

ACME Risk Analysis

Visão Geral

Esta solução implementa um sistema de análise de risco de transações financeiras baseado em microsserviços, utilizando **Arquitetura Hexagonal (Ports and Adapters)** e **autenticação JWT** para comunicação segura entre serviços. Desenvolvida em Java 17 com Spring Boot e Docker.

Arquitetura

Modulos do Sistema

A solução é composta por três módulos principais:

- Risk Analysis Service** (Porta 8080): Orquestra o fluxo de análise de risco
- Lists Service** (Porta 8081): Gerencia listas permissivas e restritivas
- Decision Engine Service** (Porta 8082): Aplica regras de negócio e calcula score de risco

Princípios da Arquitetura Hexagonal

A solução foi desenvolvida para seguir os princípios da Arquitetura Hexagonal, onde:

- Domínio Central:** Contém a lógica de negócio pura, independente de frameworks e tecnologias externas
- Portas (Ports):** Interfaces que definem contratos de entrada e saída
- Adaptadores (Adapters):** Implementações concretas que conectam o domínio ao mundo externo

Estrutura dos Serviços

Cada microsserviço segue a estrutura hexagonal:

```
src/main/java/com/acme/[service]/
├── application/           # Camada de aplicação
│   ├── port/             # Portas de entrada e saída
│   └── service/          # Serviços de aplicação (orquestração)
├── domain/               # Camada de domínio (lógica de negócio)
│   ├── model/            # Entidades de domínio
│   └── service/          # Serviços de domínio
├── infrastructure/       # Camada de infraestrutura
│   ├── adapter/          # Adaptadores de saída
│   ├── controller/       # Adaptadores de entrada (REST)
│   ├── config/           # Configurações
│   └── security/         # Componentes de segurança JWT
└── common/              # DTOs e classes comuns
```

Microsserviços

1. Risk Analysis Service (Porta 8080)

- **Responsabilidade:** Orquestrador principal do fluxo de análise de risco
- **Portas de Entrada:** RiskAnalysisPort
- **Portas de Saída:** ListsServicePort , DecisionEngineServicePort
- **Segurança:** Gera tokens JWT para comunicação com outros serviços

2. Lists Service (Porta 8081)

- **Responsabilidade:** Gerencia listas permissivas e restritivas
- **Portas de Entrada:** ListsPort
- **Portas de Saída:** ListsRepositoryPort
- **Segurança:** Valida tokens JWT recebidos

3. Decision Engine Service (Porta 8082)

- **Responsabilidade:** Aplica regras de negócio e calcula score de risco
- **Portas de Entrada:** DecisionEnginePort

- **Portas de Saída:** `RuleRepositoryPort`
- **Segurança:** Valida tokens JWT recebidos

Segurança JWT

Implementação

- **Chave Secreta:** Compartilhada entre todos os serviços
- **Geração:** Risk Analysis Service gera tokens para chamadas internas
- **Validação:** Lists Service e Decision Engine Service validam tokens recebidos
- **Filtros:** `JwtAuthenticationFilter` intercepta e valida requisições

Fluxo de Autenticação

1. Risk Analysis Service recebe requisição externa (sem JWT)
2. Gera token JWT para chamadas internas
3. Inclui token no header `Authorization: Bearer <token>`
4. Serviços de destino validam o token antes de processar

Tecnologias Utilizadas

- **Java 17**
- **Spring Boot 3.2.0**
- **Spring Security** (para JWT)
- **Spring Data JPA** (Decision Engine Service)
- **H2 Database** (em memória)
- **Maven** (gerenciamento de dependências)
- **Docker & Docker Compose**
- **JWT (JSON Web Tokens)** - biblioteca `jjwt`

Execução

Pré-requisitos

- Docker e Docker Compose instalados
- Java 17+ (para desenvolvimento local)
- Maven 3.6+ (para desenvolvimento local)

Executando a aplicação

1. Clone o repositório ou extraia o arquivo ZIP
2. Navegue até o diretório raiz do projeto
3. Execute o comando:

```
docker-compose up --build
```

1. Aguarde todos os serviços subirem (pode levar alguns minutos na primeira execução)

Verificando se os serviços estão funcionando

- Risk Analysis Service: <http://localhost:8080/risk-analysis/health>
- Lists Service: <http://localhost:8081/lists/health>
- Decision Engine Service: <http://localhost:8082/decision-engine/health>

Executar com Docker Compose

```
# Construir e executar todos os serviços
docker-compose up --build

# Executar em background
docker-compose up -d --build

# Parar os serviços
docker-compose down
```

