**Primeiros Passos**

1. **Crie a classe User, no pacote domain, para representar a entidade User do negócio.**

@Document(collection=”user“)

**public** **class** User **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@Id

**private** String id;

**private** String name;

**private** String email;

//Construtores

//Getters and Setters

//hashCode and equals

}

**Observações:**

**Serializable:** Serve para que as instâncias da classe sejam transformadas em uma sequência de bytes, de modo que possam trafegar em rede e serem salvas em cache. Por mais que geralmente só seja utilizado com a implementação de recursos como ObjectOutputStream e ObjectInputStream, já que é requisito fundamental para que o objeto seja serializado por essas ferramentas, é uma boa prática adicioná-lo em aplicações que implementam JPA/Hibernate [1], [2], [3].

**serialVersionUID:** É um identificador Hash de versão utilizado para identificar se o modelo da classe atual é compatível com qualquer outro modelo de classe antigo que contenha outro ou o mesmo código [1].

1. **Crie a classe UserResource, no pacote resources, para receber as requisições HTTP e executar algoritmos e chamar a camada de serviço em função delas.**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

@RestController

@RequestMapping(value="/users")

**public** **class** UserResource {

@GetMapping

public ResponseEntity<List<User>> findAll(){

List<User> list = service.findAll();

return ResponseEntity.ok().body(list);

}

}

1. **Insira no arquivo pom.xml as dependências do mongodb:**
2. **Crie a interface UserRepository, no pacote repository, para executar o acesso a dados com mongodb.**

@Repository

**public** **interface** UserRepository **extends** MongoRepository<User, String>{

}

1. **Crie a classe UserService, no pacote services, para gerenciar as regras de negócio dos Usuários no sistema.**

@Service

public class UserService {

@Autowired

private UserRepository repo;

public List<User> findAll(){

return repo.findAll();

}

}

1. **Atualize o arquivo application.properties com as configurações de conexão com o banco (as configurações a seguir são para conexão com um banco de dados local):**

spring.data.mongodb.uri=mongodb://localhost:27017/workshop\_mongo

**Observações:**

**Observe que para que essa configuração de conexão dê certo, seu sistema no MongoDB deve conter um cluster rodando na porta 27017 contendo um database chamado workshop\_mongo que, por sua vez, contém uma collection chamada user.**

1. **Crie a classe Instatiation, no pacote config, para que as entidades de negócio sejam instanciadas e inseridas no banco de dados durante o processo de desenvolvimento.**

@Configuration

**public** **class** Instantiation **implements** CommandLineRunner{

@Autowired

**private** UserRepository userRepository;

@Override

**public** **void** run(String... args) **throws** Exception {

userRepository.deleteAll();

User maria = **new** User(**null**, "Maria Brown", "maria@gmail.com");

User alex = **new** User(**null**, "Alex Green", "alex@gmail.com");

