Министерство образования Республики Беларусь

Учреждение образования БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОННИКИ

Факультет компьютерных систем и сетей Кафедра электронных вычислительных машин

Лабораторная работа №3

Разработка NoSQL базы данных и спецификаций прикладной программы.

Разработка серверной части прикладной программы

Студент: Р.Е. Власов

Преподаватель: А.И. Крюков

МИНСК 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ 3

2 [СОЗДАНИЕ NOSQL БД](#_TOC_250005) 4

2.1 [Миграция с Postgress в MongoDB](#_TOC_250004) 5

3 [ТЕХНИЧЕСКИЕ ТРЕБОВАНИЯ](#_TOC_250003) 7

3.1 [Серверное приложение](#_TOC_250002) 7

3.2 [Клиентское приложение. Интерфейс](#_TOC_250001) 8

4 [ВЫПОЛНЕНИЕ РАБОТЫ](#_TOC_250009) 10

4.1 [Выбор языка программирования и дополнительных компонентов](#_TOC_250008) 11  
4.2 [Взаимодействие с базой данных 12](#_TOC_250007)

4.3 [Основные части пользовательского интерфейса 13](#_TOC_250006)

4.4 [Листинг кода 14](#_TOC_250005)  
 5 [РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 1](#_TOC_250004)5

5.1 [Развертывание приложения](#_TOC_250003) 16

5.2 [Работа с приложением](#_TOC_250002) 17

6 [ВЫВОД](#_TOC_250001) 18

[ПРИЛОЖЕНИЕ А](#_TOC_250000) 19

# ЦЕЛЬ РАБОТЫ

В лабораторной работе выполняется концептуальное проектирование NoSQL БД и процесс миграции SQL базы данных на созданную NoSQL.

Темой данной лабораторной работы является переход с SQL базы данных на NoSQL в организации «Кинотеатр», разработка спецификаций серверной части (backend) программы, программирование серверной части с использованием прикладного интерфейса СУБД MongoDB;

«Кинотеатр» представляет собой стандартную модель, работающую по принципу «клиент, сеанс, фильм, билет». Взаимодействие происходит между клиентом и кинотеатром, где клиент выбирает сеанс, фильм и приобретает билет. Кинотеатр предоставляет услуги просмотра фильмов, продавая билеты на различные сеансы, организуемые в залах.

# СОЗДАНИЕ NOSQL БД

В данной лабораторной работе в качестве noSQL базы данных будет использоваться MongoDB.

MongoDB — это высокопроизводительная, масштабируемая NoSQL база данных, разработанная для обработки больших объемов данных и обеспечения высокой доступности и гибкости. Вот несколько ключевых аспектов и возможностей MongoDB:

Основные концепции:

* документно-ориентированная модель: вместо таблиц и строк, как в реляционных базах данных, MongoDB использует коллекции и документы. Документы — это записи в формате BSON (бинарный JSON), что позволяет хранить сложные данные и вложенные структуры;
* коллекции: группы документов. В MongoDB нет жесткой схемы, что позволяет гибко менять структуру данных.

Ключевые особенности:

* горизонтальное масштабирование: MongoDB поддерживает шардирование — распределение данных по нескольким серверам для повышения производительности и масштабируемости;
* высокая доступность: использование репликации для обеспечения доступности и отказоустойчивости. Реплицированные наборы (replica sets) включают несколько копий данных на разных серверах;
* гибкость в работе с данными: поддержка вложенных документов и массивов позволяет моделировать сложные структуры данных напрямую в базе;
* мощный язык запросов: MongoDB Query Language (MQL) предлагает широкий набор операций для поиска, фильтрации, и манипуляции данными.

Примеры использования:

* интернет-магазины: поддержка динамических схем позволяет легко обновлять каталоги товаров;
* реалтайм аналитика: высокая производительность MongoDB делает ее отличным выбором для систем с большими объемами данных и необходимостью быстрого ответа;
* мобильные приложения: гибкость и масштабируемость MongoDB помогают эффективно работать с различными типами данных, которые могут изменяться с течением времени.

