Versão do Spring Boot: 3.5.7
Versão do Java: 21

Dependências utilizadas

- **spring-boot-starter-data-jpa:** acesso ao banco usando JPA/Hibernate.
- **spring-boot-starter-validation:** permite validar dados com anotações (@NotNull, @Size etc).
- **spring-boot-starter-web:** criação de APIs REST com Spring MVC.
- **spring-boot-devtools:** facilita o desenvolvimento com recarregamento automático.
- **h2:** banco de dados em memória.

Estrutura do Projeto

1. Pacote *persistence*

Entities: contém as classes *Autor* e *Publicacao*, que representam as tabelas do banco.

- Um **Autor** pode estar em várias publicações.
- Uma **Publicação** sempre tem somente um autor.

Repositories:

- *AutorRepository* e *PublicacaoRepository*.
- Ao estenderem *JpaRepository*, já ganham métodos automáticos como:
  `save`, `findById`, `findAll`, `deleteById`, etc.

---

2. Pacote *controller* e *service*

- O **Controller** recebe as requisições feitas pelo front via *fetch*.

- Ele envia os dados para o **Service**, que contém as regras de negócio.
- O **Service** chama o **Repository**, que busca ou salva informações no banco.

Cada entidade tem seu próprio conjunto:

- AutorController / AutorService
- PublicacaoController / PublicacaoService
  Com métodos de CRUD: listar, salvar, atualizar, excluir e buscar.

---

3. Pacote *DTO*

Contém *AutorDTO* e *PublicacaoDTO*.

- Aqui ficam as validações como:
  `@NotBlank`, `@NotNull`, `@Size`
- O DTO é separado da entidade para:
  - Não misturar validação com estrutura do banco.
  - Evitar expor diretamente a entidade ao front.
- No Service, os dados do DTO são convertidos para a entidade antes de salvar.

---

4. Pacote *exceptions*

Responsável por padronizar o envio de erros vindos das validações dos DTOs.

- O *GlobalExceptionHandler* captura erros de validação.
- Ele organiza os erros por campo e envia ao front algo como:
  - qual campo tem erro
  - qual é a mensagem de erro
- Assim, o front consegue exibir a mensagem correta no campo correto.

Configuração do Banco de Dados (application.properties)

No projeto **blog**, o banco está configurado em memória:

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
```

Significa:

- O banco roda somente na memória.
- Ao iniciar a aplicação, o banco é recriado do zero.
- Ao parar a aplicação, o banco é apagado.
- O arquivo **data.sql** é executado automaticamente e insere dados iniciais.

No projeto **dados-pessoais**, a configuração era assim:

```
spring.datasource.url=jdbc:h2:file:C:/senac/dados-pessoais-bd
```

Significa:

- Banco fica salvo em um arquivo.
- Os dados continuam existindo mesmo após reiniciar a aplicação.
- Usado quando o projeto precisa armazenar informações permanentemente.

Também existe o arquivo **data.sql**, responsável por inserir dados iniciais quando a aplicação sobe.

Front-end

Foi utilizado **Bootstrap** em praticamente toda a interface.
Quase não há CSS próprio.
Qualquer componente visual pode ser encontrado na documentação oficial do Bootstrap.

Requisitos extras (apenas análise, não implementados)

- Funcionalidade de busca textual

Para essa funcionalidade seria necessário adicionar um campo no *index.html* onde fica a listagem, um campo de texto simples.

Quando o usuário digitar algo e clicar no botão de buscar (uma lupa), esse valor seria enviado para o mesmo endpoint que já lista as publicações, só que agora enviando também um parâmetro de URL com o texto da busca.

No back-end seria criado um objeto que representa os filtros, e esse objeto teria um campo específico para a busca textual.

Se esse campo estiver preenchido, o repository usaria o próprio JPA com um método do tipo *findByTituloContaining* ou algo parecido, retornando apenas registros onde o título salvo no banco seja igual ou parecido com o texto digitado.

- Filtro para mostrar somente publicações com datas no futuro (não publicadas)

Aqui a ideia é exatamente o mesmo esquema do filtro de texto.

Na tela poderia existir um checkbox dizendo se o usuário quer ver apenas as publicações futuras.

Esse checkbox enviaria um valor verdadeiro ou falso para o back-end.

O mesmo objeto de filtros teria esse boolean.

Se o boolean vier como verdadeiro, o repository faria uma consulta filtrando somente registros onde a data de publicação seja maior que a data atual (dataPublicacao > hoje).

- Filtro para mostrar somente publicações que pertencem ao mesmo autor

Segue o mesmo esquema também.

Na tela teria um campo igual ao do cadastro de publicação, carregando a lista de autores para o usuário escolher.

Quando o usuário selecionar um autor, o id desse autor seria enviado dentro do objeto de filtros.

No back-end, com esse id, o repository faria uma busca usando algo como *findByAutorId*, retornando apenas as publicações que pertencem ao autor selecionado.

- Segurança: somente o autor logado pode ver/alterar/excluir suas próprias publicações

Aqui seria igual ao que já foi feito no projeto *dados-pessoais*: implementar toda a parte de login e autenticação usando token JWT.

Com o usuário logado identificado pelo token, o sistema saberia qual é o autor atual.

A partir disso, na hora de listar as publicações para edição ou exclusão, o repository buscaria somente as que pertencem ao autor logado, usando o mesmo tipo de método *findByAutorId*.

Assim, cada autor só teria acesso às suas próprias publicações.