UNIVERDADE FEDERAL DO PARANÁ

SETOR DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
UFPR

Disciplina: **Desenvolvimento para Ambientes Corporativos**

Nome: Cristhian Rodrigues Kindermann GRR: 20214914

Relatório de Decisões de Design da API REST

1. Introdução

Este relatório descreve as principais decisões de arquitetura e design tomadas durante o desenvolvimento da API REST para o sistema de biblioteca. O objetivo central foi construir uma API que fosse robusta, escalável, de fácil manutenção e que proporcionasse uma excelente experiência de uso para os desenvolvedores clientes, aderindo estritamente aos princípios da arquitetura REST e alcançando o Nível 3 do Modelo de Maturidade de Richardson (HATEOAS).

2. Decisões de Arquitetura e Design

2.1. Escolha da Plataforma: Java e Spring Boot

A decisão de utilizar Java com o ecossistema Spring Boot foi estratégica. Esta plataforma oferece um ambiente maduro e de alto desempenho para o desenvolvimento de aplicações web. As principais vantagens que motivaram essa escolha foram:

- **Produtividade Acelerada:** O Spring Boot, com sua abordagem de "convenção sobre configuração", elimina grande parte da configuração manual, permitindo focar na lógica de negócio.
- **Ecossistema Abrangente:** O projeto utiliza módulos como `Spring Web` para a criação dos endpoints REST, `Spring Data JPA` para simplificar drasticamente a camada de persistência de dados, e `Spring HATEOAS` para a implementação de links dinâmicos.
- **Injeção de Dependência:** O mecanismo de Inversão de Controle (IoC) e Injeção de Dependência do Spring promove um baixo acoplamento entre os componentes, facilitando os testes unitários e a manutenção do código.

2.2. Design de Recursos e URIs

A modelagem dos recursos seguiu as melhores práticas RESTful, tratando as entidades de negócio como recursos identificáveis por URIs.

- **Uso de Substantivos no Plural:** As coleções de recursos são nomeadas com substantivos no plural (ex: `/livros`, `/autores`), tornando a API intuitiva. A URI representa "uma coleção de livros".
- **Identificação via Path Variable:** Para acessar um recurso específico, utilizamos seu identificador único na URI (ex: `/livros/{id}`). Este padrão é universalmente compreendido e fácil de implementar em qualquer cliente HTTP.
- **Hierarquia Lógica:** Embora não tenhamos implementado neste escopo, a

estrutura permite a criação de URIs hierárquicas, como `/autores/{autorId}/livros`, para representar relacionamentos entre recursos.

2.3. Estratégia de Respostas e Códigos de Status HTTP

A API utiliza o protocolo HTTP de forma semântica, aproveitando seus verbos e códigos de status para comunicar o resultado das operações de forma clara e padronizada.

- **Verbos HTTP:** `GET` para consulta (idempotente e seguro), `POST` para criação, `PUT` para atualização completa (idempotente) e `DELETE` para remoção (idempotente).
- **Códigos de Status de Sucesso:**
  - `200 OK`: Para respostas bem-sucedidas que retornam conteúdo (ex: `GET`, `PUT`).
  - `201 Created`: Para a criação de um novo recurso (`POST`), acompanhado do cabeçalho `Location` que aponta para a URI do novo recurso.
  - `204 No Content`: Para operações bem-sucedidas que não precisam retornar conteúdo (ex: `DELETE`).
- **Códigos de Status de Erro:** O uso de códigos como `400 Bad Request`, `404 Not Found` e `500 Internal Server Error` fornece feedback imediato e específico sobre a natureza da falha.

2.4. Tratamento de Erros Centralizado

Para garantir consistência e evitar código repetitivo, foi implementado um mecanismo de tratamento de exceções global utilizando a anotação `@ControllerAdvice`.

- **Exceções Customizadas:** Foi criada a exceção `ResourceNotFoundException`, que é lançada quando um recurso específico não é encontrado.
- **Handler Global:** Uma classe `GlobalExceptionHandler` intercepta essas exceções e as transforma em uma resposta JSON padronizada, contendo informações úteis como timestamp, mensagem de erro e o caminho da requisição. Isso desacopla a lógica de negócio do formato da resposta de erro.

2.5. Implementação de HATEOAS (Nível 3 de Richardson)

A decisão mais significativa para a maturidade da API foi a implementação de HATEOAS (Hypermedia as the Engine of Application State).

- **Auto-descoberta:** Em vez de o cliente ter que codificar "hard-code" as URIs para as ações, a própria API informa as próximas ações possíveis através de links na resposta.
- **Redução de Acoplamento:** Com HATEOAS, a URI para listar todos os livros pode mudar de `/livros` para `/api/v2/livros`, e um cliente bem projetado que segue o link `todos-livros` continuará a funcionar sem alterações.
- **Links Contextuais:** A API pode fornecer links que dependem do estado do recurso. Por exemplo, um livro disponível teria um link `emprestar`, enquanto um livro emprestado teria um link `devolver`.
- **Spring HATEOAS:** A implementação foi facilitada pela biblioteca `spring-boot-starter-hateoas`, que se integra ao Spring MVC e permite a construção de links de forma segura e robusta.