Respostas: Exercícios Práticos - Projeto Livraria com Spring Boot e JPA

---

1. Entendimento da Estrutura do Projeto

Perguntas e Respostas:

1. **Entidade (Livro.java):**
    - **Qual é o propósito principal da classe Livro?** A classe Livro é uma **entidade JPA** que representa a tabela livro no banco de dados. Seu propósito principal é mapear os dados da tabela para um objeto Java e vice-versa, permitindo que o Hibernate (o ORM usado pelo JPA) gerencie a persistência desses dados.
    - **Para que servem as anotações @Entity, @Table e @Id?**
        - @Entity: Marca a classe como uma entidade JPA, indicando que ela será mapeada para uma tabela de banco de dados.
        - @Table(name = "livro"): Especifica o nome da tabela no banco de dados com a qual está entidade está associada. Se omitida, o JPA usaria o nome da classe (Livro) como o nome da tabela.
        - @Id: Marca o campo como a **chave primária** da entidade, fundamental para a identificação única de cada registro no banco de dados.
    - **Como o Lombok (@Data, @NoArgsConstructor, @AllArgsConstructor) contribui para esta classe?** O **Lombok** é uma biblioteca que reduz o "boilerplate code" (código repetitivo).
        - @Data: Gera automaticamente métodos **getters** e **setters** para todos os campos, além de toString(), equals() e hashCode().
        - @NoArgsConstructor: Gera um **construtor sem argumentos**, que é exigido pelo JPA.
        - @AllArgsConstructor: Gera um **construtor com todos os argumentos**, o que é útil para instanciar a entidade com todos os valores.
2. **Repositório (LivroRepository.java):**
    - **Qual o objetivo de estender JpaRepository<Livro, Long>?** Estender JpaRepository permite que a interface LivroRepository

herde um conjunto de **métodos CRUD (Create, Read, Update, Delete) predefinidos** e poderosos, como save(), findById(), findAll(), deleteById(), entre outros. Livro é o tipo da entidade e Long é o tipo da chave primária (id).

- **Que tipos de operações este repositório é capaz de realizar sem que você escreva nenhum código específico para elas?** Sem escrever código, ele pode:
  - Salvar/Atualizar uma entidade (save).
  - Buscar uma entidade por ID (findById).
  - Buscar todas as entidades (findAll).
  - Deletar uma entidade por ID (deleteById).
  - Contar o número de entidades (count).
  - Verificar se uma entidade existe por ID (existsById).
- **Se você precisasse buscar um livro pelo título, como você declararia um método para isso nesta interface?**

  Você declararia um método com uma **query method** do Spring Data JPA, como:

  Java

  List<Livro> findByTitulo(String titulo);

  // Ou para busca ignorando maiúsculas/minúsculas e contendo o texto:

  List<Livro> findByTituloContainingIgnoreCase(String titulo);

3. **Serviço (LivroService.java):**
   - **Por que existe uma camada de Serviço separada do Controller e do Repository? Qual é a sua responsabilidade principal?** A camada de Serviço existe para encapsular a **lógica de negócios** da aplicação. Sua responsabilidade principal é coordenar as operações (que podem envolver múltiplos repositórios ou regras complexas), validar dados, e aplicar as regras de negócio antes de interagir com a camada de persistência. Ela atua como um intermediário entre o Controller e o Repository, garantindo que o Controller não precise conhecer os detalhes da persistência e que o Repository não contenha lógica de negócio.

- **O que significa a injeção de dependência via construtor (@Autowired) nesta classe?** Significa que o Spring Boot é responsável por criar e gerenciar as instâncias das classes. Ao usar @Autowired no construtor de LivroService, o Spring automaticamente encontra uma instância de LivroRepository (que é um @Repository e, portanto, um componente Spring) e a injeta como um argumento para o construtor, tornando-a disponível para uso dentro do serviço. Isso promove a inversão de controle (IoC).
- **Explique a lógica do método atualizarLivro. O que acontece se um livro com o id especificado não for encontrado?** A lógica de atualizarLivro primeiro tenta encontrar o livro pelo id usando livroRepository.findById(id).
    - Se o livro for encontrado (.map(...) é executado), ele atualiza cada campo do livroExistente com os dados do livroAtualizado e então salva (save()) a entidade atualizada de volta no banco de dados.
    - Se o livro **não for encontrado** (.orElseThrow(...) é executado), ele lança uma RuntimeException com a mensagem "Livro não encontrado com o ID: " + id.