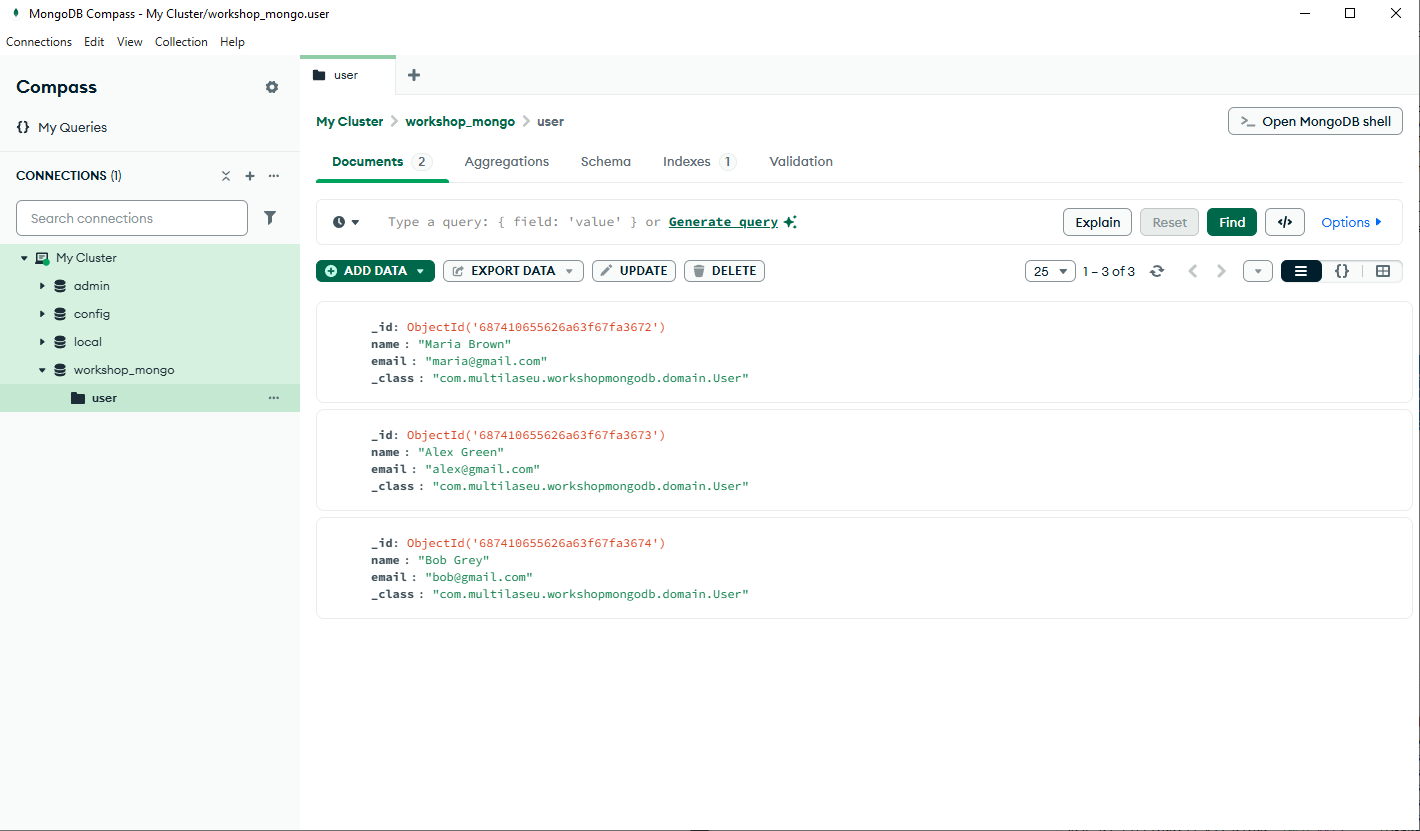
User bob = **new** User(**null**, "Bob Grey", "bob@gmail.com");

userRepository.saveAll(Arrays.*asList*(maria, alex, bob));

}

}

**Observações:**

****

**A partir daqui, a aplicação java já está se comunicando com o banco e consegue efetuar operações com os documents das collections do database.**

1. **Crie o UserDTO, um Data Transfer Object (DTO) no pacote dto, para representar um corpo de User mais enxuto e otimizado para realizar consultas com menos dados de registro no banco.**

**public** **class** UserDTO **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** String id;

**private** String name;

**private** String email;

**public** UserDTO() { }

**public** UserDTO(User obj) {

id = obj.getId();

name = obj.getName();

email = obj.getEmail();

}

//Getters and Setters

}

**Observações:**

**Em projetos mais robustos, UserDTO teria menos atributos do que User, justamente porque a sua intenção é ser uma versão mais enxuta do corpo de um User para evitar que dados de mais trafeguem em rede.**

