Sistema de Gerenciamento de Arquivos

Arquitetura com Gateway, REST e SOAP

Autor: Pedro Henrique
Data: Novembro de 2025
Repositório: github.com/pedrohcdsouza/fileserver-with-gateway

---

Sumário

1. Introdução
2. Conceitos Fundamentais
    - REST API
    - SOAP API
    - API Gateway
    - HATEOAS
3. Arquitetura do Sistema
4. Componentes
5. Fluxo de Dados
6. Implementação Técnica
7. Demonstração Prática
8. Conclusão

---

1. Introdução

Este documento apresenta um sistema completo de gerenciamento de arquivos desenvolvido com arquitetura de microserviços, integrando diferentes paradigmas de comunicação: **REST** e **SOAP**.

Objetivo

Demonstrar a integração de APIs REST e SOAP através de um API Gateway que implementa o conceito de HATEOAS (Hypermedia as the Engine of Application State).

Caso de Uso

Sistema de transmissão e gerenciamento de arquivos que permite:

- Upload de arquivos
- Download de arquivos
- Listagem de arquivos
- Obtenção de metadados
- Exclusão de arquivos

---

2. Conceitos Fundamentais

2.1 REST API (Representational State Transfer)

REST é um estilo arquitetural para sistemas distribuídos que utiliza o protocolo HTTP.

Características:

- **Stateless**: Cada requisição é independente
- **Recursos**: Identificados por URLs (ex: `/files/123`)
- **Métodos HTTP**: GET, POST, PUT, DELETE
- **Formatos**: JSON, XML
- **Simplicidade**: Fácil de entender e implementar

Exemplo de Requisição REST:

```
GET /files/abc123 HTTP/1.1
Host: localhost:8000
Accept: application/json

Response:
{
  "id": "abc123",
  "filename": "documento.pdf"
}
```

Vantagens:

- ☑ **Simplicidade** - Fácil de usar e entender
- ☑ **Escalabilidade** - Stateless facilita escalabilidade horizontal
- ☑ **Flexibilidade** - Suporta múltiplos formatos
- ☑ **Cache** - Suporte nativo a cache HTTP
- ☑ **Amplamente adotado** - Grande comunidade e ferramentas

2.2 SOAP API (Simple Object Access Protocol)

SOAP é um protocolo de comunicação baseado em XML para troca de informações estruturadas.

Características:

- **Fortemente tipado**: Contratos definidos via WSDL
- **Protocolo formal**: Especificação rígida
- **XML obrigatório**: Todas as mensagens em XML
- **Independente de transporte**: HTTP, SMTP, TCP
- **Segurança integrada**: WS-Security

Exemplo de Requisição SOAP:

```
POST /soap HTTP/1.1
Host: localhost:8001
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetFileMetadata xmlns="urn:FileService">
      <id>1</id>
    </GetFileMetadata>
  </soap:Body>
</soap:Envelope>

Response:
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetFileMetadataResponse>
      <found>true</found>
      <name>documento.pdf</name>
      <size>204800</size>
      <type>application/pdf</type>
    </GetFileMetadataResponse>
  </soap:Body>
</soap:Envelope>
```

WSDL (Web Services Description Language):

Define o contrato do serviço:

```
<definitions name="FileService">
  <!-- Tipos de dados -->
  <message name="GetFileMetadataRequest">
    <part name="id" type="xsd:int"/>
  </message>

  <!-- Operações disponíveis -->
  <portType name="FilePortType">
    <operation name="GetFileMetadata">
      <input message="tns:GetFileMetadataRequest"/>
      <output message="tns:GetFileMetadataResponse"/>
    </operation>
  </portType>

  <!-- Endpoint do serviço -->
  <service name="FileService">
    <port binding="tns:FileBinding">
      <soap:address location="http://localhost:8001/soap"/>
    </port>
```

```
    </service>
  </definitions>
```

Principais Tags WSDL:

Tag	Descrição
`<definitions>`	Raiz do documento, define namespaces
`<types>`	Define tipos de dados complexos
`<message>`	Estruturas de entrada/saída
`<portType>`	Interface abstrata com operações
`<binding>`	Protocolo concreto (SOAP/HTTP)
`<service>`	Endpoint real do serviço

