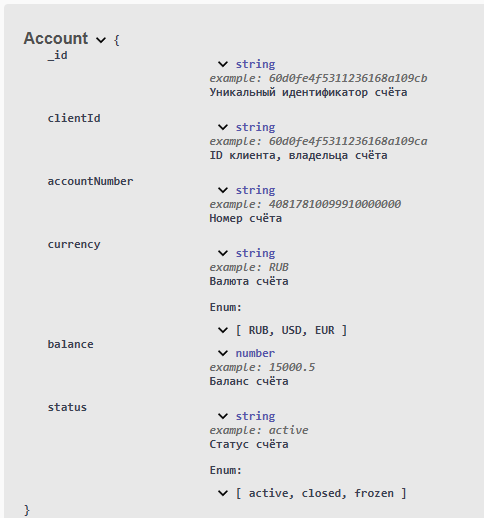


Рисунок 9 – схема данных таблицы accounts

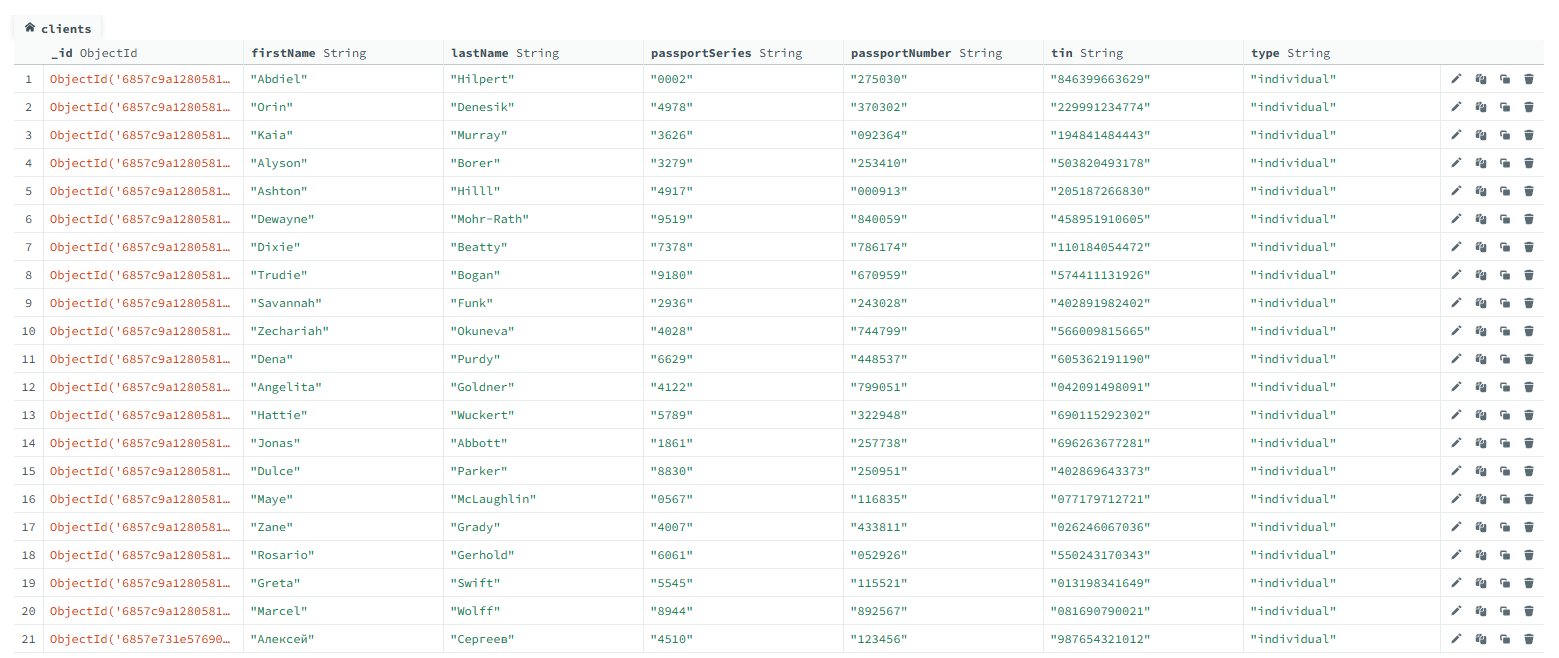


Рисунок 10 – таблица clients



Рисунок 11 – схема данных таблицы clients

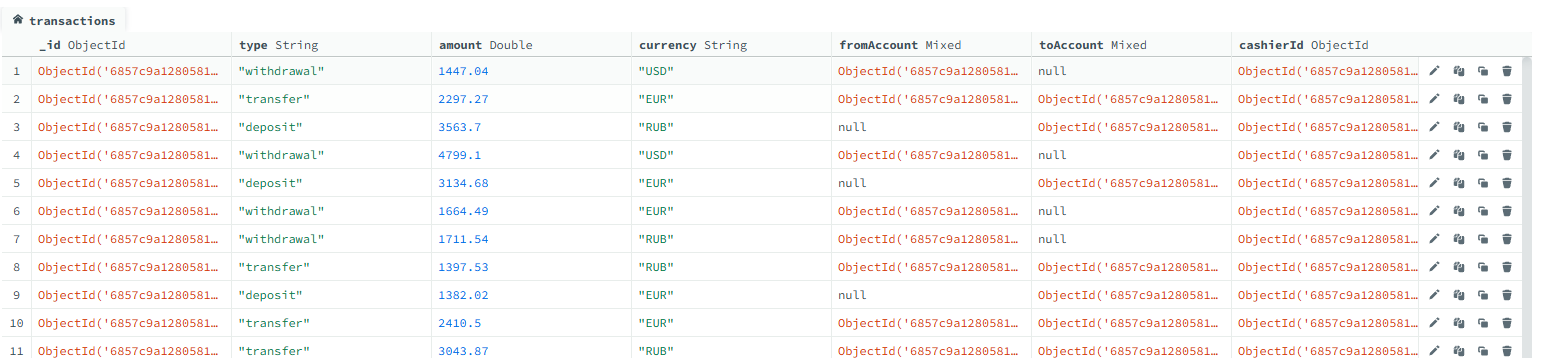


Рисунок 12 – таблица transactions

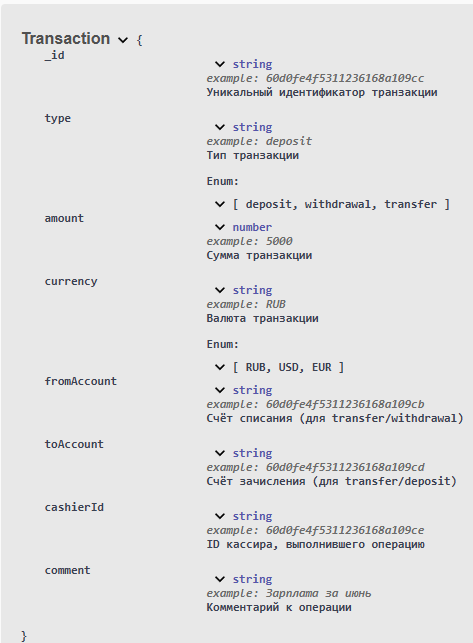


Рисунок 13 – схема данных таблицы transactions

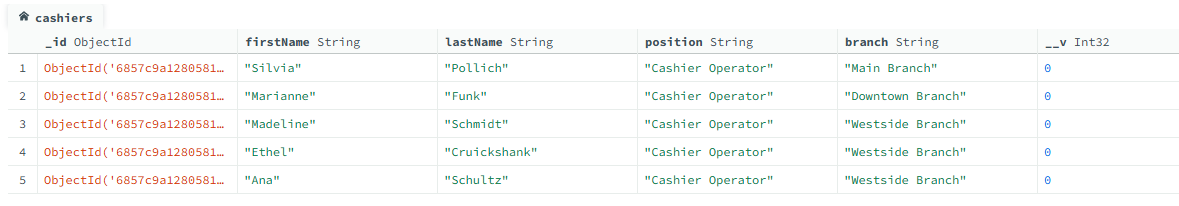


Рисунок 14 – таблица cashiers



Рисунок 15 – схема данных таблицы cashiers

### 2.4 Написание программы для работы с БД

Программная часть серверного приложения была написана на ExpressJS и Mongoose (ODM) для удобной работы в стиле ООП с mongoDB.

С общей структурой проекта вы можете ознакомиться на рисунке 13.

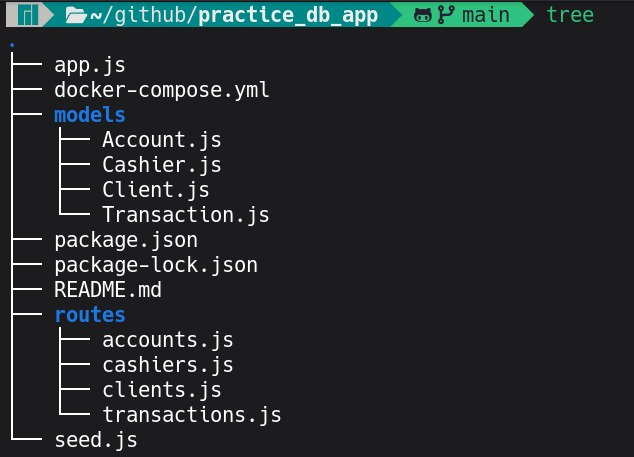


Рисунок 16 – общая структура проекта

- app.js – основной файл приложения, в нем происходит инициализация Express-сервера, подключение к базе данных и настройка маршрутов для обработки endpoints;

- docker-compose.yml – конфигурация контейнера для запуска образа с mongoDB;

- package.json – основной файл npm-пакета .Содержит информацию о проекте, его зависимостях, скриптах для запуска и сборки;

- package-lock.json - aвтоматически генерируемый файл, фиксирующий точные версии установленных npm-зависимостей;

- README.md - документация по проекту: как запускать, описание функционала, требования и т.д;

- seed.js - скрипт для заполнения базы данных начальными (тестовыми) данными;

-swagger.yaml - описание API в формате Swagger/OpenAPI. Используется для автогенерации документации и тестирования API;

- models/ - Содержит модели данных для работы с базой данных (обычно через ORM, например, Sequelize или Mongoose):

* account.js - модель для работы с аккаунтами (счетами) пользователей;
* cashier.js - модель для кассиров (например, сотрудники банка или магазина);
* client.js - модель для клиентов (пользователей, владельцев счетов);
* transaction.js - модель для транзакций (операций между счетами, пополнения, снятия и т.д.).

- routes/ - содержит файлы с маршрутами (endpoint'ами) для Express-приложения:

* accounts.js - маршруты для работы с аккаунтами (создание, просмотр, изменение, удаление счетов);
* cashiers.js - маршруты для работы с кассирами;
* clients.js - маршруты для работы с клиентами;
* transactions.js - маршруты для работы с транзакциями.