Executar Localmente (Desenvolvimento)

```
# Terminal 1 - Decision Engine Service
cd decision-engine-service
mvn spring-boot:run

# Terminal 2 - Lists Service
cd lists-service
mvn spring-boot:run

# Terminal 3 - Risk Analysis Service
cd risk-analysis-service
mvn spring-boot:run
```

Parar a Aplicação

```
docker-compose down
```

Testes

Script de Teste Automatizado

```
# jq (JSON Query) e curl (HTTP Client) devem estar instalados

# Executar testes das doc do Swagger
./test_swagger.sh

# Executar testes das APIs
./test_api.sh

# Executar testes das APIs com JWT
./test_api_jwt.sh
```

Testes Manuais

1. Análise de Risco (Endpoint Principal)

```
curl -X POST http://localhost:8080/risk-analysis \
-H "Content-Type: application/json" \
-d '{
  "cpf": "12345678901",
  "ip": "192.168.1.100",
  "deviceId": "device123",
  "txType": "PIX",
  "txValue": 1500.00
}'
```

2. Health Checks

```
curl http://localhost:8080/risk-analysis/health
curl http://localhost:8081/lists/health
curl http://localhost:8082/decision-engine/health
```

3. Gerenciamento de Regras (com JWT)

```
# Listar regras
curl -X GET http://localhost:8082/rules \
-H "Authorization: Bearer <JWT_TOKEN>"

# Criar nova regra
curl -X POST http://localhost:8082/rules \
-H "Content-Type: application/json" \
-H "Authorization: Bearer <JWT_TOKEN>" \
-d '{
  "name": "Nova Regra",
  "description": "Descrição da regra",
  "txType": "PIX",
  "condition": "
{ \"type\": \"value_range\", \"min\": \"1000\", \"max\": \"5000\" }",
  "points": 100,
  "active": true
}'
```

APIs Disponíveis

Risk Analysis Service

POST /risk-analysis

Realiza análise de risco de uma transação.

Request Body:

```
{
  "cpf": "12345678901",
  "ip": "192.168.1.1",
  "deviceId": "550e8400-e29b-41d4-a716-446655440000",
  "txType": "PIX",
  "txValue": 1500.00
}
```

Response:

```
{
  "txDecision": "Aprovada"
}
```

Lists Service

POST /lists/check

Verifica se CPF, IP ou Device ID estão em listas.

Request Body:

```
{
  "cpf": "12345678901",
  "ip": "192.168.1.1",
  "deviceId": "550e8400-e29b-41d4-a716-446655440000"
}
```

POST /lists/reload

Recarrega as listas a partir do arquivo de configuração.

Decision Engine Service

POST /decision-engine/calculate-score

Calcula o score de risco baseado nas regras configuradas.

Gerenciamento de Regras (CRUD)

- GET /decision-engine/rules - Lista todas as regras
- POST /decision-engine/rules - Cria uma nova regra
- GET /decision-engine/rules/{id} - Busca regra por ID
- PUT /decision-engine/rules/{id} - Atualiza uma regra
- DELETE /decision-engine/rules/{id} - Exclui uma regra

Configuração de Regras

O sistema vem pré-configurado com regras padrão baseadas nos requisitos do desafio:

Regras de Valor da Transação (DEFAULT)

- R0, 01 — *R* 300: +200 pontos
- R301 — *R* 5.000: +300 pontos
- R5.001 — *R* 20.000: +400 pontos
- Acima de R\$ 20.000: +500 pontos

Regras de Listas (DEFAULT)

- CPF na lista permissiva: -200 pontos
- CPF na lista restritiva: +400 pontos
- IP na lista restritiva: +400 pontos
- Device na lista restritiva: +400 pontos

Regras Específicas para CARTÃO

- R0, 01 — *R* 300: +300 pontos (ao invés de 200)

- CPF na lista permissiva: -300 pontos (ao invés de -200)

Configuração de Score

- **Risco Baixo:** 1 - 399 pontos → Aprovada
- **Risco Médio:** 400 - 699 pontos → Aprovada
- **Risco Alto:** 700+ pontos → Negada

Dados de Teste

Listas Permissivas (CPF)

- 12345678901
- 98765432100
- 11111111111

Listas Restritivas (CPF)

- 99999999999
- 88888888888
- 77777777777
- 12345678901 (também está na permissiva)

Listas Restritivas (IP)

- 192.168.1.100
- 10.0.0.50
- 172.16.0.25

Listas Restritivas (Device)