MongoDB широко используется в различных приложениях и является одним из популярных решений для работы с большими данными и высоконагруженными системами.

## Миграция с Postgress в MongoDB

Первоначально были подключены все необходимые библиотеки и зависимости, включая драйвер MongoDB и инструменты для работы с реляционной базой данных. Это обеспечило основу для взаимодействия между двумя системами хранения данных. Затем был создан класс MigrationService, который отвечал за установление соединений с обоими типами баз данных. В конструкторе этого класса настроено подключение к MongoDB с использованием строки подключения из конфигурационного файла, а также инициализированы соответствующие коллекции для каждой модели данных.

Далее был реализован метод MigrateAllAsync, выполняющий процесс миграции данных. Этот метод последовательно извлекал данные из каждой таблицы реляционной базы данных и вставлял их в соответствующие коллекции MongoDB. Для обеспечения корректной сериализации и десериализации объектов в MongoDB были добавлены необходимые атрибуты BSON к моделям данных. Это позволило правильно отображать идентификаторы и управлять связями между сущностями, исключая избыточные данные и предотвращая возможные циклические ссылки.

После завершения процесса миграции данные были проверены с помощью MongoDB Compass, графического интерфейса для работы с MongoDB. В Compass удалось визуально подтвердить, что все данные успешно перенесены и правильно структурированы в новых коллекциях. Это обеспечило уверенность в корректности выполнения миграции и позволило продолжить управление и анализ данных в MongoDB.