4. **Controlador (LivroController.java):**
    - **Para que serve a anotação @RestController?** @RestController é uma anotação de conveniência que combina @Controller e @ResponseBody. Ela indica que a classe é um controlador que lida com requisições HTTP e que os métodos da classe devem retornar dados diretamente como corpo da resposta (geralmente JSON ou XML), em vez de nomes de views.
    - **Qual a função de @RequestMapping("/api/livros")?** Define o **caminho base (base path)** para todas as requisições que esta classe de controlador irá manipular. Qualquer endpoint definido dentro de LivroController terá "/api/livros" como prefixo. Por exemplo, @PostMapping sem um caminho adicional resultará em POST /api/livros.
    - **Diferencie o uso de @PostMapping, @GetMapping, @PutMapping e @DeleteMapping em relação aos métodos HTTP.** São anotações de atalho para @RequestMapping combinadas com um método HTTP específico:

        - @PostMapping: Mapeia requisições **HTTP POST** (usadas para **criar** novos recursos).
        - @GetMapping: Mapeia requisições **HTTP GET** (usadas para **recuperar/ler** recursos).
        - @PutMapping: Mapeia requisições **HTTP PUT** (usadas para **atualizar** completamente recursos existentes).
        - @DeleteMapping: Mapeia requisições **HTTP DELETE** (usadas para **remover/deletar** recursos).
    - **Qual o papel de ResponseEntity nos métodos do controlador? Cite exemplos de códigos de status HTTP retornados.**
      ResponseEntity é uma classe que representa a **resposta HTTP completa**: corpo da resposta, cabeçalhos e código de status HTTP. Seu papel é dar **controle total** sobre a resposta que será enviada ao cliente.
        - **Exemplos de códigos de status HTTP retornados:**
            - HttpStatus.CREATED (201): Para POST de sucesso (recurso criado).
            - HttpStatus.OK (200): Para GET e PUT de sucesso (requisição bem-sucedida).
            - HttpStatus.NOT_FOUND (404): Para GET, PUT ou DELETE quando o recurso solicitado não é encontrado.
            - HttpStatus.NO_CONTENT (204): Para DELETE de sucesso (requisição bem-sucedida, mas sem conteúdo para retornar).
            - HttpStatus.INTERNAL_SERVER_ERROR (500): Para erros inesperados no servidor.

---

2. Análise e Uso de Métodos

Perguntas e Respostas:

1. **Fluxo de uma Requisição POST:**
    - **Descreva passo a passo o que acontece quando uma requisição POST é feita para /api/livros para cadastrar um novo livro. Inclua todas as camadas envolvidas e os métodos chamados.**

1. **Cliente envia requisição:** Uma requisição HTTP POST é enviada para http://localhost:8080/api/livros com o corpo contendo os dados JSON do novo livro.
2. **Controller (LivroController) recebe a requisição:** O método cadastrarLivro no LivroController é invocado porque está mapeado para POST /api/livros. O @RequestBody converte o JSON do corpo da requisição para um objeto Livro Java.
3. **Controller chama o Serviço:** livroService.salvarLivro(livro) é chamado, passando o objeto Livro recebido.
4. **Serviço (LivroService) executa a lógica de negócio:** O método salvarLivro é acionado. Nesta implementação simples, não há lógica de negócio complexa, mas poderia haver validações aqui.
5. **Serviço chama o Repositório:** livroRepository.save(livro) é chamado.
6. **Repositório (LivroRepository) interage com o JPA/Banco de Dados:** O JpaRepository (com o Hibernate por baixo dos panos) traduz a operação save() para uma instrução SQL INSERT e a executa no banco de dados. O id do livro é gerado pelo banco de dados (graças a @GeneratedValue) e retornado ao objeto Livro.
7. **Repositório retorna o livro salvo para o Serviço:** O save() retorna a instância do Livro com o ID gerado e outros campos.
8. **Serviço retorna o livro salvo para o Controller:** O método salvarLivro do serviço retorna o Livro completo para o controlador.
9. **Controller envia a resposta HTTP:** O LivroController cria um ResponseEntity contendo o objeto Livro salvo (agora com ID) e o código de status HttpStatus.CREATED (201), que é enviado de volta ao cliente como JSON.