1. **Atualize o método findAll de UserResource:**

@GetMapping

**public** ResponseEntity<List<UserDTO>> findAll(){

List<User> list = userService.findAll();

List<UserDTO> listDto = list.stream().map(x -> **new** UserDTO(x)).collect(Collectors.*toList*());

**return** ResponseEntity.*ok*().body(listDto);

}

1. **Crie a exceção ObjectNotFoundException, no pacote exception, para ajudar no tratamento de erros relacionados a busca de objetos que não existem no banco de dados.**

public class ObjectNotFoundException extends RuntimeException{

private static final long ***serialVersionUID*** = 1L;

public ObjectNotFoundException(String msg) {

super(msg);

}

}

1. **Adicione o método findById na classe UserService:**

public User findById(String id) {

Optional<User> user = userRepository.findById(id);

if(user == null) {

throw new ObjectNotFoundException("Objeto não encontrado!");

}

return user.get();

}

1. **Adicione o método findById na classe UserResource:**

*@GetMapping*(value="/{id}")

public ResponseEntity<UserDTO> findById(*@PathVariable* String id){

User obj = userService.findById(id);

UserDTO userDto = new UserDTO(obj);

return ResponseEntity.*ok*().body(userDto);

}

1. **Crie a classe StandardError, no pacote resources.exception, para representar o corpo de uma resposta de erro HTTP que será enviada quando uma exceção conhecida for lançada.**

public class StandardError {

private Long timestamp;

private Integer status;

private String error;

private String message;

private String path;

//Construtores

//Getters and Setters

}

1. **Crie a classe ResourceExceptionHandler, no pacote resources.exception, para capturar o lançamento de exceções esperadas e trata-las.**

@ControllerAdvice

public class ResourceExceptionHandler {

@ExceptionHandler(ObjectNotFoundException.class)

public ResponseEntity<StandardError> ObjectNotFound(ObjectNotFoundException e, HttpServletRequest request){

HttpStatus status = HttpStatus.**NOT\_FOUND**;

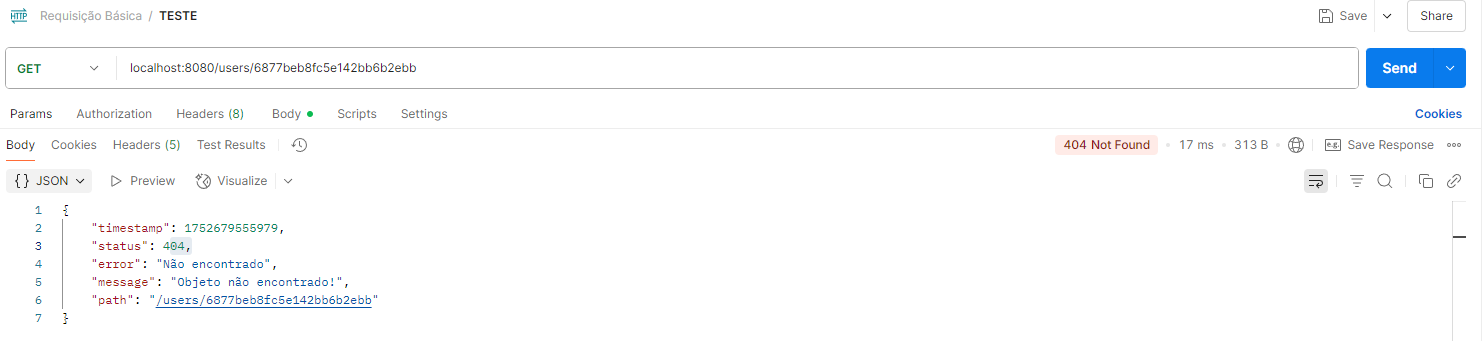
StandardError err = new StandardError(System.currentTimeMillis(), status.value(), "Não encontrado", e.getMessage(), request.getRequestURI());

return ResponseEntity.status(status).body(err);