Vantagens:

- ☑ **Contrato formal** - WSDL define exatamente a interface
- ☑ **Tipagem forte** - Validação rigorosa de dados
- ☑ **Independência de linguagem** - Qualquer linguagem pode consumir
- ☑ **Transações** - Suporte a operações atômicas
- ☑ **Segurança** - WS-Security padrão

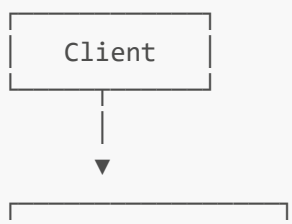
2.3 API Gateway

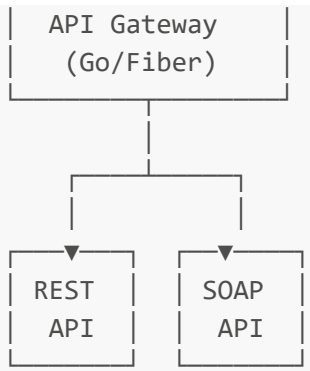
API Gateway é um ponto de entrada único que orquestra requisições entre múltiplos serviços.

Responsabilidades:

1. **Roteamento**: Direciona requisições para serviços apropriados
2. **Agregação**: Combina dados de múltiplas fontes
3. **Transformação**: Converte formatos (JSON ↔ XML)
4. **Autenticação/Autorização**: Controle de acesso centralizado
5. **Rate Limiting**: Controle de taxa de requisições
6. **Cache**: Otimização de performance
7. **Logging**: Auditoria e monitoramento

Arquitetura Gateway:





Benefícios:

- ☑ **Abstração** - Cliente não precisa conhecer serviços internos
- ☑ **Segurança** - Ponto único de controle
- ☑ **Flexibilidade** - Fácil adicionar/remover serviços
- ☑ **Monitoramento** - Centralização de logs
- ☑ **Otimização** - Cache e agregação

2.4 HATEOAS (Hypermedia as the Engine of Application State)

HATEOAS é um princípio REST que torna a API autodescritiva através de links de navegação.

Conceito:

Em vez do cliente precisar conhecer todas as URLs, a API **fornece links dinamicamente** para as ações disponíveis.

Sem HATEOAS:

```
{
  "id": "abc123",
  "filename": "documento.pdf"
}
```

O cliente precisa **saber** que pode acessar:

- `/files/abc123/download` para baixar
- `/files/abc123/metadata` para metadados
- `DELETE /files/abc123` para deletar

Com HATEOAS:

```
{
  "id": "abc123",
  "filename": "documento.pdf",
```

```
    "_links": {
      "self": {
        "href": "http://localhost:3000/files/abc123",
        "method": "GET",
        "rel": "self"
      },
      "download": {
        "href": "http://localhost:3000/files/abc123/download",
        "method": "GET",
        "rel": "download"
      },
      "metadata": {
        "href": "http://localhost:3000/files/abc123/metadata",
        "method": "GET",
        "rel": "metadata"
      },
      "delete": {
        "href": "http://localhost:3000/files/abc123",
        "method": "DELETE",
        "rel": "delete"
      }
    }
  }
```

A API **informa** todas as ações possíveis!

Vantagens:

- ☑ **Descoberta dinâmica** - Cliente explora a API sem documentação prévia
- ☑ **Desacoplamento** - Mudanças de URL não quebram clientes
- ☑ **Auto-documentação** - Links indicam operações disponíveis
- ☑ **Contexto** - Ações disponíveis dependem do estado atual