2. **Método buscarLivroPorId:**
    - **No LivroService, o método buscarLivroPorId retorna um Optional<Livro>. Por que usar Optional?** Optional é um contêiner que pode ou não conter um valor não nulo. Ele é usado para **evitar NullPointerExceptions** e para tornar explícita a intenção de que um

método pode não retornar um valor. Em vez de retornar null (o que pode levar a erros inesperados se não for verificado), Optional força o desenvolvedor a lidar com a possibilidade de ausência de um valor.

- **Como o LivroController lida com o resultado Optional (quando o livro é encontrado e quando não é)?** O LivroController usa o método map() de Optional para transformar o Optional<Livro>:
  - **Quando o livro é encontrado:** map(livro -> new ResponseEntity<>(livro, HttpStatus.OK)) é executado. Ele cria um ResponseEntity com o livro encontrado e o status 200 OK.
  - **Quando o livro não é encontrado:** orElseGet(() -> new ResponseEntity<>(HttpStatus.NOT_FOUND)) é executado. Ele cria um ResponseEntity com o status 404 Not Found, indicando que o recurso não foi encontrado.

3. **Tratamento de Erros:**
   - **Observe os blocos try-catch nos métodos atualizarLivro e deletarLivro do LivroController. Qual o objetivo de capturar a RuntimeException?** O objetivo é **capturar exceções** que podem ocorrer na camada de serviço (como a RuntimeException lançada quando um livro não é encontrado para atualização ou exclusão) e **traduzi-las em uma resposta HTTP apropriada** para o cliente. Isso evita que a aplicação retorne um erro 500 genérico e oferece um feedback mais específico ao cliente.
   - **Que tipo de código de status HTTP é retornado quando um livro não é encontrado para atualização ou exclusão?** É retornado o código de status **HttpStatus.NOT_FOUND (404)**.
4. **Diferença entre save e update:**
   - **No LivroService, o método salvarLivro e atualizarLivro ambos chamam livroRepository.save(). Explique por que save() pode ser usado para ambas as operações (inserção e atualização) no JPA.** No JPA (e Hibernate), o método save() é **polimórfico** em seu comportamento.
     - **Para inserção:** Se o objeto Livro passado para save() **não tem um ID** (ou o ID é nulo/0, indicando que não foi persistido ainda), o JPA entende que é um novo registro e executa uma operação INSERT no banco de dados. Após a inserção, o ID gerado pelo banco de dados é atribuído ao objeto.

- **Para atualização:** Se o objeto Livro passado para save() **já tem um ID válido e existente** no banco de dados (ou seja, ele já foi persistido e está sendo "reatachado" ou "mergeado"), o JPA entende que é uma atualização e executa uma operação UPDATE no registro correspondente no banco de dados.

---

3. Extensão do Projeto

Tarefas e Respostas:

1. **Adicionar Campo de Data de Publicação:**
    - **Como você adicionaria um novo campo dataPublicacao (do tipo LocalDate ou Date) à entidade Livro?**

Java

```
// Em src/main/java/com/livraria/livro/entity/Livro.java

import java.time.LocalDate; // ou java.util.Date;

// ...

@Entity

@Table(name = "livro")

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Livro {

  // ... outros campos

  private LocalDate dataPublicacao; // ou private Date dataPublicacao;

}
```

    - **Que modificações seriam necessárias no DDL do PostgreSQL para incluir este campo?** Você precisaria adicionar a coluna na tabela livro.

SQL

```
ALTER TABLE livro

ADD COLUMN data_publicacao DATE; -- Para LocalDate, o tipo geralmente é DATE
```

(Se você estiver usando ddl-auto=update no application.properties, o Hibernate pode tentar fazer isso automaticamente, mas em produção, usar DDL explícito ou migrações é safer.)