}

}

**Observações:**



**Agora o método findById está implementado com boas práticas e tratamento de exceções.**

**Aprofundamento**

1. **Atualize o UserService com os métodos insert, para inserção de registros de usuário no banco de dados, e fromDTO, para converter objetos UserDTO em User:**

public User insert(User obj) {

return userRepository.insert(obj);

}

public User fromDTO(UserDTO objDTO) {

return new User(objDTO.getId(), objDTO.getName(), objDTO.getEmail());

}

1. **Atualize o UserResource com o método insert para receber requisições POST que contenham um User como corpo:**

@PostMapping

public ResponseEntity<User> insert(@RequestBody UserDTO objDto){

User user = userService.fromDTO(objDto);

user = userService.insert(user);

URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").

buildAndExpand(user.getId()).toUri();

return ResponseEntity.created(uri).body(user);

}

1. **Atualize UserService com o método delete, para deletar registros de usuários no banco de dados:**

public void delete(String id) {

findById(id);

userRepository.deleteById(id);

}

1. **Atualize UserResource com o método delete, para receber requisições DELETE para apagar registros de usuário do banco de dados:**

@DeleteMapping(value= "/{id}")

public ResponseEntity<Void> delete(@PathVariable String id){

userService.delete(id);

return ResponseEntity.noContent().build();

}

1. **Atualize o UserService com o método update, para realizar atualização nos registros de usuários no banco de dados:**

public User update(User obj) {

Optional<User> newObj = userRepository.findById(obj.getId());

updateData(newObj.get(), obj);

return userRepository.save(newObj.get());

}

private void updateData(User user, User obj) {

user.setName(obj.getName());

user.setEmail(obj.getEmail());

}

1. **Atualize o UserResource com o método update, para receber requisições PUT para atualizar registros de usuário no banco de dados:**

*@PutMapping*(value = "/{id}")

public ResponseEntity<Void> update(*@RequestBody* User user, *@PathVariable* String id){

user.setName(id);

userService.update(user);

return ResponseEntity.*noContent*().build();

}

1. **Crie a classe Post, no pacote domain, para representar os Posts feitos pelos Usuários no sistema.**

**@Document(collection="post")**

**public class Post implements Serializable{**

**private static final long serialVersionUID = 1L;**

**@Id**

**private String id;**

**private Date date;**

**private String title;**

**private String body;**

**private User author;**

**//Construtores**

**//Getters and Setters**

**//hashCode and equals**

**}**

1. **Crie a classe PostRepository, no pacote repositor, para realizar as operações do banco de dados relacionadas à entidade Post.**

@Repository

public interface PostRepository extends MongoRepository<Post, String>{

}

1. **Atualize a classe Instantiation com algumas instâncias de Post para alimentar o banco de dados:**

sdf.setTimeZone(TimeZone.*getTimeZone*("GMT"));

Post post1 = new Post(null, sdf.parse("21/03/2018"), "Partiu Viagem", "Vou viajar para São Paulo. Abraços!", maria);

