Microshop - Микросервисная система интернет-магазина

Описание проекта

Microshop - это микросервисная система интернет-магазина, разработанная для обеспечения надежной работы в условиях высоких нагрузок. Система состоит из двух основных микросервисов:

1. **Payments Service** - сервис для управления платежами
2. **Order Service** - сервис для управления заказами

Архитектура

Система построена на основе микросервисной архитектуры с четким разделением ответственности:

- **Payments Service** отвечает за:

- Создание счета пользователя

- Пополнение счета

- Просмотр баланса

- Обработку платежей за заказы

- **Order Service** отвечает за:

- Создание заказов

- Просмотр списка заказов

- Просмотр статуса заказа

- Асинхронную обработку оплаты заказа

Использованные паттерны и технологии

1. **Паттерны:**

- Transactional Outbox в Order Service для гарантированной доставки сообщений

- Transactional Inbox и Outbox в Payments Service для обеспечения семантики exactly once

- Оптимистичная блокировка для предотвращения коллизий при параллельных операциях

2. **Технологии:**

- .NET 8.0

- Entity Framework Core

- PostgreSQL

- Docker и Docker Compose для контейнеризации

Запуск проекта

Предварительные требования

1. Установите .NET 8.0 SDK с официального сайта: https://dotnet.microsoft.com/download/dotnet/8.0
2. Установите Docker Desktop: https://www.docker.com/products/docker-desktop
3. Убедитесь, что Docker Desktop запущен и работает

Запуск

Вот здесь у меня не очень получилось, постоянно выскакивала ошибка "Program does not contain a static 'Main' method suitable for an entry point" так и не понял, как ее решить, зато локально все запускается. (примерная реализация для докера также написана)

1. Клонируйте репозиторий:

```
git clone [url-репозитория]

cd Microshop
```

2. Запустите базы данных и RabbitMQ через Docker Compose:

```
docker compose up -d payments-db orders-db rabbitmq
```

3. Примените миграции для Order Service:

```
cd src/OrderService

dotnet ef database update
```

4. Примените миграции для Payments Service:

```
cd ../PaymentsService

dotnet ef database update
```

5. Запустите Order Service:

```
cd ../OrderService

dotnet run --urls=http://localhost:5002
```

6. В новом терминале запустите Payments Service:

```
cd src/PaymentsService

dotnet run --urls=http://localhost:5001
```

После запуска сервисы будут доступны по следующим адресам:

- Order Service: http://localhost:5002/swagger
- Payments Service: http://localhost:5001/swagger

Проверка работоспособности

1. Откройте Swagger UI для Payments Service (http://localhost:5001/swagger)

2. Создайте новый аккаунт через POST /api/accounts
3. Пополните баланс через POST /api/accounts/{accountId}/topup
4. Откройте Swagger UI для Order Service (http://localhost:5002/swagger)
5. Создайте новый заказ через POST /api/orders

Тестирование

Запуск тестов

Для запуска всех тестов выполните:

```
dotnet test --collect:"XPlat Code Coverage"
```

Тестовое покрытие

Проект включает тесты для обоих сервисов:

- OrderService.Tests
- PaymentsService.Tests

Текущее покрытие кода тестами превышает 25%.

API Endpoints

Payments Service

1. **Создание счета**

- POST /api/accounts

- Body: { "userId": "string" }

2. **Пополнение счета**

- POST /api/accounts/{accountId}/deposit

- Body: { "amount": decimal }

3. **Просмотр баланса**

- GET /api/accounts/{accountId}

Order Service

1. **Создание заказа**

   - POST /api/orders

   - Body: { "userId": "string", "items": [...] }

2. **Просмотр списка заказов**

   - GET /api/orders?userId={userId}

3. **Просмотр статуса заказа**

   - GET /api/orders/{orderId}

Реализация асинхронной коммуникации

В проекте реализована асинхронная коммуникация между сервисами с использованием паттерна Outbox:

1. **Order Service Outbox:**

   - При создании заказа сообщение о необходимости оплаты сохраняется в таблице OutboxMessages

   - Фоновый процесс периодически проверяет новые сообщения и отправляет их в Payments Service

   - После успешной отправки сообщение помечается как обработанное

2. **Payments Service Inbox:**

   - При получении сообщения о необходимости оплаты, оно сохраняется в таблице InboxMessages

   - Обработка сообщения происходит только если оно еще не было обработано

   - После успешной обработки сообщение помечается как обработанное

Это обеспечивает:

- Гарантированную доставку сообщений
- Семантику exactly once при обработке платежей
- Отсутствие потери сообщений при сбоях