Как это работает вместе:

1. app.js запускает сервер, подключает модели и маршруты.
2. models/ определяет структуру данных и взаимодействие с базой.
3. routes/ реализует API для работы с этими данными.

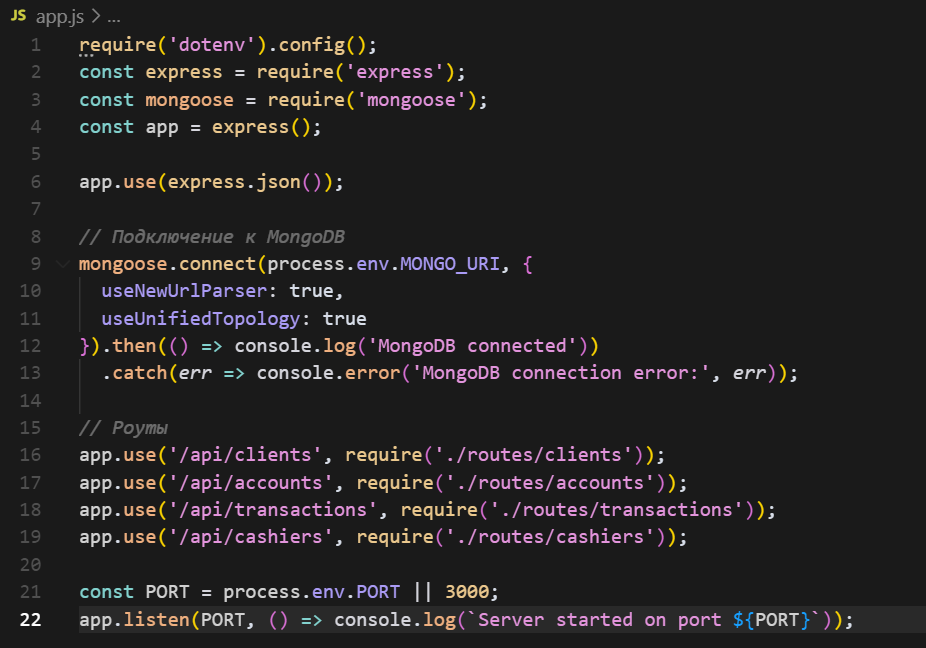
Рассмотрим реализацию app.js, а также для примера схему Транзакций и реализацию endpoint для нее.

Рисунок 17 – листинг основного скрипта приложения

В файле:

1. загрузка переменных окружения из файла для подключения к БД;
2. импорт библиотек mongoose и express;
3. создается приложение(app), на базе фреймворка express;
4. посредством модуля mongoose (ODM для NodeJS) идет подключение приложением к БД;
5. к приложению подключаются «ручки» (endpoind) и указывается путь до файлов с их реализацией в проекте;
6. приложение запускается на локальном сокете.

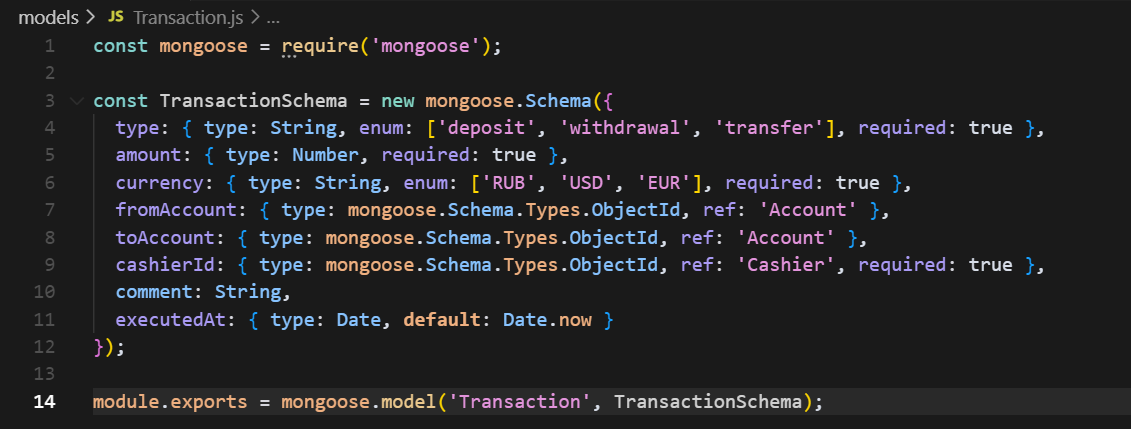
****

Рисунок 18 – листинг модели данных для таблицы Transactions (файл models/Transaction.js)

В файле:

1. импортируется библиотека mongoose для работы с MongoDB;
2. описывается схема (структура) данных для транзакции: тип, сумма, валюта, счета отправителя и получателя, кассир, комментарий, дата выполнения;
3. экспортируется модель Transaction для использования в других частях приложения.