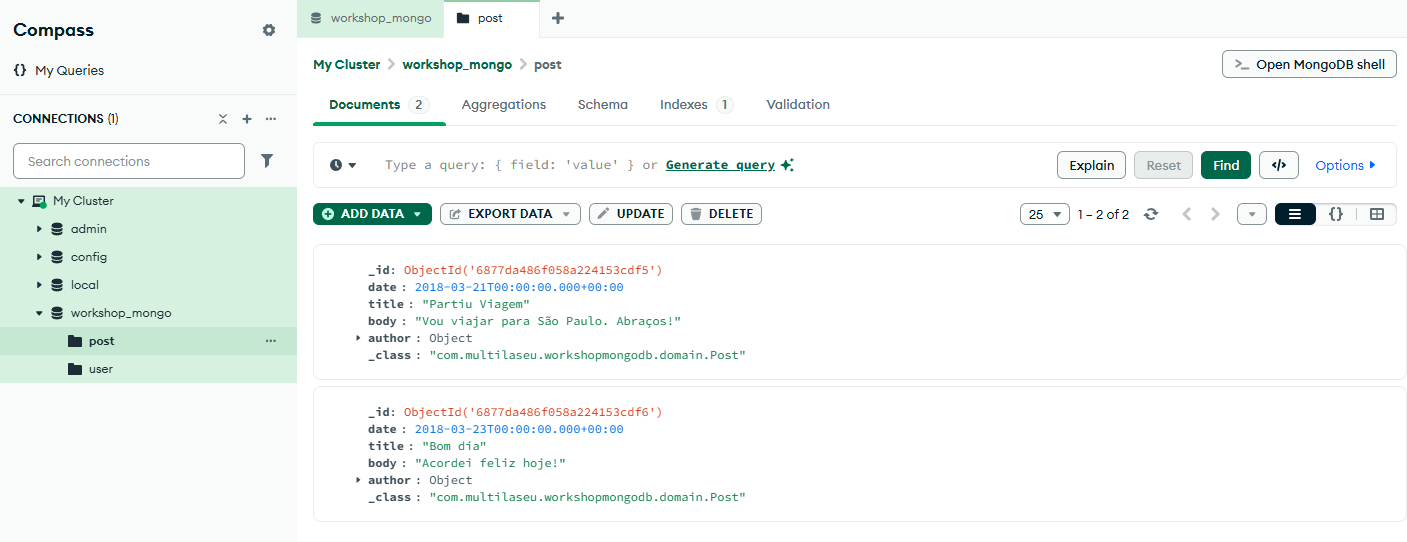
Post post2 = new Post(null, sdf.parse("23/03/2018"), "Bom dia", "Acordei feliz hoje!", maria);

postRepository.deleteAll();

postRepository.saveAll(Arrays.*asList*(post1, post2));

**Observações:**

**Cuidado: Os usuários devem ser salvos antes dos posts.**

****

**As collections post e user estão feitas, mas só o CRUD de User está pronto.**

1. **Crie a classe AuthorDTO, no pacote dto, para representar o atributo author da classe Post como uma forma mais enxuta de User.**

public class AuthorDTO implements Serializable{

private static final long ***serialVersionUID*** = 1L;

private String id;

private String name;

public AuthorDTO() {

}

public AuthorDTO(User obj) {

this.id = obj.getId();

this.name = obj.getName();

}

//Getters and Setters

}

1. **Atualize a classe Post trocando o tipo da variável author para AuthorDTO, e prossiga com as demais alterações que se seguem:**

//Atualize o constructor da classe:

public Post(String id, Date date, String title, String body, User author) {

this.id = id;

this.date = date;

this.title = title;

this.body = body;

this.author = new AuthorDTO(author);

}

//Atualize os Getters and Setters da variável author

public AuthorDTO getAuthor() {

return author;

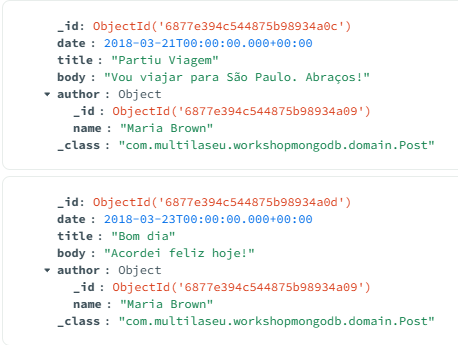
}

public void setAuthor(User author) {

this.author = new AuthorDTO(author);

}

**Observações:**

**É importante observar que na configuração atual do projeto, o atributo author de Post é uma representação aninhada de User, ou seja, uma cópia dos dados de User no banco de dados, e não um vínculo entre coleções. Mais adiante será apresentada uma outra abordagem mais simples, com vínculo e escalável para uma abundância de dados.**