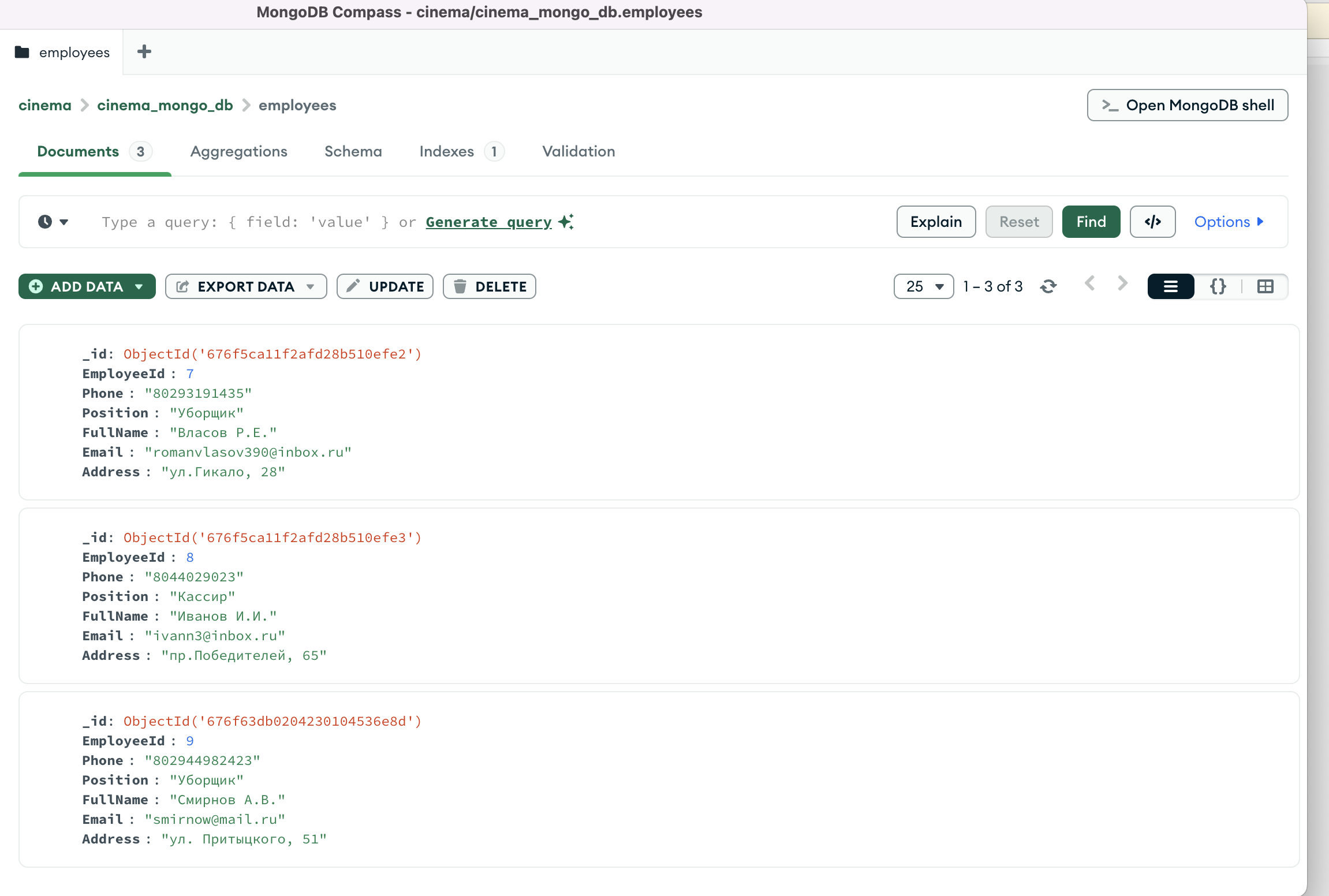


Рисунок 1.1 – Мигрированная таблица “Employees” с Postgres

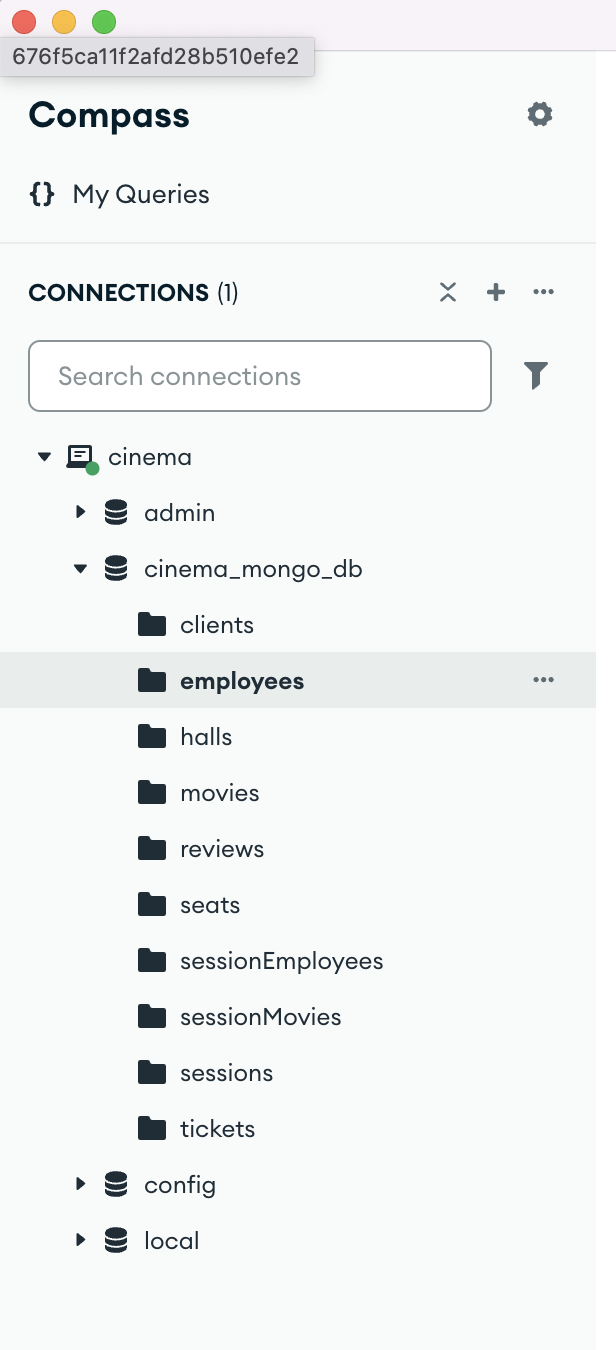


Рисунок 1.2 – Общая структура базы данных в MongoDB

# 3 ТЕХНИЧЕСКИЕ ТРЕБОВАНИЯ

Технические требования содержат принципы построения взаимодействия клиент-серверного приложения в рамках работы с базой данных, но оторвано от конкретной реализации будь то Postgres или MongoDB.

Технические требования подразделяются на требования для серверного приложения и требования для интерфейса клиентского приложения.

## Серверное приложение

1. Серверное приложение для реализации соединения с базой данный Postgres будет написано на языке C#.
2. Должны быть предусмотрены CRUD операции для всех таблиц из ER- диаграммы представленной на рисунке 1.
3. Серверные операции должны быть описаны обще, для дальнейнейшего масштабирования и наследования.
4. В серверном приложении должны быть описаны все используемые сущности базы данных.
5. Приложение должно быть оптимизированным.

## Клиентское приложение. Интерфейс

1. Клиенсткое приложение должно быть написано в C# с ипользование Win Forms, для обеспечения быстродействия и реактивности.
2. Интерфейс приложения должен отвечать принципам UI/UX. Дизайн должен быть удобен, понятен и однозначен.
3. Приложение должно иметь минималистичный дизайн.
4. Приложение должно быть оптимизированным.

# ВЫПОЛНЕНИЕ РАБОТЫ

## Выбор языка программирования и дополнительных компонентов

### В качестве языка программирования для реализации серверной и клиентской частей программы будет использоваться С#, а конкретнее само приложение будет работать на платформе пользовательского интерфейса для создания разнообразных кроссплатформенных клиентских приложений рабочего стола с использованием .Net Framework 9.0 и для взаимодействия с базой данных будет использован пакет MongoDB.Driver версии 9.0.0.