Особенности реализации

1. **Обработка параллельных операций:**

- Использование оптимистичной блокировки при работе с балансом

- Транзакционная изоляция для предотвращения грязного чтения

- Атомарные операции для обновления баланса

2. **Обработка ошибок:**

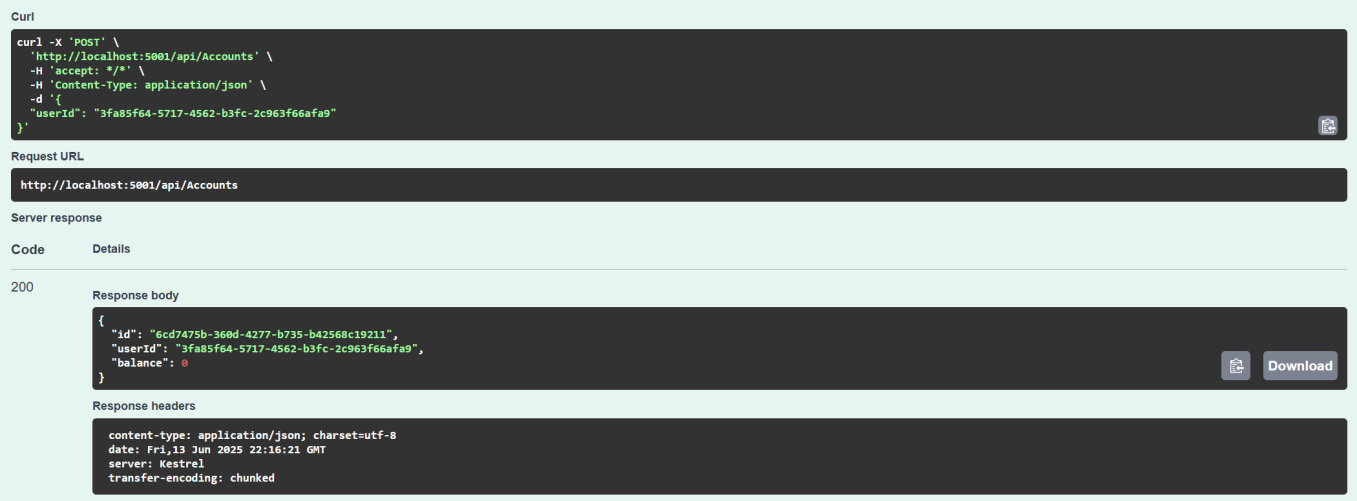
- Механизм повторных попыток для обработки временных сбоев

- Логирование всех операций для отладки

- Корректная обработка edge cases

Демонстрация ручного тестирования

Создали пользователя:



Пополнили баланс на 200:

Curl

```
curl -X 'POST' \
  'http://localhost:5001/api/Accounts/6cd7475b-360d-4277-b735-b42568c19211/topup' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "amount": 200
}'
```

Request URL

```
http://localhost:5001/api/Accounts/6cd7475b-360d-4277-b735-b42568c19211/topup
```

Server response

Code	Details
200	Response body

```
{
  "id": "6cd7475b-360d-4277-b735-b42568c19211",
  "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa9",
  "balance": 200
}
```

Download

Response headers

```
content-type: application/json; charset=utf-8
date: Fri,13 Jun 2025 22:16:36 GMT
server: Kestrel
transfer-encoding: chunked
```

Responses

Оформили заказ на 400 и на 20:

Curl

```
curl -X 'POST' \
  'http://localhost:5002/api/Orders' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa9",
  "amount": 20,
  "description": "string"
}'
```

Request URL

```
http://localhost:5002/api/Orders
```

Server response

Code	Details
200	Response body

```
{
  "id": "db6c09ca-18c0-476d-9f45-39349d303342",
  "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa9",
  "amount": 20,
  "description": "string",
  "status": 0,
  "createdAt": "2025-06-13T22:17:14.7273245Z"
}
```

Download

Response headers

```
content-type: application/json; charset=utf-8
date: Fri,13 Jun 2025 22:17:14 GMT
server: Kestrel
transfer-encoding: chunked
```

Responses

На 400 заказ не прошел, а на 20 прошел:

Code	Details
200	Response body

```
[
  {
    "id": "db6c09ca-18c0-476d-9f45-39349d303342",
    "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa9",
    "amount": 20,
    "description": "string",
    "status": 1,
    "createdAt": "2025-06-13T22:17:14.7273247Z"
  },
  {
    "id": "413cf16e-1a9e-4133-833a-8ece2744a819",
    "userId": "3fa85f64-5717-4562-b3fc-2c963f66afa9",
    "amount": 400,
    "description": "string",
    "status": 2,
    "createdAt": "2025-06-13T22:16:57.3407477Z"
  }
]
```

Download

Response headers

```
content-type: application/json; charset=utf-8
date: Fri,13 Jun 2025 22:17:22 GMT
server: Kestrel
transfer-encoding: chunked
```

Responses

Баланс обновился:

Curl

```
curl -X 'GET' \
  'http://localhost:5001/api/Accounts/6cd7475b-360d-4277-b735-b42568c19211/balance' \
  -H 'accept: */*'
```

Request URL

```
http://localhost:5001/api/Accounts/6cd7475b-360d-4277-b735-b42568c19211/balance
```

Server response

Code	Details
200	Response body

```
{
  "balance": 180
}
```

Download

Response headers

```
content-type: application/json; charset=utf-8
date: Fri,13 Jun 2025 22:17:27 GMT
server: Kestrel
transfer-encoding: chunked
```

Responses