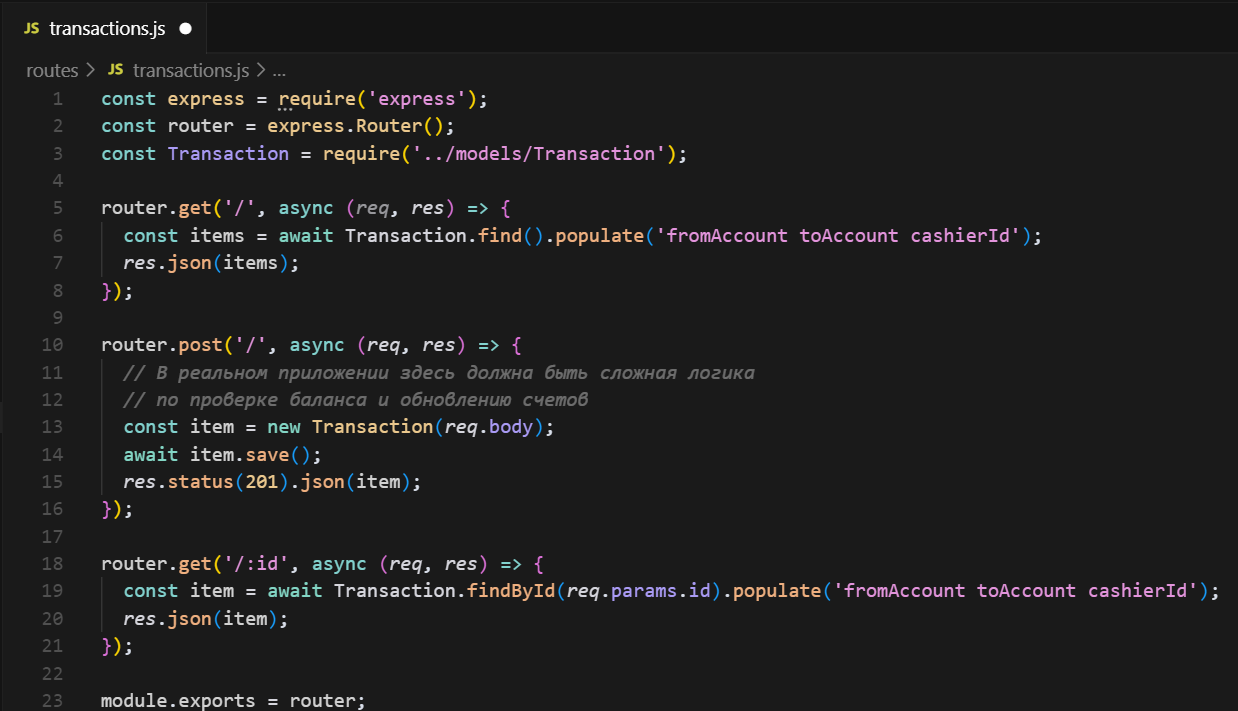
****

Рисунок 19 – листинг endpoints для схемы данных Transactions

(файл routes/transactions.js)

В файле:

1. импортируется express и создаётся router для маршрутов;
2. импортируется модель Transaction;
3. описывается GET-роут по адресу / — возвращает список всех транзакций с деталями счетов и кассира;
4. описывается POST-роут по адресу / — создаёт новую транзакцию на основе данных из запроса;
5. описывается GET-роут по адресу /:id — возвращает одну транзакцию по её id с деталями счетов и кассира;
6. экспортируется router для подключения в основное приложение.

C аналогичным листингом остальных файлов, а именно схемы коллекций(таблиц) для БД и реализация endpoint для них, вы можете ознакомиться подробнее в Приложении к работе.

### 2.5 CRUD операции к таблицам БД и проверка работоспособности решения (Postman, MongoDB Compass)

Для проверки созданных endpoint к таблицам БД, я использовал утилиту Postman. Рассмотрим основные запросы и результаты их выполнения.

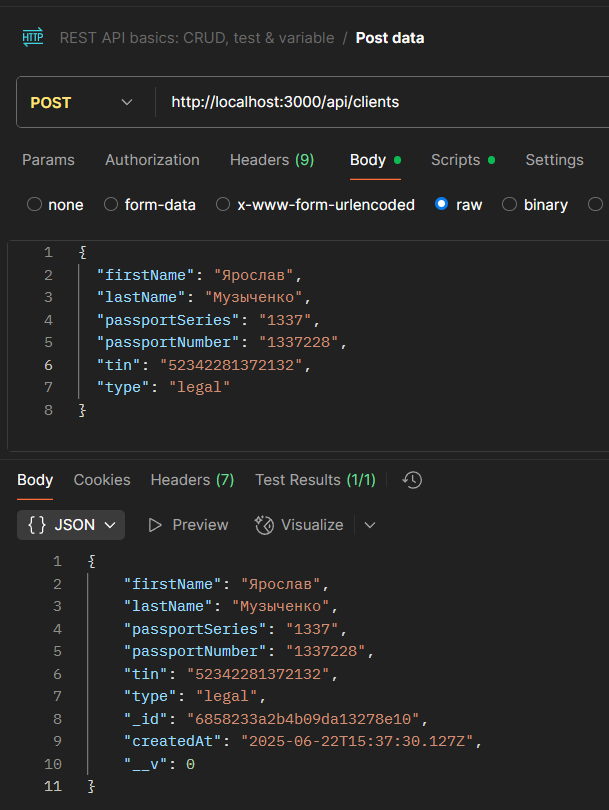


Рисунок 20 – POST запрос на endpoint /clients – создание нового пользователя

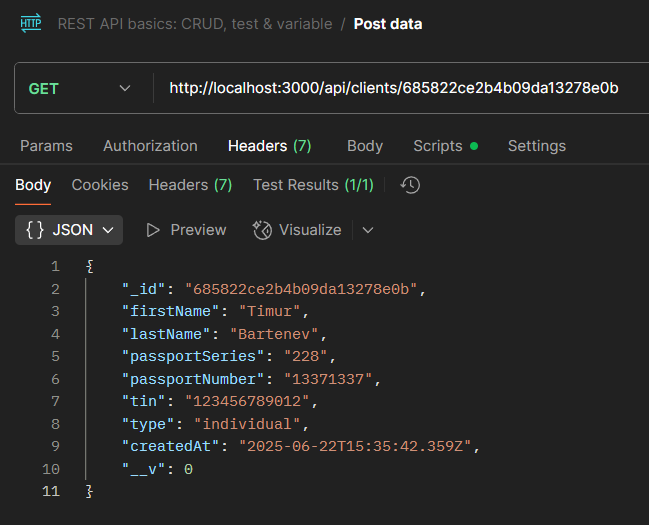


Рисунок 21 – GET запрос на endpoint /clients/{id} – получение конкретного пользователя по его ID