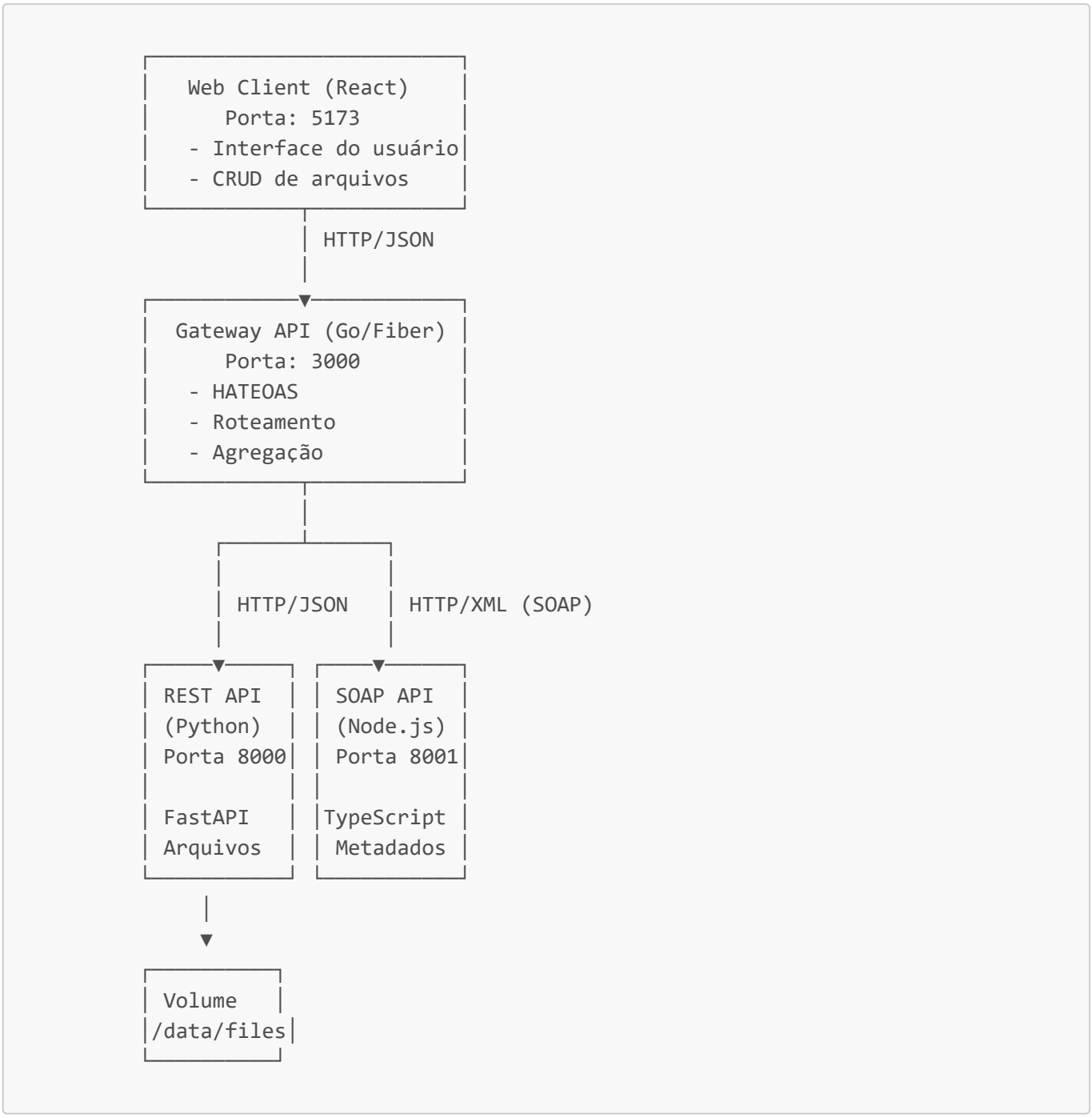
Relações Semânticas (rel):

Relação	Descrição
`self`	O próprio recurso
`collection`	Coleção de recursos
`create`	Criar novo recurso
`update`	Atualizar recurso
`delete`	Deletar recurso
`download`	Download de arquivo
`metadata`	Metadados do recurso

---

3. Arquitetura do Sistema

3.1 Visão Geral



3.2 Tecnologias Utilizadas

Componente	Linguagem	Framework	Porta	Responsabilidade
Web Client	JavaScript	React 18 + Vite	5173	Interface do usuário
Gateway	Go 1.21	Fiber	3000	Orquestração e HATEOAS
REST API	Python 3.11	FastAPI	8000	Gestão de arquivos
SOAP API	TypeScript/Node 18	soap	8001	Metadados

