ESCOLA SUPERIOR DA TECNOLOGIA DA INFORMAÇÃO

Desenvolvimento de Serviços Com Spring Boot - TP3

Aluno: Uendel Ives de Araujo

E-mail: uendel.araujo@al.infnet.edu.br

Setembro 2025

Resumo

Este relatório descreve, em detalhes técnicos e justificativos, o desenvolvimento da API REST denominada 'Empresa API'. A aplicação foi realizada em Java 21 com Spring Boot 3.5.5 e implementa CRUD completo para cinco entidades: Employees, Products, Suppliers, Customers e Departments. O objetivo do documento é explicar as escolhas técnicas, arquitetura, endpoints, comportamento HTTP, estratégia de testes, procedimentos para execução e recomendações para avaliação acadêmica.

SUMÁRIO

LINK PARA ACESSAR O CÓDIGO NO GITHUB

https://github.com/uendelAraujoInfnet/web-service-rest-api-crud-tp3

LINK PARA ACESSAR AOS PRINTS / PROVAS

https://github.com/uendelAraujoInfnet/web-service-rest-api-crud-tp3/tree/main/Prints

LINK PARA ACESSAR O POSTMAN

https://www.postman.com/api-help-desk/api-rest-tp3-materia-desenv-serv-spring-boot/overview

1. Requisitos e escopo

O trabalho solicitou a implementação de um serviço web REST com CRUD completo (GET, POST, PUT, DELETE) para cinco recursos distintos. Os dados deveriam ser persistidos via JPA e banco H2 (in-memory) e o projeto deveria incluir testes que cobrissem as funcionalidades desenvolvidas. O escopo entregue neste projeto inclui:
- Entidades: Employee, Product, Supplier, Customer, Department.
- Endpoints REST seguindo padrões RESTful.
- Persistência JPA com H2 configurado para ambiente de desenvolvimento e testes.
- Validações de entrada usando Jakarta Bean Validation.
- Tratamento global de exceções com retorno adequado de status HTTP.
- Testes: Controller (MockMvc), Service (Mockito) e Repository (@DataJpaTest).

2. Escolhas técnicas e gerenciamento de dependências

Linguagem e runtime:
- Java 21: escolhido por estabilidade, suporte a long-term features e compatibilidade com ferramentas modernas.

Frameworks e bibliotecas:
- Spring Boot 3.5.5: versão solicitada, fornece integração entre MVC, Data JPA, e utilitários de configuração.
- Spring Web (spring-boot-starter-web): para criação dos endpoints REST e servidor Tomcat embarcado.
- Spring Data JPA (spring-boot-starter-data-jpa): abstrai persistência e oferece repositórios prontos para CRUD.
- H2 Database: banco em memória, ideal para desenvolvimento e testes automatizados sem infraestrutura externa.
- Spring Boot Starter Validation: Jakarta Bean Validation (JSR-380) para garantir integridade

dos dados de entrada.
- Lombok (opcional): reduz boilerplate de getters/setters/builders para maior produtividade. Opcionalmente pode ser removido.
- spring-boot-starter-test: inclui JUnit5, Mockito, MockMvc e utilitários para testes.

Gerenciamento de dependências:
- Uso do parent `spring-boot-starter-parent` para gerenciamento centralizado de versões.
- Definição explícita da `java.version` e configuração do maven-compiler-plugin com “release” = 21 para consistência.
- Dependências de runtime vs test declaradas com os escopos adequados (ex.: H2 em runtime, starter-test em test).
- Recomenda-se habilitar ferramentas de atualização automática de dependências como Dependabot/ Renovate em repositório.

3. Modelagem de dados e entidades

Para cobrir os requisitos, foram modeladas cinco entidades JPA simples e representativas dos requisitos do domínio. As entidades foram desenhadas privilegiando clareza e campos úteis para testes e operações básicas.

Resumo das entidades:
- Employee: id, name, email (único), role, salary.
- Product: id, name, sku (único), stock, price.
- Supplier: id, name, contactName, email, phone.
- Customer: id, name, email, phone, address.
- Department: id, name, description (exemplo de entidade extra do domínio).

Justificativas:
- Chaves primárias geradas (IDENTITY) facilitam testes com H2 e são adequadas para aplicações monolíticas.
- Campos com restrições de validação nas anotações (ex.: @NotBlank, @Email, @PositiveOrZero) para prevenir dados inválidos.
- Unicidade superficial em campos como email (Employee) e sku (Product) para simular restrições reais do domínio.

4. Implementação dos endpoints REST

Arquitetura e camadas:
- Controller: responsável por expor endpoints HTTP e mapear para objetos de domínio (usando @RestController e @RequestMapping).
- Service: camada de regras mínimas e orchestration; isola lógica do controller e facilita testes unitários com mocks.
- Repository: interfaces que estendem JpaRepository, provendo operações CRUD prontas.

Padrões aplicados:
- Separation of concerns: controllers não acessam diretamente o EntityManager nem

executam lógica de negócios.
- DTOs: neste escopo, optou-se por usar as entidades diretamente para simplificar (adequado para MVP e avaliação acadêmica); para produção recomenda-se introduzir DTOs e mapeamento (MapStruct ou ModelMapper).

Exemplos de endpoints e comportamentos:
- GET /api/employees -> 200 OK (lista de Employees)
- GET /api/employees/{id} -> 200 OK (entidade) ou 404 Not Found quando ausente
- POST /api/employees -> 201 Created + Location header (recurso criado) ou 400 Bad Request em validação
- PUT /api/employees/{id} -> 200 OK com recurso atualizado ou 404 Not Found
- DELETE /api/employees/{id} -> 204 No Content ou 404 Not Found

Mesmo padrão aplicado às outras entidades (Products, Suppliers, Customers, Departments).