## 4.2 Взаимодействие с базой данных

Для подключения к базе данных будет использован класс MongoClient, при создании которого будет передаваться строка подключения в формате

«mongodb://localhost:27017», где: localhost=(адрес расположения базы данных); 27017(порт);

Для получения нужной базы данных будет использован интерфейс IMongoClient, для получения которого будет использована функция класса MongoClient GetDatabase, в которую передаётся название базы данных в виде строки.

Для получения данных из коллекций будет использованы интерфейс контроллеров сущностей, которые будет заполняться при помощи функции GetCollection интерфейса IMongoClient, в которую передаётся название инересующей коллекции.

На клиентской стороне данные визуализируются в удобной форме, например, на странице TicketsPage, где отображается список билетов с такими параметрами, как ID, цена, время покупки, категория, ID сеанса, ID места и ID клиента. Пользователь может редактировать или добавлять данные через отдельную форму (TicketFormPage), где также реализованы валидация и удобный выбор клиента, места и сеанса.

public class ClientsController : ControllerBase

{

private readonly IMongoCollection<Client> \_clientsCollection;

public ClientsController(IMongoDatabase database)

{

\_clientsCollection = database.GetCollection<Client>("clients");

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Client>>> GetAllClients()

{

var clients = await \_clientsCollection.Find(\_ => true).ToListAsync();

return Ok(clients);