4. Componentes

4.1 Web Client (React)

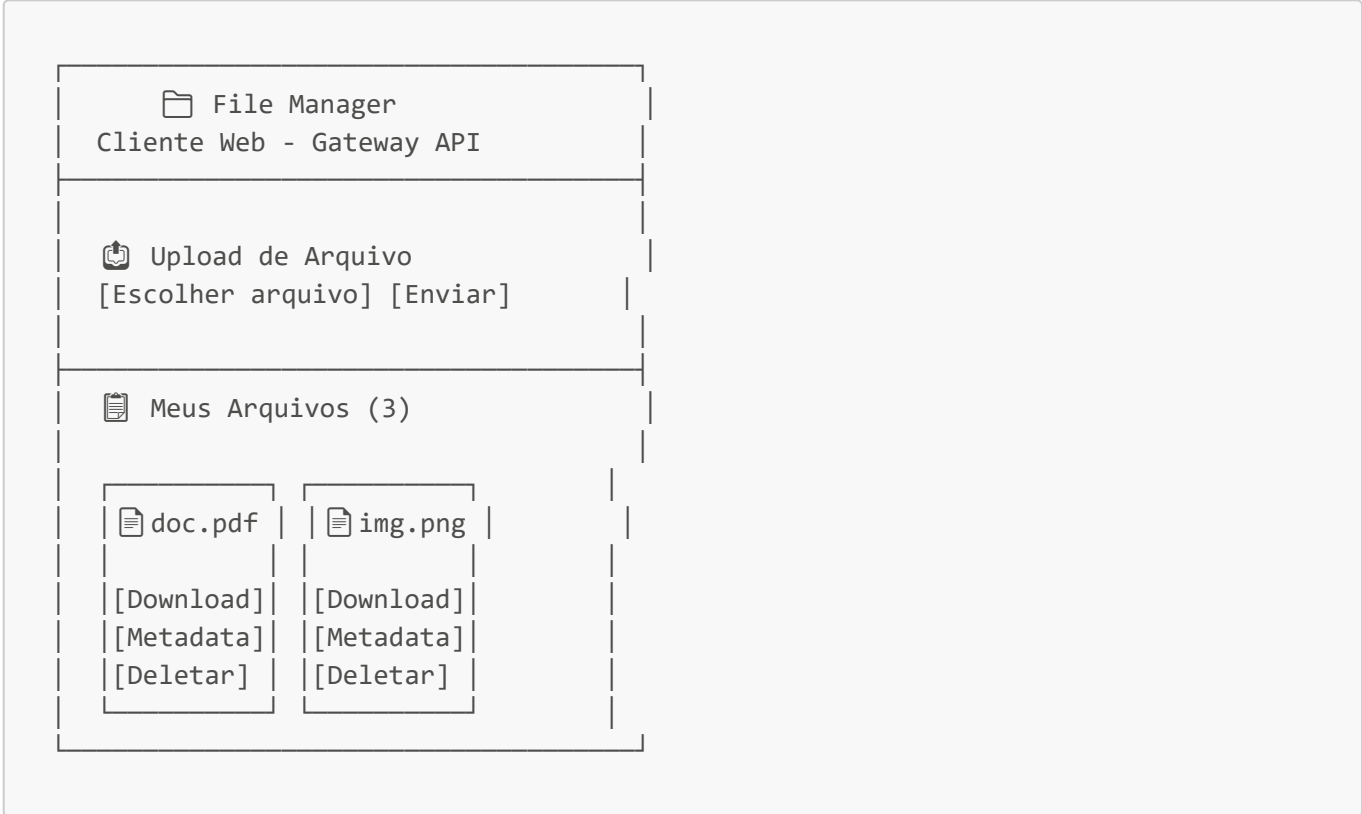
Funcionalidades:

1. **Upload de arquivos** (Create)
2. **Listagem de arquivos** (Read)
3. **Download de arquivos** (Read)
4. **Visualização de metadados** (Read - SOAP)
5. **Exclusão de arquivos** (Delete)

Tecnologias:

- React 18 com Hooks
- Axios para requisições HTTP
- CSS3 com design responsivo
- Vite como build tool

Interface:



4.2 Gateway API (Go/Fiber)

Responsabilidades:

1. **Roteamento**: Direciona requisições
2. **HATEOAS**: Adiciona links de navegação
3. **Agregação**: Combina REST + SOAP
4. **Transformação**: JSON ↔ XML
5. **Documentação**: Swagger UI integrado

Endpoints:

Método	Rota	Descrição	Destino
GET	`/`	Raiz da API com links	Gateway
GET	`/docs`	Swagger UI	Gateway
GET	`/files`	Lista arquivos	REST API
POST	`/files`	Upload arquivo	REST API
GET	`/files/:id`	Info do arquivo	REST API
GET	`/files/:id/download`	Download	REST API
GET	`/files/:id/metadata`	Metadados	SOAP API
DELETE	`/files/:id`	Deleta arquivo	REST API

Exemplo de Resposta HATEOAS:

```
{
  "id": "abc123",
  "filename": "documento.pdf",
  "_links": {
    "self": {
      "href": "http://localhost:3000/files/abc123",
      "method": "GET",
      "rel": "self"
    },
    "download": {
      "href": "http://localhost:3000/files/abc123/download",
      "method": "GET",
      "rel": "download"
    },
    "metadata": {
      "href": "http://localhost:3000/files/abc123/metadata",
      "method": "GET",
      "rel": "metadata"
    },
    "delete": {
      "href": "http://localhost:3000/files/abc123",
      "method": "DELETE",
      "rel": "delete"
    }
  }
}
```

4.3 REST API (Python/FastAPI)

Responsabilidades:

1. Upload de arquivos
2. Download de arquivos
3. Listagem de arquivos
4. Exclusão de arquivos
5. Armazenamento em disco

Estrutura de Armazenamento:

```
/data/files/
├── {uuid}_documento.pdf
├── {uuid}_imagem.png
└── {uuid}_video.mp4
```

Endpoints:

```
@app.post("/files")  # Upload
@app.get("/files")   # Listar
@app.get("/files/{id}")  # Download
@app.delete("/files/{id}")  # Deletar
```

Documentação Automática:

- Swagger UI: `http://localhost:8000/docs`
- ReDoc: `http://localhost:8000/redoc`

4.4 SOAP API (Node.js/TypeScript)

Responsabilidades:

1. Fornecer metadados de arquivos
2. Servir arquivo WSDL
3. Processar requisições SOAP/XML

Operação SOAP:

```
GetFileMetadata(id: int): {
  found: boolean
  name: string
  size: int
  type: string
}
```

WSDL: Disponível em: `http://localhost:8001/soap?wsdl`

Fake Database:

```
const fakeDb = {
  1: { name: "documento.pdf", size: 204800, type: "application/pdf" },
  2: { name: "foto.png", size: 512000, type: "image/png" },
  3: { name: "musica.mp3", size: 3400000, type: "audio/mpeg" }
};
```

---

5. Fluxo de Dados

5.1 Fluxo de Upload

```
1. Usuário seleciona arquivo no Web Client
   │
   ▼
2. Web Client envia POST /files (FormData)
   │ Content-Type: multipart/form-data
   ▼
3. Gateway recebe e valida arquivo
   │
   ▼
4. Gateway reabre arquivo e envia para REST API
   │ POST http://rest-api:8000/files
   ▼
5. REST API salva no volume /data/files
   │ Gera UUID para o arquivo
   ▼
6. REST API retorna {id, filename, path}
   │
   ▼
7. Gateway adiciona links HATEOAS
   │
   ▼
8. Web Client recebe resposta com links
   │
   ▼
9. Interface atualiza lista de arquivos
```

Exemplo de Requisição:

```
// Web Client
const formData = new FormData()
formData.append('file', selectedFile)

const response = await axios.post(
  'http://localhost:3000/files',
  formData
)
```

```
// Resposta com HATEOAS
{
  "id": "abc123",
  "filename": "documento.pdf",
  "message": "File uploaded successfully",
  "_links": {
    "self": {...},
    "download": {...},
    "metadata": {...},
    "delete": {...}
  }
}
```