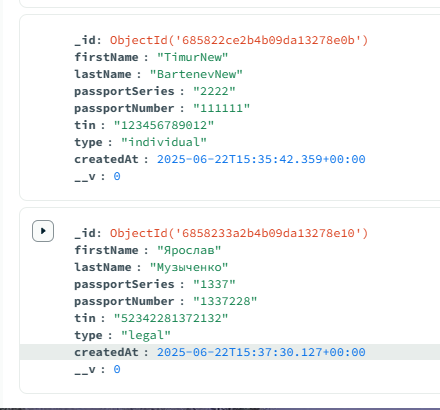


Рисунок 22 – проверка что действия над записями и информацией в записях таблиц действительно корректно применяются в БД

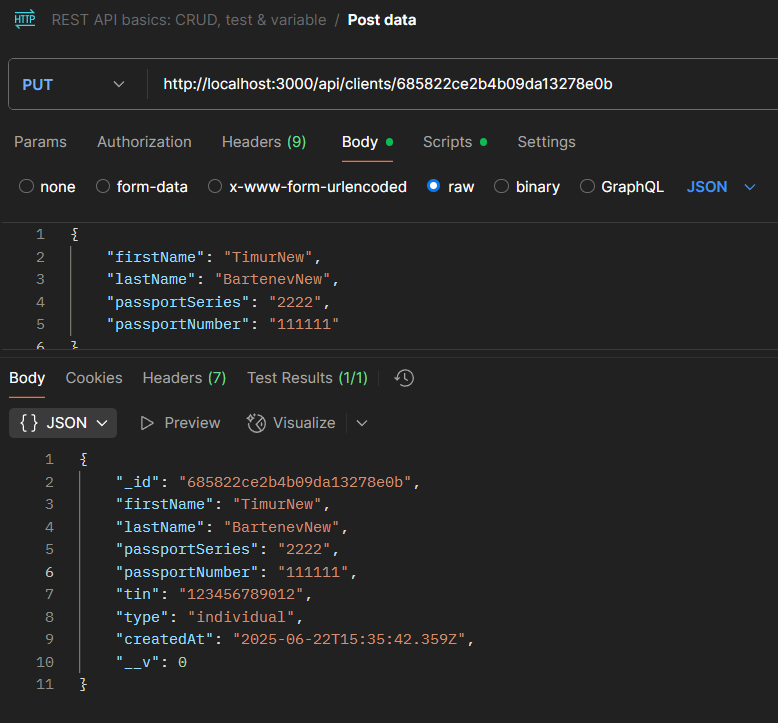


Рисунок 23 – PUT запрос на endpoint /clients/{id} – обновление информации о пользователе банка по его ID

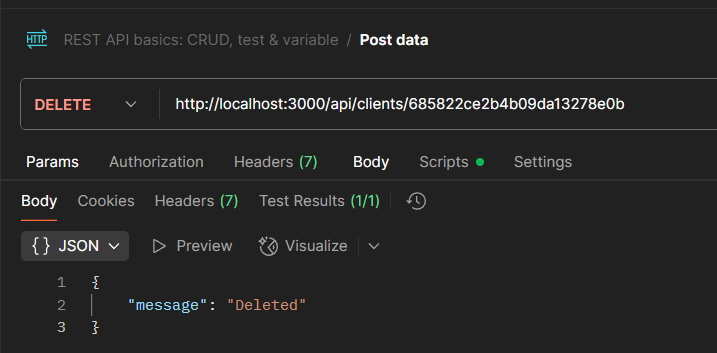


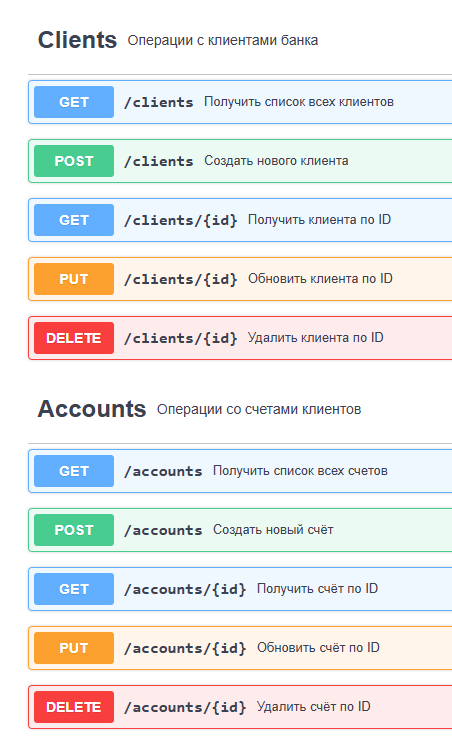
Рисунок 24 – DELETE запрос на endpoint /clients/{id} – удаление записи о пользователе из таблицы clients по его ID

Рисунок 25 – swagger спецификация endpoint к сущностям БД

В целом для остальных endpoints реализованы похожие базовые CRUD операции, там, где это возможно, вот как выглядит список возможных операций в swagger-документации, вы можете ознакомиться на рисунках 23 и 24.