1. **Atualize a classe User com a variável Posts para que os usuários estejam associados aos seus posts:**

*@DBRef*(lazy = true)

private List<Post> posts = new ArrayList<>();

//Getters and Setters

1. **Atualize a classe Instantiation para salvar o registro de posts de um usuário de teste em seu registro de usuário:**

maria.getPosts().addAll(Arrays.*asList*(post1, post2));

userRepository.save(maria);

**Observações:**

**Observe que o código acima deve estar depois do momento em que os posts estão sendo salvos.**

**Como é possível visualizar, a anotação @DBRef estabelece um vínculo entre os registros da coleção post e os da coleção User através dos ids de posts.**

1. **Atualize a classe UserResource com o método findPosts, para resgatar os posts de um usuário:**

*@GetMapping*(value = "/{id}/posts")

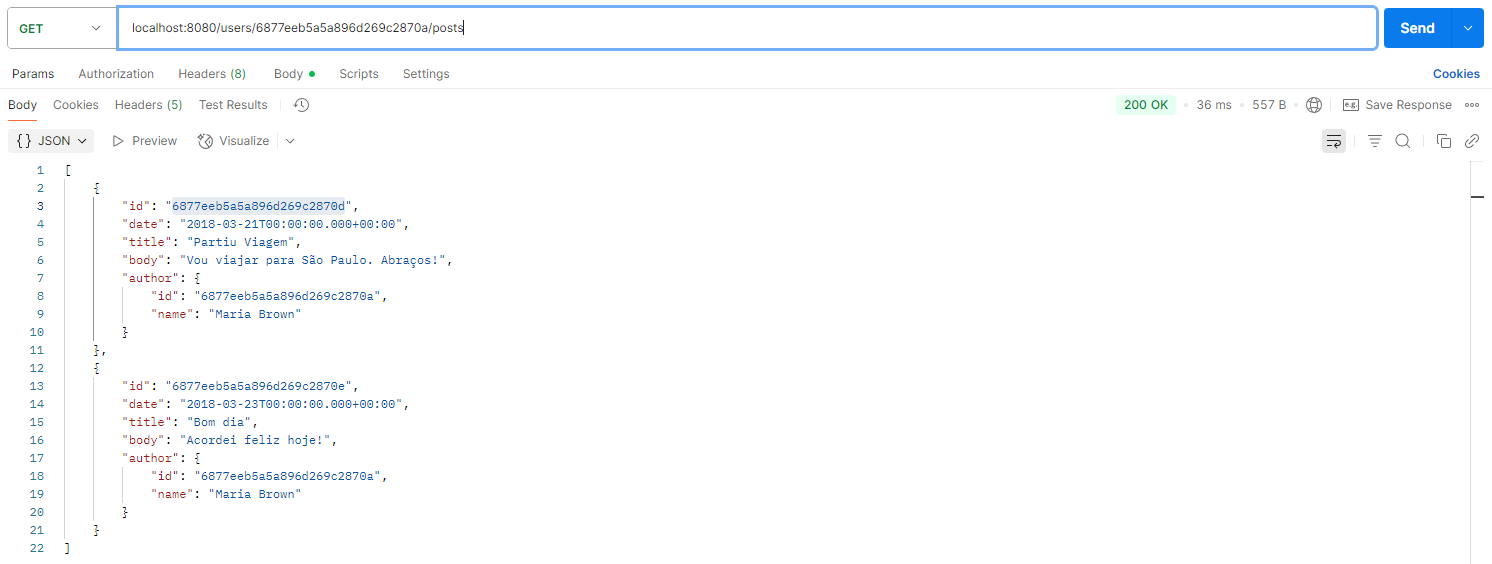
public ResponseEntity<List<Post>> findPost(*@PathVariable* String id){

User user = userService.findById(id);

return ResponseEntity.*ok*().body(user.getPosts());