5.2 Fluxo de Listagem

```
1. Web Client solicita lista
   │ GET /files
   ▼
2. Gateway encaminha para REST API
   │ GET http://rest-api:8000/files
   ▼
3. REST API lista diretório /data/files
   │ Retorna [{id, filename}, ...]
   ▼
4. Gateway adiciona links HATEOAS para cada arquivo
   │
   ▼
5. Web Client renderiza cards com botões
```

Exemplo de Resposta:

```
{
  "files": [
    {
      "id": "abc123",
      "filename": "documento.pdf",
      "_links": {
        "self": {"href": "/files/abc123", "method": "GET"},
        "download": {"href": "/files/abc123/download", "method": "GET"},
        "metadata": {"href": "/files/abc123/metadata", "method": "GET"},
        "delete": {"href": "/files/abc123", "method": "DELETE"}
      }
    }
  ],
  "_links": {
    "self": {"href": "/files", "method": "GET"},
    "upload": {"href": "/files", "method": "POST"}
```

```
  }
}
```

5.3 Fluxo de Download

```
1. Usuário clica em "Download"
   │
   ▼
2. Web Client requisita arquivo
   │ GET /files/{id}/download
   ▼
3. Gateway encaminha para REST API
   │ GET http://rest-api:8000/files/{id}
   ▼
4. REST API localiza arquivo no volume
   │ Lê arquivo {uuid}_{filename}
   ▼
5. REST API retorna arquivo binário
   │ Content-Disposition: attachment
   ▼
6. Gateway repassa para cliente
   │
   ▼
7. Navegador inicia download
```

5.4 Fluxo de Metadados (SOAP)

```
1. Usuário clica em "Metadados"
   │
   ▼
2. Web Client requisita metadados
   │ GET /files/{id}/metadata
   ▼
3. Gateway monta envelope SOAP
   │
   ▼
4. Gateway envia requisição SOAP para SOAP API
   │ POST http://soap-api:8001/soap
   │ Content-Type: text/xml
   │
   │ <?xml version="1.0"?>
   │ <soap:Envelope>
   │   <soap:Body>
   │     <GetFileMetadata>
   │       <id>1</id>
   │     </GetFileMetadata>
   │   </soap:Body>
```

```
   │ </soap:Envelope>
   │
   ▼
5. SOAP API processa XML
   │ Valida contra WSDL
   │ Busca dados no fakeDb
   ▼
6. SOAP API retorna XML
   │
   │ <soap:Envelope>
   │   <soap:Body>
   │     <GetFileMetadataResponse>
   │       <found>true</found>
   │       <name>documento.pdf</name>
   │       <size>204800</size>
   │       <type>application/pdf</type>
   │     </GetFileMetadataResponse>
   │   </soap:Body>
   │ </soap:Envelope>
   │
   ▼
7. Gateway converte XML para JSON
   │ Adiciona links HATEOAS
   ▼
8. Web Client exibe modal com metadados
```

Exemplo de Resposta do Gateway:

```
{
  "metadata": {
    "id": "1",
    "found": true,
    "name": "documento.pdf",
    "size": 204800,
    "type": "application/pdf",
    "_links": {
      "self": {"href": "/files/1/metadata"},
      "file": {"href": "/files/1"},
      "download": {"href": "/files/1/download"}
    }
  },
  "soap_raw_xml": "<?xml version='1.0'?>..."
}
```

5.5 Fluxo de Exclusão

```
1. Usuário clica em "Deletar"
   │ Confirma ação
```

```
   ▼
2. Web Client envia DELETE
   │ DELETE /files/{id}
   ▼
3. Gateway encaminha para REST API
   │ DELETE http://rest-api:8000/files/{id}
   ▼
4. REST API remove arquivo do disco
   │ os.remove({uuid}_{filename})
   ▼
5. REST API confirma exclusão
   │
   ▼
6. Gateway adiciona link para coleção
   │ _links: { files: {...} }
   ▼
7. Web Client atualiza lista
```