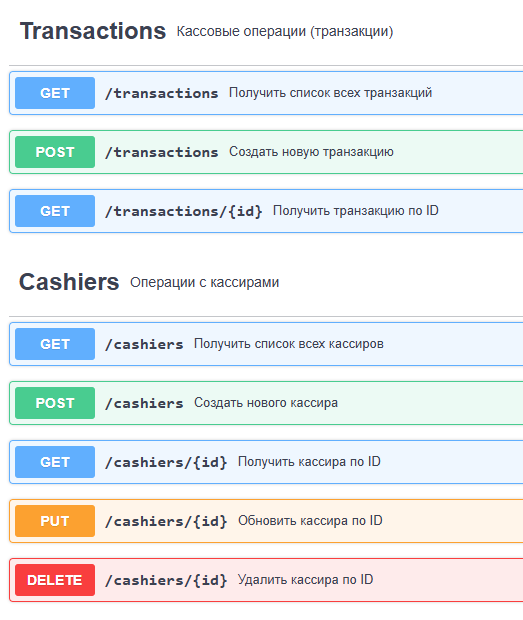


Рисунок 26 – swagger документация endpoint к сущностям БД

### 2.6 Создание собственных запросов к таблицам БД через MQL (MongoDB Query Language)

Для каждой коллекции(таблицы) реализовал несколько проверочных и полезных запросов для выборки и последующего анализа.

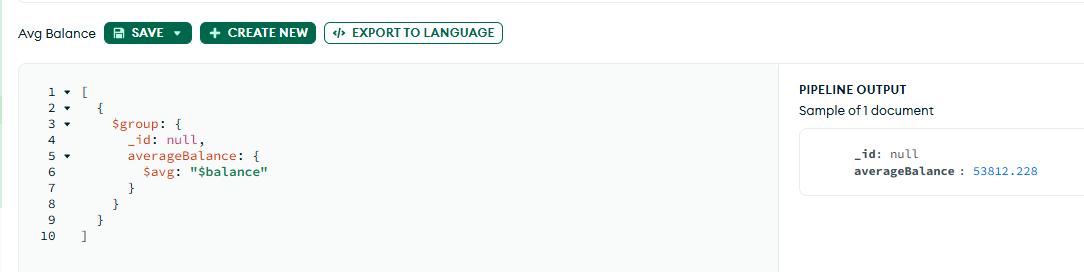


Рисунок 27 – вывод среднего кол-ва средств на балансе аккаунтов (во всех валютах)

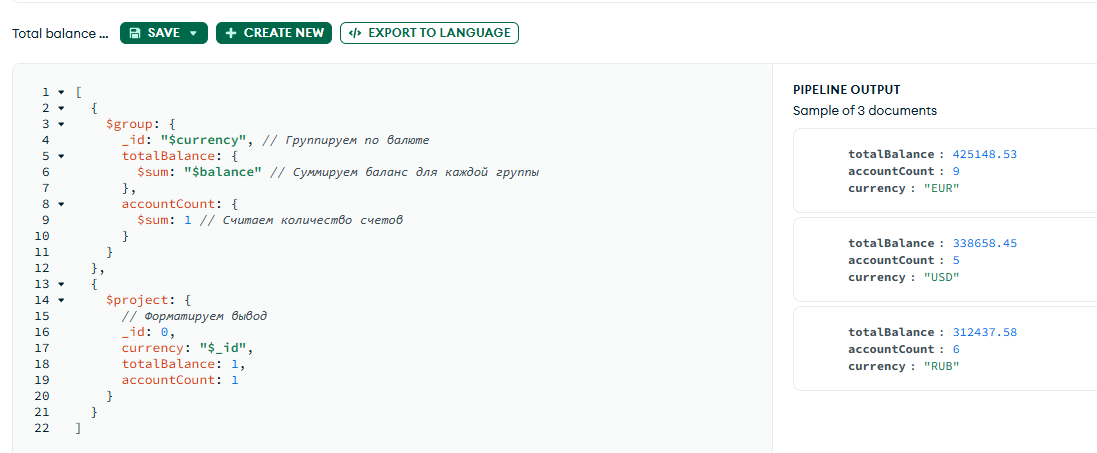


Рисунок 28 – вывод общего кол-ва средств на счетах пользователей с разделением по валютным единицам