}

**Observações:**



**Como pode-se observar, agora é possível fazer a busca dos registros de posts de um usuário através de seu id.**

1. **Crie a classe PostService, no pacote services, para que a operação findById seja implementada ao paradigma dos posts.**

@Service

public class PostService {

@Autowired

private PostRepository postRepository;

public Post findById(String id) {

Optional<Post> post = postRepository.findById(id);

if(post.get() == null) {

throw new ObjectNotFoundException("Objeto não encontrado!");

}

return post.get();

}

}

1. **Crie a classe PostResource, no pacote resources, para receber as requisições HTTP relacionadas à entidade Post.**

@RestController

@RequestMapping(value="/posts")

public class PostResource {

@Autowired

private PostService postService;

@GetMapping(value = "/{id}")

public ResponseEntity<Post> findById(@PathVariable String id){

Post post = postService.findById(id);

return ResponseEntity.ok().body(post);

}

}

1. **Crie a classe CommentsDTO, no pacote dto, para representar os comentários feitos nos posts pelos usuários.**

public class CommentDTO implements Serializable{

private static final long ***serialVersionUID*** = 1L;

private String text;

private Date date;

private AuthorDTO author;

//Construtores

//Getters and Setters

}

1. **Atualize a classe Posts com um novo atributo List<CommentDTO>:**

private List<CommentDTO> comments = new ArrayList<>();

//Getters and Setters

1. **Atualize a interface PostRepository com uma query method gerenciada pelo Spring Data:**

List<Post> findByTitleContainingIgnoreCase(String text);

**Observações:**

**findByTitleContainingIgnoreCase:** Esse é um método gerenciado pelo próprio Spring Data pela logística de **query methods**, que são métodos cujos nome definem para o Spring Data como gerar uma requisição baseada nas palavras-chave contidas nele. Neste caso, após findBy insere-se o nome da variável que será buscada dos registros da collection Post; em seguida, uma palavra descreve o que a requisição fará, que no momento é verificar se um text está contido em um Title (o parâmetro da função normalmente é definido por essa palavra); por fim, IgnoreCase é um atributo auxiliar de Containing para ignorar letras maiúsculas e minúsculas.

Para saber mais: <https://docs.spring.io/spring-data/data-document/docs/current/reference/html/>

1. **Atualize a classe PostService com a função findByTitle para rastrear um post pelo que compõe o título:**

public List<Post> findByTitle(String text){

return postRepository.findByTitleContainingIgnoreCase(text);

}

1. **Atualize a classe PostResource com o endpoint findByTitle:**

*@GetMapping*("/titlesearch")

public ResponseEntity<List<Post>> findByTitle(*@RequestParam*(value="text", defaultValue="") String text){