---

6. Implementação Técnica

6.1 Docker Compose

Todos os componentes são orquestrados via Docker Compose:

```yaml
version: '3.8'

services:
  rest-api:
    build: ./rest-api
    ports: ["8000:8000"]
    volumes:
      - files-data:/data/files
    networks:
      - fileserver-network

  soap-api:
    build: ./soap-api
    ports: ["8001:8001"]
    networks:
      - fileserver-network

  gateway-api:
    build: ./gateway-api
    ports: ["3000:9000"]
    depends_on:
      - rest-api
      - soap-api
    environment:
      - REST_URL=http://rest-api:8000
      - SOAP_URL=http://soap-api:8001/soap
```

```
    networks:
      - fileserver-network

  web-client:
    build: ./web-client
    ports: ["5173:80"]
    depends_on:
      - gateway-api
    networks:
      - fileserver-network

volumes:
  files-data:

networks:
  fileserver-network:
```

6.2 Comunicação entre Serviços

Gateway → REST API (HTTP/JSON)

```
// Gateway envia para REST API
resp, err := client.R().Get(restURL + "/files")

var restFiles []map[string]interface{}
json.Unmarshal(resp.Body(), &restFiles)
```

Gateway → SOAP API (HTTP/XML)

```
// Gateway monta envelope SOAP
soapEnvelope := fmt.Sprintf(`
  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
      <GetFileMetadata xmlns="urn:FileService">
        <id>%s</id>
      </GetFileMetadata>
    </soap:Body>
  </soap:Envelope>
`, id)

// Envia requisição SOAP
resp, err := client.R().
  SetHeader("Content-Type", "text/xml").
  SetBody(soapEnvelope).
  Post(soapURL)
```

6.3 Implementação HATEOAS

```
// Função para criar links HATEOAS
func createFileLinks(id string) map[string]Link {
  return map[string]Link{
    "self": {
      Href:   baseURL + "/files/" + id,
      Method: "GET",
      Rel:    "self",
    },
    "download": {
      Href:   baseURL + "/files/" + id + "/download",
      Method: "GET",
      Rel:    "download",
    },
    "metadata": {
      Href:   baseURL + "/files/" + id + "/metadata",
      Method: "GET",
      Rel:    "metadata",
    },
    "delete": {
      Href:   baseURL + "/files/" + id,
      Method: "DELETE",
      Rel:    "delete",
    },
  }
}

// Adiciona links na resposta
return c.JSON(FileResponse{
  ID:       id,
  Filename: filename,
  Links:    createFileLinks(id),
})
```

---

7. Demonstração Prática

7.1 Executando o Sistema

```
# Clone o repositório
git clone https://github.com/pedrohcdsouza/fileserver-with-gateway
cd fileserver-with-gateway

# Inicie todos os serviços
docker-compose up --build

# Aguarde inicialização (30-60 segundos)
```

7.2 Acessando as Interfaces

Componente	URL	Descrição
Web Client	http://localhost:5173	Interface web
Gateway	http://localhost:3000	API Gateway
Gateway Swagger	http://localhost:3000/docs	Documentação
REST API	http://localhost:8000	REST API
REST Swagger	http://localhost:8000/docs	Documentação
SOAP API	http://localhost:8001/soap	SOAP Endpoint
WSDL	http://localhost:8001/soap?wsdl	Contrato SOAP

7.3 Testando com cURL

Listar arquivos

```
curl http://localhost:3000/files
```

Upload de arquivo

```
curl -X POST http://localhost:3000/files \
  -F "file=@documento.pdf"
```

Download

```
curl -O -J http://localhost:3000/files/abc123/download
```

Metadados (SOAP)

```
curl http://localhost:3000/files/1/metadata
```

Deletar

```
curl -X DELETE http://localhost:3000/files/abc123
```

7.4 Testando SOAP Diretamente

```
curl -X POST http://localhost:8001/soap \
  -H "Content-Type: text/xml" \
  -d '<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetFileMetadata xmlns="urn:FileService">
      <id>1</id>
    </GetFileMetadata>
  </soap:Body>
</soap:Envelope>'
```

---

8. Conclusão

8.1 Objetivos Alcançados

- ☑ **API Gateway desenvolvido** - Go/Fiber com roteamento e agregação
- ☑ **HATEOAS implementado** - Links dinâmicos em todas as respostas
- ☑ **Documentação completa** - Swagger UI, OpenAPI, READMEs
- ☑ **REST API funcional** - Python/FastAPI para arquivos
- ☑ **SOAP API funcional** - Node.js/TypeScript para metadados
- ☑ **WSDL documentado** - Explicação detalhada das tags
- ☑ **Cliente Web** - Interface React moderna e responsiva
- ☑ **Integração completa** - Todos os componentes se comunicam
- ☑ **Docker** - Infraestrutura containerizada

