RapidPay Payment Service

This project is a POC implementation of a payment service and is not a final product.

The project is built on a microservices architecture with distributed event-driven transaction model with eventual consistency. The primary focus of this architecture is on:

* Scalability – ensuring horizontal scalability through containerized microservices and distributed caching. It can be further enhanced with service load balancing, orchestration and auto-scaling mechanisms.
* Performance – the system is designed to efficiently handle high transaction loads while maintaining low response times and optimal resource utilization. Performance optimizations focus on reducing database load, minimizing synchronous dependencies, and leveraging distributed processing.
* Fault tolerance – the system ensures fault tolerance through event-driven processing and transaction rollback. Failures do not cause data loss and services remain operational.

This service is designed as a high-performance solution for real-time payment transaction processing.

Technology Stack

Backend: .NET 8, C#, MediatR, CQRS, EF

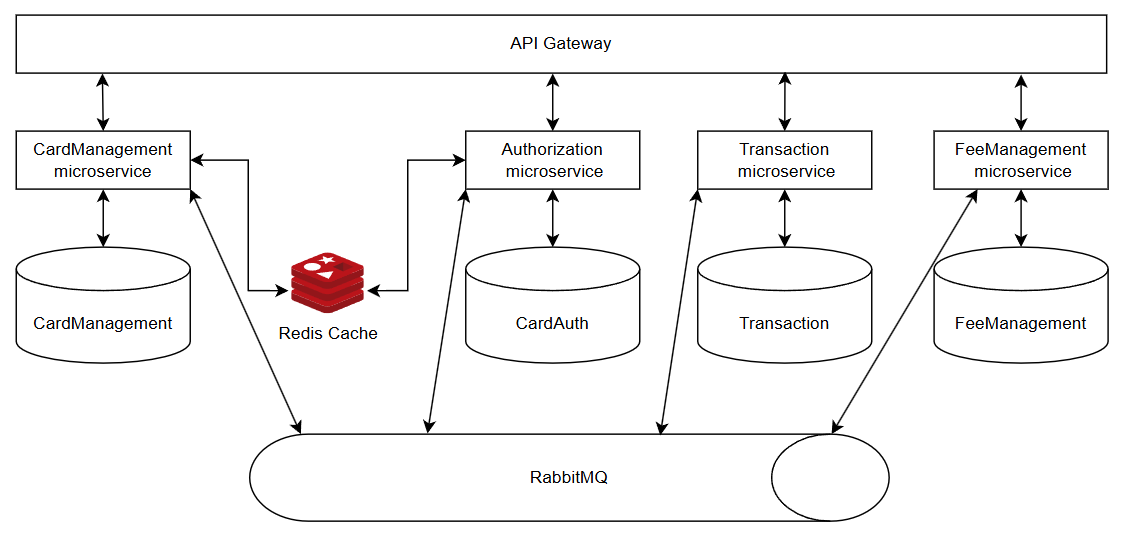
Communication: RabbitMQ, MassTransit, REST API

Storage: MS SQL Server, Redis

API Gateway: Ocelot

Containerization: Docker

Architecture



Endpoints

RapidPay.CardManagement service

Responsible for card management including balance changes.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | api/v1/cards/create | Create a card |
| GET | api/v1/cards/{cardNumber} | Retrieve card data |
| PUT | api/v1/cards/update | Update card details |

RapidPay.Authorization service

Responsible for card and transaction authorization.

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | api/v1/authorization/transaction | Authorizes a transaction and initiates the transfer |
| POST | api/v1/authorization/card | Verifies card status for authorization |

RapidPay.ApiGateaway

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **Endpoint** | **Description** |
| POST | api/v1/auth/login | Generates JWT token |

Communication

|  |  |  |  |
| --- | --- | --- | --- |
| **Event** | **Producer** | **Consumer** | **Description** |
| CardUpdatedEvent | CardManagement | Authorization, CardManagement | *Card details changed* |
| DepositFundsEvent | Transaction | CardManagement | *Funds deposit requested* |
| FundsWithdrawnEvent | CardManagement | Transaction | *Funds withdrawn from the sender* |
| RefundRequestedEvent | Transaction | CardManagement | *Funds refund requested* |
| TransactionAuthorizedEvent | Authorization | Transaction | *Transaction authorized* |
| TransactionCompletedEvent | CardManagement | Transaction | *Transaction completed* |
| TransactionFailedEvent | CardManagement, Transaction | Transaction | *Transaction failed* |
| TransactionRefundedEvent | CardManagement | Transaction | *Funds are refunded to the sender* |
| WithdrawFundsEvent | Transaction | CardManagement | *Withdrawal request from sender* |
| FeeUpdatedEvent | FeeManagement | CardManagement | *Fee value updated* |

