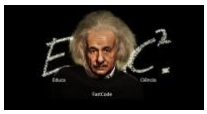


Arquitetura RESTful com Java

Implementação Profunda e Estruturada

1. Configuração do Projeto com Maven

Inicie criando um projeto Maven e adicione as dependências essenciais ao pom.xml, garantindo um ambiente modular e escalável. Inclua as seguintes dependências:

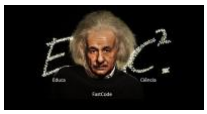
```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

2. Modelagem de Entidade com JPA (Usuario.java)

Implemente a entidade Usuario utilizando as anotações adequadas do JPA. A estrutura deve garantir a integridade e a unicidade dos dados:

```
@Entity
@Table(name = "usuarios")
@Data
@Builder
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String nome;
```



```
    @Column(nullable = false, unique = true)
    private String email;
}
```

Essa abordagem assegura que os campos estejam corretamente validados no nível do banco de dados.

3. Repositório com Consultas Personalizadas (UsuarioRepository.java)

Utilize JpaRepository para operações CRUD e implemente consultas personalizadas para facilitar a busca de dados:

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
    @Query("SELECT u FROM Usuario u WHERE u.email = ?1")
    Optional<Usuario> findByEmail(String email);
}
```

Essa estrutura proporciona flexibilidade e eficiência nas operações de banco de dados.

4. Controlador REST (UsuarioController.java)

Implemente um controlador REST robusto, que respeite os princípios da arquitetura RESTful, e utilize ResponseEntity para retornar respostas adequadas:

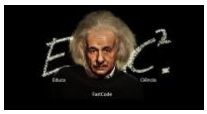
```
@RestController
@RequestMapping("/api/v1/usuarios")
public class UsuarioController {
    @Autowired
    private UsuarioService service;

    @GetMapping
    public ResponseEntity<List<Usuario>> listarTodos() {
        return ResponseEntity.ok(service.listarUsuarios());
    }

    @PostMapping
    public ResponseEntity<Usuario> criarUsuario(@RequestBody Usuario usuario) {
        Usuario novoUsuario = service.criarUsuario(usuario);
        return ResponseEntity.status(HttpStatus.CREATED).body(novoUsuario);
    }

    @GetMapping("/{id}")
    public ResponseEntity<Usuario> buscarPorId(@PathVariable Long id) {
        return service.buscarPorId(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}
```

A implementação de códigos de status HTTP apropriados é crucial para uma comunicação clara entre cliente e servidor.



5. Camada de Serviço e Tratamento de Exceções (UsuarioService.java e GlobalExceptionHandler.java)

Crie uma camada de serviço para centralizar a lógica de negócios, garantindo que o controlador permaneça leve e focado:

```
@Service
public class UsuarioService {
    @Autowired
    private UsuarioRepository repository;

    public Usuario criarUsuario(Usuario usuario) {
        if (repository.findByEmail(usuario.getEmail()).isPresent()) {
            throw new IllegalArgumentException("Email já cadastrado.");
        }
        return repository.save(usuario);
    }

    public Optional<Usuario> buscarPorId(Long id) {
        return repository.findById(id);
    }

    public List<Usuario> listarUsuarios() {
        return repository.findAll();
    }
}
```

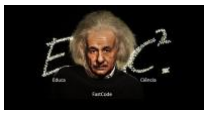
Para um tratamento de exceções eficiente, implemente um GlobalExceptionHandler:

```
@ControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<String> handleIllegalArgumentException(IllegalArgumentException e)
{
        return ResponseEntity.badRequest().body(e.getMessage());
    }
}
```

Esse gerenciamento de exceções garante respostas padronizadas e robustas em caso de falhas.

6. Testes Unitários e de Integração

Implemente uma estratégia abrangente de testes unitários utilizando **JUnit** e **MockMvc** para garantir a integridade e a confiabilidade da aplicação:



```
@SpringBootTest
@AutoConfigureMockMvc
public class UsuarioControllerTest {
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testListarTodos() throws Exception {
        mockMvc.perform(get("/api/v1/usuarios"))
               .andExpect(status().isOk());
    }

    @Test
    public void testCriarUsuario() throws Exception {
        String usuarioJson = "{\"nome\":\"Teste\",\"email\":\"teste@exemplo.com\"}";
        mockMvc.perform(post("/api/v1/usuarios")
                .contentType(MediaType.APPLICATION_JSON)
                .content(usuarioJson))
                .andExpect(status().isCreated());
    }
}
```

Os testes garantem que todos os endpoints funcionem conforme o esperado e detectem falhas antes da implementação em produção.

7. Otimizações e Melhores Práticas

- **Banco de Dados H2**: Utilize H2 como banco de dados em memória, habilitando o console para validações diretas:

  ```
  properties
  spring.h2.console.enabled=true
  spring.datasource.url=jdbc:h2:mem:testdb
  ```

- **Documentação com Swagger**: Implemente **Swagger** para documentar a API de forma dinâmica, facilitando a exploração e testes dos endpoints:

  ```
  <dependency>
      <groupId>io.springfox</groupId>
      <artifactId>springfox-boot-starter</artifactId>
      <version>3.0.0</version>
  </dependency>
  ```

Com esta abordagem detalhada e técnica, sua API REST em Java estará não apenas alinhada às melhores práticas, mas também pronta para escalar e operar em ambientes de produção de alto desempenho, mantendo a integridade, segurança e qualidade do código.