}

}

## 4.3 Основные части пользовательского интерфейса

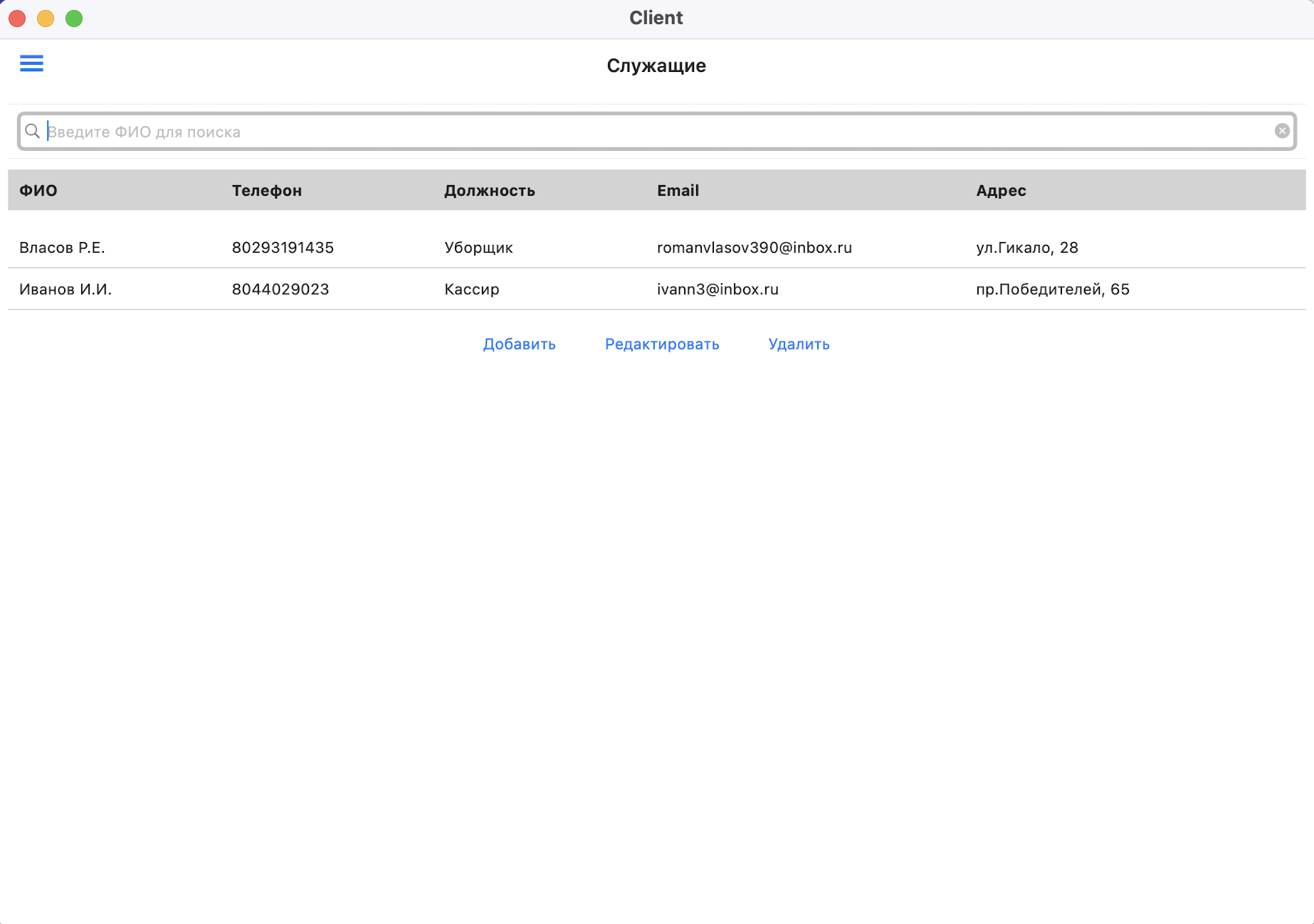
## Пользовательский интерфейс приложения представлен в виде главного окна с бургер-меню (Рисунок 4.3.2), предоставляющего доступ ко всем таблицам базы данных кинотеатра. Основные функции интерфейса:

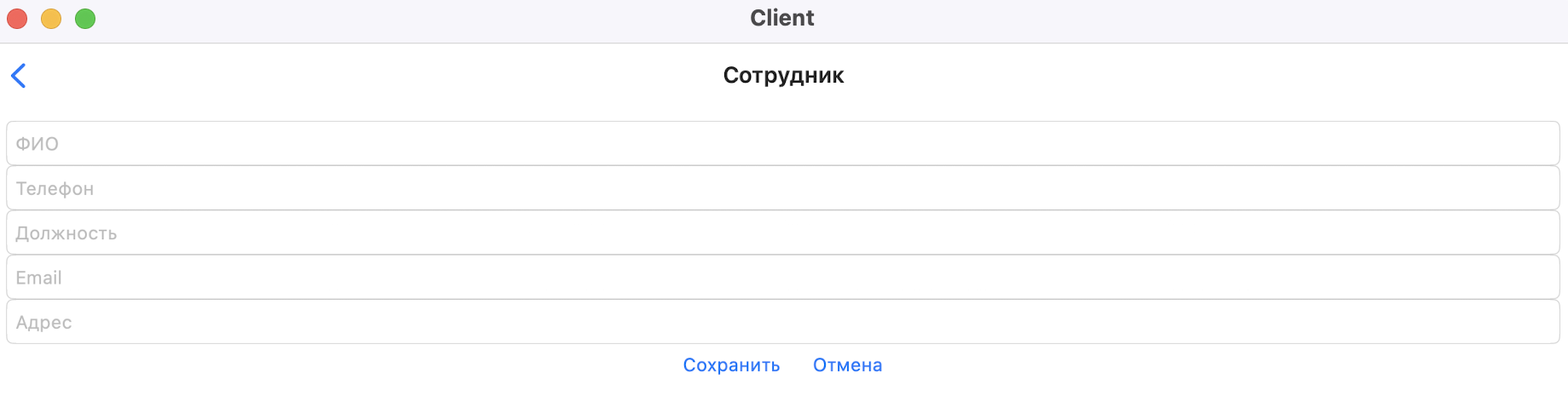
## 1.Бургер-меню слева в верхней части экрана: позволяет выбрать нужную таблицу (например, клиенты, сотрудники, фильмы, залы, билеты и т.д.).

## 2.Поисковая строка в верхней части окна: позволяет фильтровать данные в таблице, вводя ключевые слова или параметры.

## 3.Основная рабочая область: отображает выбранную таблицу с данными в виде списка, организованного по колонкам. Для каждой записи в таблице можно увидеть все основные поля, такие как ФИО, телефон, должность, email и адрес для сотрудников.

## 4.Кнопки управления под таблицей: Добавить: открывает новое окно для ввода информации о новой записи. Редактировать: открывает отдельное окно для внесения изменений в выбранную запись. Удалить: позволяет удалить выбранную запись из базы данных.

Рисунок 4.3.1 – Пример пользовательского интерфейса для таблицы “Служащие”

Рисунок 4.3.2 – Пример пользовательского интерфейса для добавления объекта “Служащие”

### 

### Рисунок 4.3.3 – Пример пользовательского интерфейса для редактирования объекта “Служащие”

### 

### Рисунок 4.3.4 – Пример пользовательского интерфейса для удаления объекта “Служащие”

## Листинг кода

### Листинг кода программы представлен в приложении А.

# РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## Развертывание приложения

1. **Установка и настройка:**

1.1 Убедитесь, что установлен JetBrains Rider. Если нет, скачайте и установите его с официального сайта JetBrains.

1.2 Убедитесь, что на вашем Mac установлены следующие зависимости:

• .NET SDK

• MongoDB

2. **Создание нового проекта:**

2.1 Запустите Rider.

2.2 В стартовом окне выберите “New Solution”.

2.3 Выберите шаблон “ASP.NET Core Web Application”.

2.4 Укажите имя проекта (например, “CinemaWebService”) и папку для сохранения. Нажмите “Create”.

2.5 В появившемся окне выберите шаблон “Web API” и убедитесь, что выбрана версия .NET 6 (или выше). Нажмите “Create”.