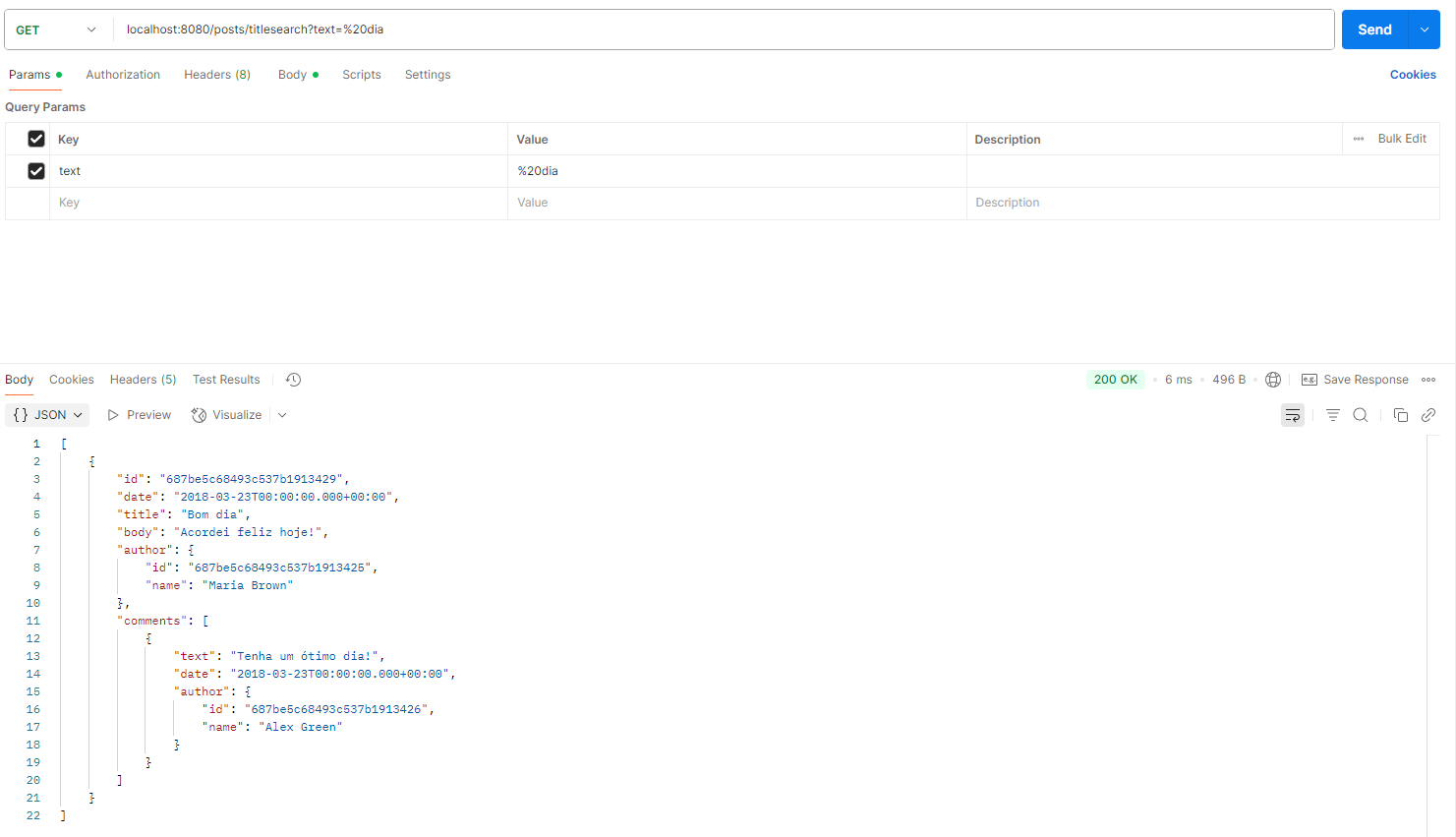
text = URL.*decodeParam*(text);

List<Post> list = postService.findByTitle(text);

return ResponseEntity.*ok*().body(list);

}

**Observações:**

****

**Como pode-se observar, sem qualquer programação explícita a requisição foi montada pelo Spring Data de acordo com o título da função no PostRepository.**

**Bibliografia**

|  |  |
| --- | --- |
| **[1]** | **Entendendo o serialVersionUID | Alura.** Disponível em: <https://www.alura.com.br/artigos/entendendo-o-serialversionuid>. Acesso em: 11 jul. 2025. |
| **[2]** | **Reddit - The heart of the internet**. Disponível em: <https://www.reddit.com/r/javahelp/comments/1cdo4k2/explain\_like\_im\_five\_what\_is\_serializable/?tl=pt-br >. Acesso em: 11 jul. 2025. |
| **[3]** | JAVA EFETIVO (LIVRO. **Item 86: Tenha cautela ao implementar a Serializable**. Disponível em: <https://dev.to/javaefetivo/item-86-tenha-cautela-ao-implementar-a-serializable-258i>. Acesso em: 11 jul. 2025.  ‌ |