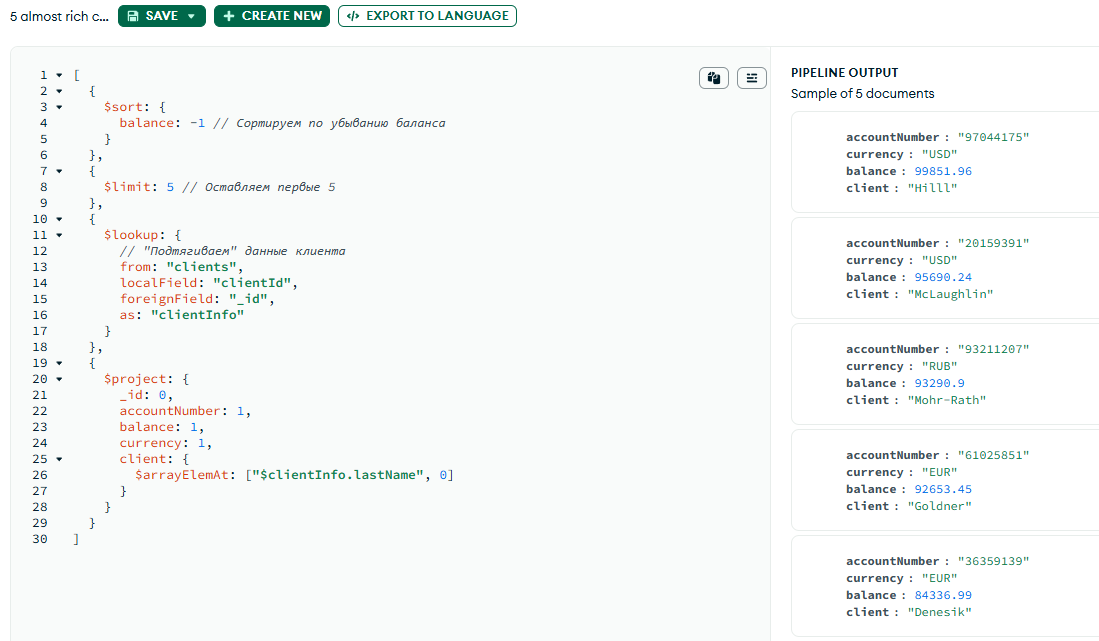


Рисунок 29 – вывод пяти самых богатых клиентов по всем валютам

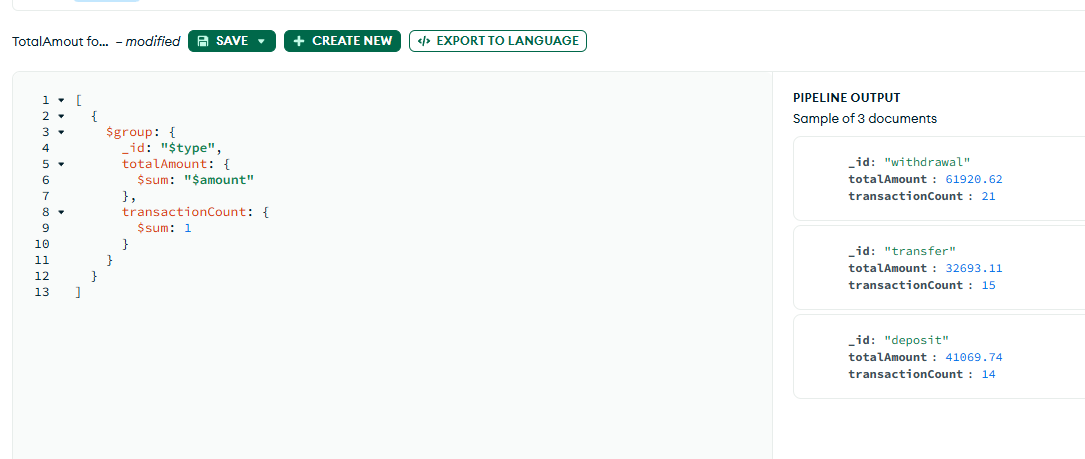


Рисунок 30 – вывод средней суммы для каждого типа перевода

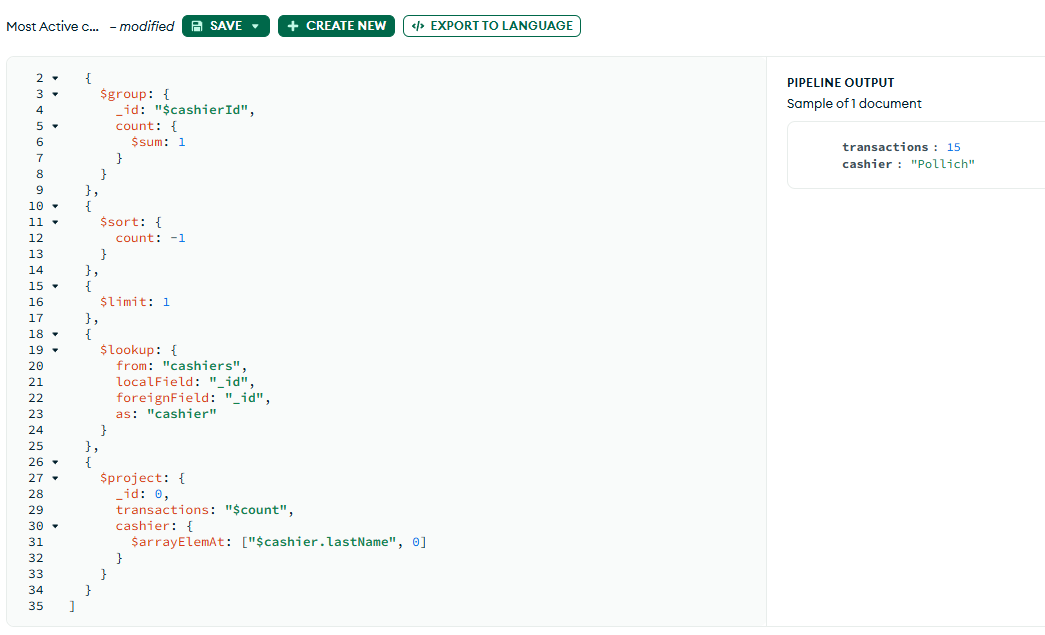


Рисунок 31 – вывод информации о самом эффективном кассире (провел больше всего транзакций)

# ЗАКЛЮЧЕНИЕ

Учебная практика по программному модулю ПМ.01"Ознакомление с технологией разработки программного обеспечения. Создание пользовательского интерфейса", проводилась согласно, положения о практике обучающихся, осваивающих основные профессиональные образовательные программы среднего профессионального образования.