3. **Добавление библиотеки MongoDB.Driver:**

3.1 Откройте файл \*.csproj вашего проекта.

3.2 Добавьте в секцию <ItemGroup> следующую строку для установки MongoDB.Driver:

4 Сохраните файл и выполните команду dotnet restore в терминале для загрузки всех зависимостей.

### 5. Запуск приложения:

5.1 В меню Rider нажмите "Run" или выберите конфигурацию запуска в правом верхнем углу и нажмите кнопку запуска (зеленая стрелка).

5.2 Приложение запустится локально, и вы сможете получить доступ к API через браузер по адресу http://localhost:27017.

При необходимости отладки используйте встроенные инструменты Rider для точек останова и анализа логов.

## Работа с приложением

1. Для выбора таблицы из базы данных нажмите на иконку бургер-меню в верхнем левом углу экрана. В меню выберите нужную таблицу (например, “Клиенты”, “Сотрудники”, “Билеты” и т.д.). После выбора данные из выбранной таблицы появятся в основной рабочей области.

2. Для редактирования существующих записей нажмите кнопку “Редактировать” под таблицей. Откроется новое окно, где вы сможете внести изменения в выбранную запись.

3. Для удаления записей выберите нужную строку в таблице, а затем нажмите кнопку “Удалить”. Приложение запросит подтверждение перед удалением данных.

4. Для добавления новой записи нажмите кнопку “Добавить” под таблицей. Откроется отдельное окно для ввода данных новой записи.

# 6 ВЫВОД

В результате работы над лабораторной работой была создана NoSQL база данных организации «Кинотеатр» на основе MongoDB. Была выполнена миграция с Postgress SQL в MongoDB.

Были описаны технические требования для серверного и клиентского приложения с учетом специфики разработки на языках высокого уровня.

Программа для работы с базами данных MongoDB была успешно установлена на ПК.

Была разработана спецификация серверной части (backend) программы; написана серверной часть с использованием прикладного интерфейса СУБД MongoDB;

# ПРИЛОЖЕНИЕ А

(обязательное) Листинг кода

Файл Program.cs

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<IMongoClient>(sp =>

{

var mongoConnectionString = builder.Configuration.GetConnectionString("MongoDbConnection");

return new MongoClient(mongoConnectionString);

});

builder.Services.AddScoped(sp =>

{

var client = sp.GetRequiredService<IMongoClient>();

return client.GetDatabase("cinema\_mongo\_db");

});

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseAuthorization();

app.MapControllers();

app.Run();

Файл MigrationService.cs

public class MigrationService

{

private readonly ApplicationDbContext \_dbContext;

private readonly IMongoCollection<Client> \_clientsCollection;

private readonly IMongoCollection<Employee> \_employeesCollection;

private readonly IMongoCollection<Hall> \_hallsCollection;

private readonly IMongoCollection<Movie> \_moviesCollection;

private readonly IMongoCollection<Review> \_reviewsCollection;

private readonly IMongoCollection<Seat> \_seatsCollection;

private readonly IMongoCollection<Session> \_sessionsCollection;

private readonly IMongoCollection<SessionEmployee> \_sessionEmployeesCollection;

private readonly IMongoCollection<SessionMovie> \_sessionMoviesCollection;

private readonly IMongoCollection<Ticket> \_ticketsCollection;

public MigrationService(ApplicationDbContext dbContext, IConfiguration configuration)

{

\_dbContext = dbContext;

var mongoConnectionString = configuration.GetConnectionString("MongoDbConnection");

var mongoClient = new MongoClient(mongoConnectionString);

var mongoDatabase = mongoClient.GetDatabase("cinema\_mongo\_db");

\_clientsCollection = mongoDatabase.GetCollection<Client>("clients");

\_employeesCollection = mongoDatabase.GetCollection<Employee>("employees");

\_hallsCollection = mongoDatabase.GetCollection<Hall>("halls");

\_moviesCollection = mongoDatabase.GetCollection<Movie>("movies");