Transaction statuses

* Pending – transaction initialized and waiting for processing
* Authorized – transaction authorized
* Funds withdrawn – funds were withdrawn from sender
* Completed – transaction successful
* Failed – transaction failed
* Refund pending – transaction failed after funds were withdrawn from sender
* Refunded – funds are refunded to sender

Transaction concurrency

Transaction concurrency achieved through Redis Distributed Lock:

* It allows safely locking card processing to prevent race conditions
* It easily scales in a Redis cluster
* It operates faster than database transactions

Concern: lock has TTL so if transaction hangs at one of the steps that can disrupt data state.

Solution: two-phase commit. First, funds are reserved in Redis. If the transaction succeeds, they are deducted; if it fails, the reservation is canceled. This prevents race conditions and ensures consistency. But this will not be part of current POC for now.

Responses

This POC uses standard HTTP responses. Some operations return structured response objects as CardResponseDto.

{

"cardNumber": "123451234512345",

"balance": 1000.00,

"creditLimit": 500.00,

"usedCredit": 0.00

}

Use cases

Use case 1. Successful transaction.

1. Create next cards using POST api/v1/cards/create

|  |  |
| --- | --- |
| {  "cardNumber": "641430385215810",  "initialBalance": 50,  "creditLimit": 100  } | {  "cardNumber": "541430385215810",  "initialBalance": 50,  "creditLimit": 50  } |

1.1 Cards are created.



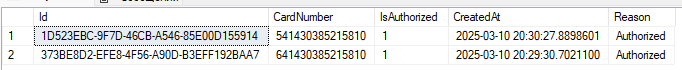
*CardManagement.dbo.Cards*

1.2 Changes in the balance are logged.



*CardManagement.dbo.CardsLogs*

1.3 After creating a card, authorization is automatically requested. Cards are authorized.

 *CardAuth.dbo.AuthorizationLogs*

1.4 The statuses are changed to active after authorization.



*CardAuth.dbo.CardAuthorizations*

1. Init transaction using POST api/v1/authorization/transaction

{

"senderCardNumber": "541430385215810",

"receiverCardNumber": "641430385215810",

"amount": 75

}

2.1 Changes in the balance are logged.

 *CardManagement.dbo.CardsLogs*

2.2 Cards updated, credit is partially used for 541430385215810.

 *CardManagement.dbo.Cards*

2.3 Transaction completed successfully.

 *Transaction.dbo.Transactions*

Use case 2. Failed transaction but funds withdrawn from sender.

1. Create cards as in step 1 of Use case 1.
2. Simulate transaction error to produce TransactionFailedEvent with NeedRefund = true.
3. Init transaction as in step 2 of Use case 1.

3.1 Balance is not changed.



*CardManagement.dbo.Cards*

3.2 Funds were withdrawn but refunded due to inability to complete transaction. *CardManagement.dbo.CardsLogs*

3.3 Transaction completed with Refunded status.



*Transaction.dbo.Transactions*

Configuration

There no specific configurations of Redis\RabbitMQ, just run the servers and make sure the files below are configured.

ocelot.json

To configure Ocelot API Gateway, define routes that map incoming requests (UpstreamPathTemplate) to downstream services (DownstreamPathTemplate).

{

"Routes": [

{

"DownstreamPathTemplate": "/api/auth/login",

"DownstreamScheme": "https",

"DownstreamHostAndPorts": [{"Host": "<authorization service host>", "Port": <host port> }],

"UpstreamPathTemplate": "/api/auth/login",

"UpstreamHttpMethod": [ "POST" ]

},

...

],

"GlobalConfiguration": {

"BaseUrl": "<base url>"

}

}

appsettings.json

Each microservice should have its own database configuration while sharing Redis and RabbitMQ for caching and messaging.

"ConnectionStrings": {

"DefaultConnection": ""

},

"Redis": {

"ConnectionString": "",

"CacheDurationSeconds": 0

},

"RabbitMQ": {

"HostName": "",

"UserName": "",

"Password": ""

}

Each microservice and Gateway API should have JWT configured.

"Jwt": {

"Key": "<128-bit key>",

"Issuer": "<issuer>",

"Audience": "<audience>"

}

What can be better?

* Approach works well for POC but could be improved by adding Saga Orchestration for stronger consistency.
* Add unit and integration tests.
* Improve API responses and add middleware for handling.
* Add JWT refresh tokens.
* Two-phase commit for transactions.
* Secure sensitive data.
* Improve logging, add file logging.