- 550e8400-e29b-41d4-a716-446655440000
- 6ba7b810-9dad-11d1-80b4-00c04fd430c8

- 6ba7b811-9dad-11d1-80b4-00c04fd430c8

Exemplo de Teste Completo

```
curl -X POST http://localhost:8080/risk-analysis \
-H "Content-Type: application/json" \
-d '{
  "cpf": "12345678901",
  "ip": "192.168.1.1",
  "deviceId": "550e8400-e29b-41d4-a716-446655440001",
  "txType": "PIX",
  "txValue": 1500.00
}'
```

Monitoramento

Logs

- Todos os serviços geram logs detalhados
- Nível DEBUG habilitado para desenvolvimento
- Logs incluem informações de JWT e fluxo de requisições

Como Visualizar os Logs

Os logs de cada serviço podem ser visualizados com:

```
docker-compose logs -f [nome-do-serviço]
```

Exemplo:

```
docker-compose logs -f risk-analysis-service
```

Health Checks

- Cada serviço expõe endpoint `/health`
- Docker Compose configurado com health checks automáticos

Desenvolvimento

Adicionando Novas Regras

1. Criar JSON de condição no formato apropriado
2. Usar endpoint POST `/rules` do Decision Engine Service
3. Regras são aplicadas automaticamente no próximo cálculo

Modificando Listas

1. Editar arquivo `lists-service/src/main/resources/lists.json`
2. Usar endpoint POST `/lists/reload` para recarregar

Testando Localmente

1. Usar Postman ou curl para testes
2. Incluir header `Authorization: Bearer <token>` para serviços protegidos
3. Verificar logs para debugging

Arquivos Importantes

- `docker-compose.yml` : Orquestração dos serviços
- `install_jq.sh` : Script de instalação do jq
- `test_api.sh` : Script de testes automatizados
- `test_swagger.sh` : Script de testes automatizados
- `test_api_jwt.sh` : Script de testes automatizados
- `architecture_diagram.png` : Diagrama da arquitetura
- `solution-design.drawio` : Design da solução produtiva
- `lists.json` : Dados das listas permissivas/restritivas
- `data.sql` : Dados iniciais das regras de negócio

Considerações de Segurança

- JWT com chave secreta compartilhada (adequado para ambiente interno)
- Tokens com expiração configurável
- Validação rigorosa de tokens em todos os endpoints protegidos
- Health checks públicos para monitoramento
- Logs não expõem informações sensíveis

Considerações de Desenvolvimento

Arquitetura Hexagonal

- Separação clara entre domínio, aplicação e infraestrutura
- Lógica de negócio isolada e testável
- Facilita manutenção e evolução do código

Segurança JWT

- Comunicação segura entre microserviços
- Autenticação baseada em tokens
- Proteção contra acesso não autorizado

Melhorias de Design

- Código mais modular e organizados
- Responsabilidades bem definidas
- Facilita testes unitários e de integração

Documentação das APIs (Swagger/OpenAPI)

Acesso à Interface Swagger UI

Cada microsserviço possui sua própria documentação interativa Swagger UI, acessível após a execução dos serviços:

URLs de Acesso:

- **Risk Analysis Service:** <http://localhost:8080/swagger-ui.html>
- **Lists Service:** <http://localhost:8081/swagger-ui.html>
- **Decision Engine Service:** <http://localhost:8082/swagger-ui.html>

Funcionalidades da Interface Swagger:

- **Documentação Interativa:** Visualização de todos os endpoints disponíveis
- **Teste de APIs:** Execução de requisições diretamente pela interface
- **Esquemas de Dados:** Visualização dos DTOs de entrada e saída
- **Autenticação JWT:** Interface para inserir tokens Bearer nos serviços protegidos
- **Exemplos de Requisições:** Payloads de exemplo para facilitar os testes

Configuração de Segurança no Swagger:

Os serviços **Lists Service** e **Decision Engine Service** requerem autenticação JWT. Na interface Swagger:

1. Clique no botão **"Authorize"** no topo da página
2. Insira o token JWT no formato: `Bearer <seu-token-jwt>`
3. Clique em **"Authorize"** para aplicar o token a todas as requisições

Documentação OpenAPI (JSON):

As especificações OpenAPI em formato JSON estão disponíveis em: - **Risk Analysis Service:** <http://localhost:8080/v3/api-docs> - **Lists Service:** <http://localhost:8081/v3/api-docs> - **Decision Engine Service:** <http://localhost:8082/v3/api-docs>