\_reviewsCollection = mongoDatabase.GetCollection<Review>("reviews");

\_seatsCollection = mongoDatabase.GetCollection<Seat>("seats");

\_sessionsCollection = mongoDatabase.GetCollection<Session>("sessions");

\_sessionEmployeesCollection= mongoDatabase.GetCollection<SessionEmployee>("sessionEmployees");

\_sessionMoviesCollection = mongoDatabase.GetCollection<SessionMovie>("sessionMovies");

\_ticketsCollection = mongoDatabase.GetCollection<Ticket>("tickets");

}

public async Task MigrateAllAsync()

{

var clients = \_dbContext.Clients.ToList();

if (clients.Any())

await \_clientsCollection.InsertManyAsync(clients);

var employees = \_dbContext.Employees.ToList();

if (employees.Any())

await \_employeesCollection.InsertManyAsync(employees);

var halls = \_dbContext.Halls.ToList();

if (halls.Any())

await \_hallsCollection.InsertManyAsync(halls);

var movies = \_dbContext.Movies.ToList();

if (movies.Any())

await \_moviesCollection.InsertManyAsync(movies);

var reviews = \_dbContext.Reviews.ToList();

if (reviews.Any())

await \_reviewsCollection.InsertManyAsync(reviews);

var seats = \_dbContext.Seats.ToList();

if (seats.Any())

await \_seatsCollection.InsertManyAsync(seats);

var sessions = \_dbContext.Sessions.ToList();

if (sessions.Any())

await \_sessionsCollection.InsertManyAsync(sessions);

var sessionEmployees = \_dbContext.SessionEmployees.ToList();

if (sessionEmployees.Any())

await \_sessionEmployeesCollection.InsertManyAsync(sessionEmployees);

var sessionMovies = \_dbContext.SessionMovies.ToList();

if (sessionMovies.Any())

await \_sessionMoviesCollection.InsertManyAsync(sessionMovies);

var tickets = \_dbContext.Tickets.ToList();

if (tickets.Any())

await \_ticketsCollection.InsertManyAsync(tickets);

}

}

}

Файл ClientsController.cs

namespace CinemaWebService.Controllers

{

[ApiController]

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

public class ClientsController : ControllerBase

{

private readonly IMongoCollection<Client> \_clientsCollection;

public ClientsController(IMongoDatabase database)

{

\_clientsCollection = database.GetCollection<Client>("clients");

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Client>>> GetAllClients()

{

var clients = await \_clientsCollection.Find(\_ => true).ToListAsync();

return Ok(clients);

}

[HttpGet("{id}")]

public async Task<ActionResult<Client>> GetClientById(int id)

{

var client = await \_clientsCollection.Find(c => c.ClientId == id).FirstOrDefaultAsync();

if (client == null)

{

return NotFound();

}

return Ok(client);

}

[HttpPost]

public async Task<ActionResult<Client>> CreateClient(Client client)

{

var maxId = await \_clientsCollection.AsQueryable().MaxAsync(c => (int?)c.ClientId) ?? 0;

client.ClientId = maxId + 1;

await \_clientsCollection.InsertOneAsync(client);

return CreatedAtAction(nameof(GetClientById), new { id = client.ClientId }, client);

}

[HttpPut("{id}")]

public async Task<IActionResult> UpdateClient(int id, Client updatedClient)

{

if (id != updatedClient.ClientId)

{

return BadRequest("Идентификаторы не совпадают");

}

var replaceResult = await \_clientsCollection.ReplaceOneAsync(c => c.ClientId == id, updatedClient);

if (replaceResult.MatchedCount == 0)

{

return NotFound();

}

return NoContent();

}

[HttpDelete("{id}")]

public async Task<IActionResult> DeleteClient(int id)

{

var deleteResult = await \_clientsCollection.DeleteOneAsync(c => c.ClientId == id);

if (deleteResult.DeletedCount == 0)

{

return NotFound();

}

return NoContent();

}

}

}