- **Quais métodos no LivroService e LivroController precisariam ser ajustados para permitir o cadastro e atualização deste novo campo?**
  - **LivroService:** No método atualizarLivro, você precisaria adicionar uma linha para copiar a nova data:

Java

livroExistente.setDataPublicacao(livroAtualizado.getDataPublicacao());

O método salvarLivro já funcionaria automaticamente se o campo estiver no JSON de entrada, devido ao Lombok e ao JPA.

  - **LivroController:** Nenhuma alteração de código explícita seria necessária nos métodos @PostMapping e @PutMapping pois o @RequestBody mapearia o JSON automaticamente para o objeto Livro, desde que o JSON de entrada contenha o campo dataPublicacao.

2. **Buscar Livros por Editora:**
   - **Proponha como você implementaria uma funcionalidade para buscar todos os livros de uma determinada editora.**
   - **Quais seriam as alterações necessárias no LivroRepository, LivroService e LivroController para expor este novo endpoint? Defina o URL e o método HTTP.**
     - **LivroRepository.java:** Adicione o método de consulta:

Java

// Em src/main/java/com/livraria/livro/repository/LivroRepository.java

import java.util.List;

// ...

public interface LivroRepository extends JpaRepository<Livro, Long> {

  List<Livro> findByEditora(String editora); // Spring Data JPA cria a query automaticamente

}

- **LivroService.java:** Adicione o método de serviço:

Java

```
// Em src/main/java/com/livraria/livro/service/LivroService.java
// ...
public List<Livro> buscarLivrosPorEditora(String editora) {
  return livroRepository.findByEditora(editora);
}
```

- **LivroController.java:** Adicione o endpoint no controlador:

Java

```
// Em src/main/java/com/livraria/livro/controller/LivroController.java
// ...
@GetMapping("/editora/{editora}") // URL: /api/livros/editora/{nome_da_editora}
public ResponseEntity<List<Livro>> buscarLivrosPorEditora(@PathVariable String editora) {
  List<Livro> livros = livroService.buscarLivrosPorEditora(editora);
  if (livros.isEmpty()) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND); // Ou OK com lista vazia, dependendo da semântica desejada
  }
  return new ResponseEntity<>(livros, HttpStatus.OK);
}
```

- **URL e Método HTTP:** GET /api/livros/editora/{nome_da_editora}

3. **Paginação e Ordenação:**
   - **Como você modificaria o endpoint GET /api/livros para suportar paginação (ex: retornar 10 livros por página)?**
   - **Que parâmetro você esperaria na requisição para indicar a página e o tamanho da página? (Pesquise sobre Pageable no Spring Data JPA).**

- **LivroRepository.java:** A interface JpaRepository já estende PagingAndSortingRepository, que oferece métodos para paginação. Nenhuma alteração é necessária aqui para o método findAll.
- **LivroService.java:** Modifique o método listarTodosLivros para aceitar um Pageable:

Java

```
// Em src/main/java/com/livraria/livro/service/LivroService.java
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
// ...
public Page<Livro> listarTodosLivros(Pageable pageable) {
  return livroRepository.findAll(pageable);
}
```

- **LivroController.java:** Modifique o endpoint GET /api/livros:

Java

```
// Em src/main/java/com/livraria/livro/controller/LivroController.java
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
// ...
@GetMapping
public ResponseEntity<Page<Livro>> listarTodosLivros(Pageable pageable) {
  Page<Livro> livrosPage = livroService.listarTodosLivros(pageable);
  return new ResponseEntity<>(livrosPage, HttpStatus.OK);
}
```

- **Parâmetros na Requisição:** Os parâmetros esperados seriam:

- page: Número da página (começa em 0 por padrão no Spring Data JPA).
- size: Quantidade de itens por página.
- sort: Campo para ordenação e direção (ex: sort=titulo,asc ou sort=qtdPaginas,desc).
- Exemplo de URL: http://localhost:8080/api/livros?page=0&size=5&sort=titulo,asc

4. **Validação de Dados:**
    - **Se você quisesse garantir que o campo titulo de um livro não seja nulo ou vazio, como você faria isso no Livro Entity e/ou no LivroService? (Pesquise sobre jakarta.validation.constraints).** Você usaria anotações de validação da especificação Jakarta Validation (anteriormente JSR 303/380) no campo da Entity. O Spring Boot com spring-boot-starter-validation processa essas anotações automaticamente quando você usa @Valid no Controller.
        - **Livro.java:**