В ходе прохождения учебной практики поставленная цель была достигнута, а также были выполнены задачи:

* изучение языков и методов Web-разработки;
* создание макета сайта;
* реализация формы авторизации и регистрации пользователя.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

ПРИЛОЖЕНИЕ

const mongoose = require('mongoose');

const { faker } = require('@faker-js/faker');

require('dotenv').config();

const Client = require('./models/Client');

const Account = require('./models/Account');

const Transaction = require('./models/Transaction');

const Cashier = require('./models/Cashier');

async function **seed**() {

  await mongoose.connect(process.env.MONGO\_URI, {

    useNewUrlParser: true,

    useUnifiedTopology: true

  });

  console.log('Clearing old data...');

  await Client.deleteMany();

  await Account.deleteMany();

  await Transaction.deleteMany();

  await Cashier.deleteMany();

  console.log('Seeding new data...');

*// Кассиры*

  const cashiers = [];

  for (let i = 0; i < 5; i++) {

    cashiers.push(await Cashier.create({

      firstName: faker.person.firstName('female'),

      lastName: faker.person.lastName('female'),

      position: 'Cashier Operator',

      branch: faker.helpers.arrayElement(['Main Branch', 'Westside Branch', 'Downtown Branch'])

    }));

  }

*// Клиенты и их счета*

  const accounts = [];

  for (let i = 0; i < 20; i++) {

    const client = await Client.create({

      firstName: faker.person.firstName(),

      lastName: faker.person.lastName(),

      passportSeries: faker.string.numeric(4),

      passportNumber: faker.string.numeric(6),

      tin: faker.string.numeric(12),

      type: 'individual'

    });

    accounts.push(await Account.create({

      clientId: client.\_id,

      accountNumber: faker.finance.accountNumber(),

      currency: faker.helpers.arrayElement(['RUB', 'USD', 'EUR']),

      balance: faker.finance.amount({ min: 1000, max: 100000, dec: 2 }),

      status: 'active'

    }));

  }

*// Транзакции*

  for (let i = 0; i < 50; i++) {

    const type = faker.helpers.arrayElement(['deposit', 'withdrawal', 'transfer']);

    const fromAccount = faker.helpers.arrayElement(accounts);

    let toAccount = null;

    if (type === 'transfer') {

        do {

            toAccount = faker.helpers.arrayElement(accounts);

        } while (toAccount.\_id.equals(fromAccount.\_id));

    }

    await Transaction.create({

      type,

      amount: faker.finance.amount({ min: 100, max: 5000, dec: 2 }),

      currency: fromAccount.currency,

      fromAccount: (type === 'withdrawal' || type === 'transfer') ? fromAccount.\_id : null,

      toAccount: (type === 'deposit' || type === 'transfer') ? (type === 'deposit' ? fromAccount.\_id : toAccount.\_id) : null,

      cashierId: faker.helpers.arrayElement(cashiers).\_id,

      comment: faker.lorem.sentence()

    });

  }

  console.log('Test data seeded successfully!');

  await mongoose.disconnect();

}

seed().catch(*err* => {

  console.error(err);

  mongoose.disconnect();

});

Приложениe 1 – листинг программного кода файла seed.js ( скрипт генерирует начальные и тестовые данные в таблицы БД)

const mongoose = require('mongoose');

const CashierSchema = new mongoose.Schema({

  firstName: { type: String, required: true },

  lastName: { type: String, required: true },

  position: String,

  branch: String *// Отделение банка*

});

module.exports = mongoose.model('Cashier', CashierSchema);

Приложенииe 2 – листинг программного кода файла models/Сashier.js ( скрипт создает схему(модель) данных для документов(записей) в коллекции(таблице) Cashier

const mongoose = require('mongoose');

const AccountSchema = new mongoose.Schema({

  clientId: { type: mongoose.Schema.Types.ObjectId, ref: 'Client', required: true },

  accountNumber: { type: String, unique: true, required: true },

  currency: { type: String, enum: ['RUB', 'USD', 'EUR'], required: true },

  balance: { type: Number, default: 0 },

  openedAt: { type: Date, **default**: Date.now },

  status: { type: String, enum: ['active', 'closed', 'frozen'], default: 'active' }

});

module.exports = mongoose.model('Account', AccountSchema);

Приложенииe 3 – листинг программного кода файла models/Account.js ( скрипт создает схему(модель) данных для документов(записей) в коллекции(таблице) Cashier

const mongoose = require('mongoose');

const ClientSchema = new mongoose.Schema({

  firstName: { type: String, required: true },

  lastName: { type: String, required: true },

  middleName: String,

  passportSeries: { type: String, required: true },

  passportNumber: { type: String, required: true },

  tin: { type: String, unique: true }, *// ИНН*

  type: { type: String, enum: ['individual', 'legal'], required: true },

  createdAt: { type: Date, **default**: Date.now }

});

module.exports = mongoose.model('Client', ClientSchema);

Приложенииe 4 – листинг программного кода файла models/Client.js ( скрипт создает схему(модель) данных для документов(записей) в коллекции(таблице) Cashier