Exemplos de Uso via Swagger UI

1. Testando o Risk Analysis Service:

1. Acesse <http://localhost:8080/swagger-ui.html>
2. Expanda o endpoint `POST /risk-analysis`
3. Clique em **"Try it out"**
4. Use o payload de exemplo:

```
{
  "cpf": "12345678901",
  "ip": "192.168.1.100",
  "deviceId": "device123",
  "txType": "PIX",
  "txValue": 1500.00
}
```

1. Clique em **"Execute"** para ver a resposta

2. Testando o Lists Service (com JWT):

1. Acesse <http://localhost:8081/swagger-ui.html>
2. Clique em **"Authorize"** e insira um token JWT válido
3. Expanda o endpoint `POST /lists/check`
4. Execute a requisição com os dados desejados

3. Gerenciando Regras no Decision Engine Service:

1. Acesse <http://localhost:8082/swagger-ui.html>
2. Configure a autenticação JWT
3. Use os endpoints de CRUD de regras (`/rules`)
4. Teste o cálculo de score via `POST /decision-engine/calculate-score`

Gerando Token JWT

Adicionado endpoint GET para gerar tokens JWT baseado em `client_id` específicos, facilitando os testes através da interface do Swagger UI.

Implementação Técnica

1. Endpoints Criados para Autenticação

GET /auth/token

- **Descrição:** Gera token JWT válido baseado no `client_id`
- **Parâmetro:** `clientId` (query parameter)
- **Resposta:** Token JWT com informações do serviço

GET /auth/client-ids

- **Descrição:** Lista todos os `client_ids` válidos e seus serviços
- **Resposta:** Mapeamento `client_id` → nome do serviço

2. Client IDs Configurados

Client ID	Serviço
7f073c43-d91b-4138-b7f0-85f8d73490bf	lists-service
a1b2c3d4-e5f6-7890-abcd-ef1234567890	decision-engine-service
12345678-90ab-cdef-1234-567890abcdef	risk-analysis-service

3. Arquivos Criados

Nomes dos Arquivos:

- `TokenRequest.java` - DTO para requisição de token
- `TokenResponse.java` - DTO para resposta de token
- `TokenService.java` - Serviço de domínio para gerenciamento de tokens
- `TokenController.java` - Controller com endpoints de autenticação
- `SecurityConfig.java` - Adicionada rota `/auth/**` como pública
- `OpenApiConfig.java` - Documentação atualizada com instruções de uso

Testes Realizados

✅ Testes de Funcionalidade

```
# Gerar token válido
curl "http://localhost:8080/auth/token?clientId=7f073c43-d91b-4138-b7f0-85f8d73490bf"
# Resposta: {"token":"eyJ...", "tokenType":"Bearer", "expiresIn":3600, ...}

# Listar client IDs válidos
curl "http://localhost:8080/auth/client-ids"
# Resposta: {"7f073c43...":"lists-service", "a1b2c3d4...":"decision-engine-service", ...}

# Testar client ID inválido
curl "http://localhost:8080/auth/token?clientId=invalid-client-id"
# Resposta: {"error":"Client ID inválido: invalid-client-id"}
```

✅ Swagger UI Acessível

- Risk Analysis Service: <http://localhost:8080/swagger-ui.html> ✅
- Documentação completa dos novos endpoints ✅
- Exemplos de client_ids na documentação ✅

Como Usar no Swagger UI

Passo 1: Gerar Token

1. Acesse <http://localhost:8080/swagger-ui.html>
2. Vá para a seção "Autenticação"
3. Use o endpoint `GET /auth/token`
4. Insira um client_id válido (ex: `7f073c43-d91b-4138-b7f0-85f8d73490bf`)
5. Execute e copie o token retornado

Passo 2: Usar Token nas APIs Protegidas





1. Vá para qualquer endpoint protegido (Lists Service ou Decision Engine)
2. Clique em "Authorize" no Swagger UI

3. Cole o token no formato: `Bearer <token>`
4. Agora pode testar as APIs protegidas!

Exemplo de Resposta do Token

```
{
  "token":
  "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJzZXJ2aWNlLWVubW11bmljYXRpb24iLCJpc3MiOiJsaXN0cy1",
  "tokenType": "Bearer",
  "expiresIn": 3600,
  "clientId": "7f073c43-d91b-4138-b7f0-85f8d73490bf",
  "serviceName": "lists-service"
}
```

Segurança

-  **Validação de Client ID:** Apenas client_ids pré-configurados são aceitos
-  **Tokens com Expiração:** Tokens válidos por 1 hora (3600 segundos)
-  **Endpoint Público:** `/auth/**` acessível sem autenticação para facilitar testes
-  **Tratamento de Erros:** Mensagens claras para client_ids inválidos