Java

```
// Em src/main/java/com/livraria/livro/entity/Livro.java
import jakarta.validation.constraints.NotBlank; // Para não nulo e não vazio
import jakarta.validation.constraints.Size;
// ...
public class Livro {
  // ...
  @NotBlank(message = "O título não pode ser vazio")
  @Size(max = 100, message = "O título deve ter no máximo 100 caracteres")
  private String titulo;
  // ...
}
```

- **LivroController.java:** Adicione @Valid ao @RequestBody no método cadastrarLivro (e atualizarLivro):

Java

```
// Em src/main/java/com/livraria/livro/controller/LivroController.java

import jakarta.validation.Valid;

// ...

@PostMapping

public ResponseEntity<Livro> cadastrarLivro(@Valid @RequestBody Livro livro) {

  // ...

}


@PutMapping("/{id}")

public ResponseEntity<Livro> atualizarLivro(@PathVariable Long id, @Valid
@RequestBody Livro livro) {

  // ...

}
```

- Se a validação falhar, o Spring automaticamente retornará um 400 Bad Request com os detalhes do erro de validação.

---

4. Reflexão e Melhorias

Perguntas e Respostas:

1. **DTOs (Data Transfer Objects):**
    - **Atualmente, o projeto usa a própria entidade Livro para receber e retornar dados no Controller. Quais são os riscos e desvantagens dessa abordagem?**
        - **Acoplamento:** A API fica acoplada diretamente ao modelo de persistência, dificultando mudanças no banco de dados sem afetar a API.
        - **Exposição de dados sensíveis:** Campos internos do modelo (como senhas, IDs de relacionamento internos, campos de auditoria) podem ser expostos desnecessariamente via API.
        - **Over-fetching/Under-fetching:** O cliente sempre recebe todos os campos da entidade, mesmo que precise de apenas

alguns (over-fetching). Ou, não há como enriquecer a resposta com dados de outras entidades (under-fetching) sem modificar a entidade principal.

- **Segurança:** A entidade pode conter anotações JPA que não são relevantes para a API, e vice-versa. Dados não intencionais podem ser enviados ou recebidos, explorando falhas como "Mass Assignment".
- **Versão da API:** Dificulta a manutenção de diferentes versões da API, pois as alterações na entidade afetam todas as versões.

- **Como a introdução de DTOs (LivroRequestDTO, LivroResponseDTO) poderia melhorar a segurança, a flexibilidade e a clareza da sua API?**
  - **Desacoplamento:** DTOs separam o contrato da API do modelo de persistência, permitindo que cada um evolua independentemente.
  - **Controle de exposição:** Você define explicitamente quais campos são visíveis para o cliente (em DTOs de resposta) e quais campos o cliente pode enviar (em DTOs de requisição). Isso aumenta a segurança e evita "Mass Assignment".
  - **Otimização:** Você pode criar DTOs específicos para cada caso de uso, enviando apenas os dados necessários (reduzindo o tráfego de rede) ou agregando dados de várias entidades para uma resposta mais rica.
  - **Versionamento da API:** Facilita a criação e manutenção de diferentes versões da API, cada uma com seus próprios DTOs.
  - **Clareza:** O DTO pode ter nomes de campos mais amigáveis para a API do que para o banco de dados.

2. **Tratamento Global de Exceções:**
   - **Perceba que a lógica de try-catch para lidar com RuntimeException está replicada em alguns métodos do LivroController. Como você centralizaria o tratamento de exceções em Spring Boot para evitar essa repetição? (Pesquise sobre @ControllerAdvice ou ResponseEntityExceptionHandler).** Para centralizar o tratamento de exceções e evitar a repetição de

blocos try-catch em cada controlador, você pode usar as anotações **@ControllerAdvice** e **@ExceptionHandler**. Cria-se uma classe separada (geralmente chamada GlobalExceptionHandler ou RestExceptionHandler) anotada com @ControllerAdvice. Dentro dessa classe, métodos anotados com @ExceptionHandler podem ser criados para lidar com tipos específicos de exceções. Quando uma exceção for lançada em qualquer controlador, o Spring irá procurar um método @ExceptionHandler correspondente no @ControllerAdvice para tratá-la.