5. Validação e tratamento de erros (códigos HTTP)

Validação de entrada:
- Implementada com Jakarta Bean Validation (@NotBlank, @Email, @PositiveOrZero) e ativada nos controllers através de @Valid.
- Quando a validação falha, um handler global captura MethodArgumentNotValidException e retorna 400 Bad Request com um payload descrevendo os erros por campo.

Tratamento de exceções e mapeamento de status:
- ResourceNotFoundException -> mapeado para 404 Not Found (quando um recurso buscado/alterado/excluído não existe).
- MethodArgumentNotValidException -> 400 Bad Request com lista de campos e mensagens.
- Exceções não tratadas -> 500 Internal Server Error (com log para investigação).

Justificativa das escolhas de status:
- Seguir convenções REST torna o comportamento previsível para clientes; por exemplo, 201 Created para POST permite que clientes confiem no Location header para acessar o novo recurso.
- 204 No Content para DELETE deixa explícito que a resposta não contém corpo, economizando banda.

6. Estratégia de testes

Visão geral:
A estratégia combinou testes unitários e testes de integração:
- Controller tests: @WebMvcTest + MockMvc, mockando serviços, validando contratos HTTP, códigos de status e payloads JSON.
- Service tests: Mockito unit tests, validando comportamento quando repositórios retornam resultados esperados ou ausentes (ResourceNotFoundException).
- Repository integration tests: @DataJpaTest com H2 para validar mapeamentos JPA e

operações CRUD reais no banco em memória.

Cobertura e evidência:
- Para cada entidade foram criados: controller tests (create/list/get/update/delete), service tests (casos positivos e negativos) e repository tests.
- Comandos para execução: mvn test (executa todos os testes).

Detalhes importantes sobre os testes
- MockMvc permite testar o comportamento HTTP sem subir a aplicação completa, útil para isolar controllers.
- @DataJpaTest inicializa um contexto limitado do Spring com H2, útil para validar esquemas JPA e anotações.
- Recomenda-se, para entrega acadêmica, incluir evidências: logs do mvn test, screenshots do IDE com testes executando e relatórios gerados.

7. Como executar a aplicação e gerar evidências

Requisitos:
- JDK 21 instalado e configurado no PATH.
- Maven (ou usar ./mvnw se o wrapper estiver presente).

Passos para executar localmente:
1) Build: `mvn clean package`
2) Rodar: `mvn spring-boot:run` (ou `java -jar target/empresa-api-0.0.1-SNAPSHOT.jar`)
3) Acessar endpoints em: http://localhost:8080/api/
4) H2 console: http://localhost:8080/h2-console (JDBC URL: jdbc:h2:mem:empresa-db)

Execução dos testes:
- `mvn test` -> roda unitários e integration tests.
- Para rodar apenas os testes de integração, use tags/filtragem de surefire (opcional).

Postman:
- Collection e Environment foram fornecidos; importe os arquivos JSON no Postman e execute a sequência de testes ou utilize Newman para execução CLI.

Evidências recomendadas para o relatório:
- Prints do terminal mostrando a aplicação iniciada (Spring Boot banner + porta).
- Prints do Postman com requests/responses (inclusive headers Location).
- Saída do `mvn test` mostrando BUILD SUCCESS.
- H2 console com tabelas populadas durante testes (se aplicável).

8. Organização do repositório e documentação do código

Estrutura sugerida do repositório (padrões Maven):

```
src/main/java/com/example/empresaapi
 ├─ controller
 ├─ service
```

```
  ├─ repository
  ├─ model
  ├─ exception
  └─ EmpresaApiApplication.java
src/test/java/...
pom.xml
README.md
.gitignore
```

Documentação:
- README com instruções de build, execução, testes e descrição dos endpoints.
- Postman Collection anexada para facilitar execução por avaliadores.
- Comentários em código apenas quando necessário; prefira nomes autoexplicativos e métodos com responsabilidade única.

9. Sugestões de melhoria e próximas etapas

Para elevar a aplicação a um padrão mais próximo de produção, recomenda-se:
- Introduzir DTOs e MapStruct para separar modelos de persistência da API pública.
- Adicionar camadas de segurança: autenticação (JWT) e autorização (roles).
- Implementar versionamento de API (v1, v2) via URL ou header.
- Substituir H2 por um banco relacional real (PostgreSQL/MySQL) com scripts de migração (Flyway ou Liquibase).
- Implementar monitoramento/observabilidade: logs estruturados, métricas (Micrometer) e traces (OpenTelemetry).
- Adicionar cobertura de testes automatizados contínuos e integração com CI (GitHub Actions) para executar testes e gerar relatório de cobertura (JaCoCo).

10. Conclusão

O projeto implementa de forma clara e consistente os requisitos solicitados: cinco endpoints com CRUD completo, persistência via JPA/H2, validação de dados, tratamento consistente de erros e uma suíte de testes que cobre camadas relevantes (controller, service e repository). As escolhas técnicas foram justificadas para o contexto acadêmico, priorizando clareza, facilidade de avaliação e manutenção.

O package entregue inclui também material de apoio (Postman Collection, .gitignore, README) que facilita a execução e validação por parte dos avaliadores.

Comandos úteis

- Build: mvn clean package

- Run: mvn spring-boot:run

- Test: mvn test