Benefícios da Rota de Autenticação

1. **Facilita Testes:** Não precisa mais gerar tokens manualmente
2. **Interface Amigável:** Tudo integrado no Swagger UI
3. **Segurança Mantida:** Apenas client_ids válidos funcionam
4. **Documentação Clara:** Instruções diretas na interface
5. **Produtividade:** Desenvolvedores podem testar APIs rapidamente

Desenho da Solução

Visão Geral da Arquitetura

O projeto inclui um **Desenho da Solução** detalhado que aborda aspectos críticos de **resiliência**, **escalabilidade** e **alto desempenho** para ambientes de produção.

Arquivos do Desenho:

- `solution-design.drawio` : Diagrama visual da arquitetura (editável no draw.io)
- `SOLUTION_DESIGN.md` : Documentação técnica detalhada da solução

Como Visualizar o Desenho:

1. **Online:** Acesse draw.io e abra o arquivo `solution-design.drawio`
2. **Desktop:** Instale o draw.io desktop e abra o arquivo
3. **VS Code:** Use a extensão "Draw.io Integration" para visualizar diretamente no editor

Principais Aspectos Abordados:

Alto Volume de Transações (TPS)

- **Arquitetura Horizontalmente Escalável:** Múltiplas instâncias de cada microsserviço
- **Load Balancer/API Gateway:** Distribuição inteligente de carga
- **Auto-scaling:** Baseado em métricas de CPU, latência e throughput
- **Target:** 10,000+ transações por segundo

Resiliência a Falhas

- **Circuit Breaker Pattern:** Proteção contra cascata de falhas
- **Health Checks:** Monitoramento contínuo da saúde dos serviços
- **Graceful Degradation:** Funcionamento parcial em caso de falhas
- **Multi-AZ Deployment:** Tolerância a falhas de infraestrutura

- **Disaster Recovery:** RTO < 5min, RPO < 1min

⚡ Escalabilidade

- **Stateless Design:** Serviços sem estado para escalabilidade dinâmica
- **Container Orchestration:** Kubernetes com Horizontal Pod Autoscaler
- **Database Scaling:** Read replicas, connection pooling, sharding
- **Cache Distribuído:** Redis Cluster para performance otimizada

📊 Alto Desempenho

- **Targets de Performance:** Latência < 100ms P95, Disponibilidade 99.9%
- **Cache Multi-Layer:** L1 (local) + L2 (distribuído)
- **Processamento Assíncrono:** Message queues para desacoplamento
- **Otimizações de Rede:** HTTP/2, connection pooling, compression

Componentes da Arquitetura de Produção:

Infraestrutura:

- **Load Balancer:** NGINX, AWS ALB, Kong API Gateway
- **Container Platform:** Kubernetes com auto-scaling
- **Database:** PostgreSQL/MySQL com cluster master/slave
- **Cache:** Redis Cluster distribuído
- **Message Queue:** Kafka/RabbitMQ para processamento assíncrono





Monitoramento e Observabilidade:

- **Métricas:** Prometheus + Grafana
- **Logs:** ELK Stack (Elasticsearch, Logstash, Kibana)
- **Tracing:** Jaeger para distributed tracing
- **Alerting:** PagerDuty, Slack para notificações críticas

Segurança:

- **Network Security:** VPC, Security Groups, WAF
- **Data Protection:** Encryption at rest/transit, PII masking
- **Authentication:** JWT com algoritmos seguros (RS256)
- **Compliance:** GDPR, PCI-DSS considerations

Evolução da Implementação Atual:

A implementação atual (desenvolvimento) já incorpora os fundamentos da arquitetura de produção: -  **Arquitetura Hexagonal:** Base sólida para escalabilidade -  **Segurança JWT:** Comunicação segura entre serviços -  **Health Checks:** Monitoramento básico implementado -  **Containerização:** Docker pronto para orquestração

Próximos Passos para Produção:

1. **Implementar Circuit Breakers:** Resilience4j ou Hystrix
2. **Configurar Auto-scaling:** Kubernetes HPA
3. **Setup de Monitoramento:** Prometheus/Grafana stack
4. **Database Externa:** Migração do H2 para PostgreSQL/MySQL
5. **Cache Distribuído:** Implementação do Redis Cluster
6. **CI/CD Pipeline:** Automação de deploy e testes

Desenvolvido por Igor Meira - meira.igor@gmail.com