SmartTransfer

O **SmartTransfer** consiste em uma aplicação simples de agendamento de transferências bancárias.

Funcionalidades da aplicação

- Autenticação de usuários;
- Proteção de rotas por perfil de acesso (RBAC);
- CORS;
- Bearer Token como modelo de autenticação;
- Auditoria de modificação com `AuditorAware` do Spring Data junto com `UserDetails` do Spring Security;
- Specifications para filtros dinâmicos avançados, utilizando `MetaModels` em tempo de compilação para checagem de tipos nas consultas;
- Segmentação de responsabilidade (`Controllers`, `Services`, `Repositories`);
- JPA como ORM do projeto;
- Seed de banco de dados através do JPA;
- Mapeamento entre DTOs e Entidades com MapStruct, evitando repetição de código;
- Configuração do projeto via Gradle, utilizando Groovy;
- Persistência dos dados em banco de memória H2;
- Documentação via Swagger.

Tecnologias utilizadas

- Java 11
- Gradle
- Spring Boot
- Spring Security
- JSON Web Token (JWT)
- Lombok
- MapStruct
- Hibernate
- H2 Database
- JPA
- MetaModels

Como executar o projeto

Como utilizei o IntelliJ para implementar a solução, o passo a passo abaixo considera essa IDE, mas o processo é semelhante em outras:

```
# Clone o repositório
git clone https://github.com/NaskIII/SmartTransfer.git t
```

1. Abra o projeto no IntelliJ e aceite a elevação solicitada pelo Windows Defender (se aplicável).
2. Vá até o menu de execução no topo, clique nos três pontos ao lado do botão de "Debug" > **Edit Configurations**.
3. Na nova janela, clique no botão `+` e selecione **Gradle**.
4. Nomeie a configuração como `SmartTransfer`.
5. Em **Run**, insira `bootRun`.
6. Em **Gradle Project**, selecione `SmartTransfer`.
7. Em **Environment Variables**, adicione:
   `spring.profiles.active=dev`

Após isso, basta executar o projeto.

Recursos adicionais

- Console do H2: http://localhost:8081/h2-console
  - JDBC URL: `jdbc:h2:mem:dev-database`
  - Usuário: `sa`
  - Senha: (em branco)
- Swagger UI: http://localhost:8081/swagger-ui/index.html

Decisões técnicas

- Implementei usuários e auditoria para garantir rastreabilidade em operações sensíveis, como transferências monetárias. Essa abordagem permite auditorias em casos de falhas ou fraudes. Para este projeto, campos como `createdBy`, `createdDate`, `updatedBy` e `updatedDate` são suficientes, mas em um sistema real seria recomendável registrar cada requisição com detalhes como IP, headers e payload no ElasticSearch.
- Optei por utilizar `BigDecimal` para tratar valores monetários devido à sua precisão e compatibilidade com operações financeiras, além da boa integração com a API.
- A utilização de `Specifications` com `MetaModels` permite construir filtros dinâmicos robustos, mesmo com parâmetros opcionais. Essa abordagem é nativamente suportada pelo JPA e facilita a criação de buscas flexíveis e seguras.
- Com o uso do `MapStruct`, é possível evitar a duplicação de lógica entre DTOs e entidades. Essa decisão também abre espaço para o uso de classes genéricas, como `BaseService` e `BaseRepository`, que facilitam a manutenção e escalabilidade do projeto ao centralizar lógica de CRUD em componentes reutilizáveis.
- Os cálculos de taxas foram implementados em uma classe isolada, considerando a simplicidade do escopo. Em um sistema corporativo, as taxas deveriam ser armazenadas em uma tabela de banco

de dados com suporte a CRUD, possibilitando alterações sem necessidade de recompilar o sistema.

- Implementei controle de acesso baseado em papéis (RBAC) por ser uma abordagem simples e eficiente. Neste projeto, cada usuário possui apenas um papel. Em um ambiente real, seria interessante permitir múltiplos papéis por usuário para maior flexibilidade.

- Devido ao prazo apertado, testes unitários, funcionais e de integração não foram implementados. No entanto, toda a estrutura da aplicação foi planejada para facilitar a inclusão desses testes futuramente. A escolha do Gradle como ferramenta de build foi feita justamente por sua flexibilidade e poder de configuração, o que permite integrar facilmente bibliotecas de testes, criar múltiplos ambientes de execução e definir tarefas automatizadas de build e verificação.

- Em relação as chaves primárias das entidades, utilizei String e Long devido as limitações do banco de dados H2. Originalmente eles foram implementados utilizando UUID, porém quando esse tipo de dado é salvo no banco de dados, ele perdia a formatação e/ou ele prechia 230 caractereis com 0. Lendo alguns artigos verifiquei que o H2 não tem um suporte muito bom para com esse tipo de dado. Sendo assim, optei por utilizar Long e String.

- Particularmente, sou contra o uso de números sequenciais como chave primária por uma questão de segurança. Imagine, por exemplo, um endpoint que lista usuários por ID, resultando em uma URL como: `http://sistema.com/users/item/{id}`. Se o `{id}` for um número, como `5`, é possível inferir a existência dos registros `1`, `2`, `3` e `4`. Mesmo que a aplicação implemente uma política robusta de autorização e bloqueie acessos indevidos, o simples fato de a URL expor a previsibilidade de IDs já representa uma vulnerabilidade em potencial. Utilizando UUIDs como chave primária, esse problema é eliminado, pois o identificador será algo como `550e8400-e29b-41d4-a716-446655440000`, tornando impossível deduzir a existência de outros registros com base no padrão da URL.

Considerações Pessoais

- Pessoalmente, prefiro escrever o código em inglês, pois isso se alinha melhor ao vocabulário técnico da programação e facilita a colaboração com pessoas de diferentes nacionalidades.