8.2 Comparação REST vs SOAP

Aspecto	REST	SOAP
Formato	JSON, XML, etc.	XML obrigatório
Protocolo	HTTP	HTTP, SMTP, TCP, etc.
Contrato	Opcional (OpenAPI)	Obrigatório (WSDL)
Simplicidade	☑ Muito simples	⚠ Complexo
Tipagem	Fraca	☑ Forte
Performance	☑ Rápido	⚠ Mais lento (XML)
Uso	APIs públicas, mobile	Sistemas corporativos
Estado	Stateless	Stateful possível

8.3 Benefícios da Arquitetura

Separação de Responsabilidades

- **Gateway**: Orquestração, segurança, HATEOAS
- **REST API**: Gestão de arquivos
- **SOAP API**: Metadados estruturados
- **Web Client**: Interface do usuário

Escalabilidade

- Cada serviço pode escalar independentemente
- Load balancer pode distribuir carga
- Stateless facilita replicação

Manutenibilidade

- Código organizado por responsabilidade
- Fácil adicionar novos serviços
- Testes isolados por componente

Flexibilidade

- Fácil trocar implementações
- Múltiplos clientes (Web, Mobile, Desktop)
- Protocolos diferentes coexistem

8.4 Melhorias Futuras

1. **Autenticação** - OAuth2, JWT
2. **Banco de dados** - PostgreSQL para metadados
3. **Cache** - Redis para otimização
4. **Message Queue** - RabbitMQ para processamento assíncrono
5. **Monitoramento** - Prometheus + Grafana
6. **Logging** - ELK Stack
7. **CI/CD** - GitHub Actions
8. **Testes** - Unitários, integração, e2e

8.5 Lições Aprendidas

REST

- ☑ Simples e eficiente para APIs públicas
- ☑ JSON é leve e fácil de trabalhar
- ⚠ Falta de contrato formal pode gerar problemas

SOAP

- ☑ WSDL garante contrato claro

- ☑ Tipagem forte previne erros
- ⚠ XML é verboso e mais lento
- ⚠ Curva de aprendizado maior

API Gateway

- ☑ Centralização simplifica arquitetura
- ☑ HATEOAS melhora muito a experiência
- ⚠ Ponto único de falha (necessita HA)

8.6 Referências

- REST API Design
- SOAP Web Services
- HATEOAS
- API Gateway Pattern
- OpenAPI Specification
- WSDL Specification

8.7 Contato

Repositório: github.com/pedrohcdsouza/fileserver-with-gateway

---

Apêndices

A. Estrutura de Diretórios

```
fileserver-with-gateway/
├── docker-compose.yml
├── README.md
├── DOCUMENTATION.md           # Este documento
├── web-client/
│   ├── src/
│   │   ├── App.jsx
│   │   ├── main.jsx
│   │   └── index.css
│   ├── Dockerfile
│   ├── package.json
│   └── README.md
├── gateway-api/
│   ├── app/
│   │   └── main.go
│   ├── docs/
│   │   └── swagger.html
│   ├── Dockerfile
│   ├── openapi.yaml
│   └── README.md
├── rest-api/
```

```
│   ├── app/
│   │   └── main.py
│   ├── Dockerfile
│   ├── requirements.txt
│   └── README.md
└── soap-api/
    ├── app/
    │   └── main.ts
    ├── Dockerfile
    ├── package.json
    ├── tsconfig.json
    └── README.md
```

B. Portas Utilizadas

Porta	Serviço	Protocolo
5173	Web Client	HTTP
3000	Gateway API	HTTP/JSON
8000	REST API	HTTP/JSON
8001	SOAP API	HTTP/XML

C. Comandos Úteis

```
# Iniciar serviços
docker-compose up --build

# Parar serviços
docker-compose down

# Ver logs
docker-compose logs -f gateway-api

# Rebuild específico
docker-compose up --build gateway-api

# Limpar volumes
docker-compose down -v
```

Fim do Documento