Exemplo simplificado:

Java

```
// Em src/main/java/com/livraria/livro/exception/ResourceNotFoundException.java (exceção customizada)
public class ResourceNotFoundException extends RuntimeException {
  public ResourceNotFoundException(String message) {
    super(message);
  }
}

// Em src/main/java/com/livraria/livro/exception/GlobalExceptionHandler.java
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;

@ControllerAdvice
public class GlobalExceptionHandler {

  @ExceptionHandler(ResourceNotFoundException.class)
```

```
  public ResponseEntity<String>
handleResourceNotFoundException(ResourceNotFoundException ex) {
    return new ResponseEntity<>(ex.getMessage(), HttpStatus.NOT_FOUND);
  }

  @ExceptionHandler(Exception.class) // Catch-all para outras exceções não
tratadas especificamente
  public ResponseEntity<String> handleGeneralException(Exception ex) {
    return new ResponseEntity<>("Ocorreu um erro interno no servidor.",
HttpStatus.INTERNAL_SERVER_ERROR);
  }
}
```

E no LivroService, em vez de new RuntimeException(...), você lançaria new ResourceNotFoundException(...).

3. **Considerações de Produção:**
    - **Quais seriam suas principais preocupações ao levar este projeto para um ambiente de produção? (Pense em segurança, log, monitoramento, etc.).**
        - **Segurança:**
            - **Autenticação e Autorização:** Implementar mecanismos de segurança (Spring Security) para garantir que apenas usuários autorizados acessem os endpoints e com as permissões corretas (ex: JWT, OAuth2).
            - **Validação de Entrada:** Reforçar validações de todos os dados de entrada para prevenir ataques (SQL Injection, XSS, etc.).
            - **Segredos:** Gerenciar credenciais de banco de dados e outras chaves secretas de forma segura (variáveis de ambiente, HashiCorp Vault, AWS Secrets Manager, etc.), não diretamente no application.properties.
            - **HTTPS:** Utilizar HTTPS para criptografar a comunicação entre o cliente e o servidor.

        - **Logs:** Configurar um sistema de log robusto (SLF4J + Logback/Log4j2) com níveis de log adequados para monitorar a aplicação e depurar problemas em produção.
        - **Monitoramento:** Integrar ferramentas de monitoramento (Prometheus, Grafana, Micrometer, Spring Boot Actuator) para coletar métricas de desempenho, saúde da aplicação e logs centralizados.
        - **Confiabilidade/Resiliência:** Implementar mecanismos de circuit breaker, retry, fallback para lidar com falhas de serviços externos.
        - **Performance:** Otimizar queries de banco de dados, usar cache (Spring Cache, Redis) onde apropriado, e garantir que a aplicação possa lidar com a carga esperada (testes de carga).
        - **Backup e Recuperação:** Definir políticas de backup e recuperação de dados do banco.
        - **Containerização/Orquestração:** Empacotar a aplicação em containers Docker e orquestrá-la com Kubernetes para escalabilidade, portabilidade e gerenciamento.
    - **O que você faria de diferente com a propriedade spring.jpa.hibernate.ddl-auto no application.properties em um ambiente de produção?** Em produção, a propriedade spring.jpa.hibernate.ddl-auto nunca deve ser configurada para create, create-drop ou update.
        - A melhor prática é configurá-la para **validate** ou **none**.
        - **validate**: O Hibernate validará o schema do banco de dados existente com o que ele espera do seu modelo JPA. Se houver alguma diferença, a aplicação não irá iniciar, o que é útil para identificar problemas de migração.
        - **none**: O Hibernate não fará nada com o schema do banco de dados. Este é o mais seguro e significa que você é totalmente responsável por gerenciar as migrações do schema, geralmente usando ferramentas dedicadas como **Flyway** ou **Liquibase**. Essas ferramentas permitem um controle versionado e incremental das alterações do seu banco de dados, o que é crucial em ambientes de produção.