EXPLICAÇÃO COMPLETA DO PROJETO (TAREFA API – SPRING BOOT + JPA + MYSQL)

Autor: Leonam Uirley do Nascimento Cassemiro (RU 4672144) – guia gerado sob demanda

0) VISÃO GERAL RÁPIDA

Você criou uma API REST para gerenciar “tarefas”. Ela foi construída com:

- Spring Boot 3.5.x (Java 17 ou 21)
- Spring Web (para expor endpoints HTTP)
- Spring Data JPA (para integrar com o banco relacional via JPA/Hibernate)
- Driver do MySQL (para conectar ao MySQL)
- Lombok (para gerar automaticamente getters/setters/equals/hashCode/toString e construtores)
- Jackson (embutido no Spring Web – serialização JSON)

O fluxo básico é:

Cliente (Postman / navegador) → Controller (HTTP) → Repository (persistência JPA) → Banco My SQL

e volta com uma resposta JSON.

1) ESTRUTURA DE PASTAS (MUITO IMPORTANTE)

src/

■■ main/

■■ java/

■ ■■ com/tarefaTrabalho/demo/

■ ■■ DemoApplication.java ← classe principal (método main)

■ ■■ controller/ ← TarefaController.java

■ ■■ modelo/ ← Tarefa.java (a entidade JPA)

■ ■■ repositorio/ ← TarefaRepository.java (interface)

■■ resources/

■■ application.properties ← configurações (DB, JPA, etc.)

■■ static/ ← arquivos estáticos (quando houver)

■■ templates/ ← páginas Thymeleaf (quando houver)

REGRAS DE OURO EM JAVA:

- A primeira linha “package ...;” deve refletir exatamente a pasta. Ex.:

```
package com.tarefaTrabalho.demo.repositorio;
```

→ arquivo deve estar em src/main/java/com/tarefaTrabalho/demo/repositorio/...

- Uma CLASSE/INTERFACE pública deve estar em um ARQUIVO com o mesmo nome.

Ex.: “public interface TarefaRepository { ... }” → arquivo deve se chamar TarefaRepository.java

(Seu erro recente foi exatamente isso: o arquivo chamava Repositorio.java, mas a interface públi ca

se chamava TarefaRepository. O compilador exige nomes iguais.)

2) ARQUIVO DemoApplication.java (PONTO DE PARTIDA DA APLICAÇÃO)

Código típico:

```
package com.tarefaTrabalho.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
SpringApplication.run(DemoApplication.class, args);
    }
}
```

O que cada coisa faz:

- @SpringBootApplication: é um “combo” de três anotações:

* @Configuration → marca esta classe como fonte de Beans/Spring context.

* @EnableAutoConfiguration → liga a autoconfiguração do Spring (ele “chuta” configs padrão com base nas dependências).

* @ComponentScan → faz o scan de componentes (controllers, services, repositories) a partir do pacote atual.

- SpringApplication.run(...): inicializa o contexto do Spring, sobe o servidor embutido (Tomcat/Jetty),
registra os Controllers, configura o JPA/Hibernate e conecta ao banco (se as configs estiverem corretas).

- Porta padrão: 8080 (pode mudar via application.properties: server.port=8081).

3) application.properties (CONFIGURAÇÕES)

Exemplo usado:

```
spring.datasource.url=jdbc:mysql://localhost:3306/tarefasdb?createDatabaseIfNotExist=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.open-in-view=true
```

Explicações linha a linha:

- spring.datasource.url: URL de conexão JDBC. “createDatabaseIfNotExist=true” faz o MySQL criar o schema
se não existir (depende da permissão do usuário).

- spring.datasource.username / password: credenciais do MySQL. Ajuste para seu ambiente.

- spring.datasource.driver-class-name: qual driver JDBC será usado (MySQL 8+ usa com.mysql.cj.jdbc.Driver).

- spring.jpa.hibernate.ddl-auto=update: o Hibernate cria/atualiza as tabelas conforme as entidades (bom p/ dev).

(Outros valores: none, validate, create, create-drop).

- spring.jpa.show-sql=true: loga os SQLs no console.

- spring.jpa.open-in-view=true: mantém sessão aberta durante a requisição (útil com lazy-loading — use com cuidado).

Dicas:

- Charset/Timezone do MySQL podem exigir parâmetros na URL (ex.: &useSSL;=false&allowPublicKeyRetrieval;=true).

- Em produção, evite “update” e prefira migrações com Flyway/Liquibase.

4) ENTIDADE JPA: Tarefa.java (DOMÍNIO + MAPEAMENTO)

Local: src/main/java/com/tarefaTrabalho/demo/modelo/Tarefa.java

Código típico:

```
package com.tarefaTrabalho.demo.modelo;

import jakarta.persistence.*;
import lombok.*;
import java.time.LocalDate;

@AllArgsConstructor           // gera construtor com todos os campos
@NoArgsConstructor            // gera construtor sem argumentos
@Data                         // gera getters, setters, equals, hashCode, toString
@Entity                       // marca como entidade JPA
public class Tarefa {

    @Id                       // chave primária
    @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;          // corresponde a coluna BIGINT (auto-incremento)

  private String nome;      // mapeado como VARCHAR

  private LocalDate dataEntrega; // mapeado como DATE

  private String responsavel;// VARCHAR

    }
```

Anotações/Conceitos:

- @Entity: indica ao JPA/Hibernate que esta classe é persistente (gera tabela “tarefa”, por padrão).

- @Id/@GeneratedValue(...IDENTITY): define a PK e usa auto-incremento do banco.

- Tipos Java → Tipos SQL: String → VARCHAR; LocalDate → DATE.

- Lombok:

  * @Data = @Getter + @Setter + @EqualsAndHashCode + @ToString + @RequiredArgsConstructor (mais alguns detalhes).

  * Evita muito código “boilerplate”, mas requer o plugin do Lombok no IntelliJ e a dependência no pom.

- Se quisesse personalizar o nome da tabela/colunas: @Table(name="TAREFA"), @Column(name ="data_entrega", length=60, nullable=false, unique=true) etc.

- Validação (opcional): com spring-boot-starter-validation, você pode anotar:

```
@NotBlank(message="nome é obrigatório"), @Size(min=2,max=60), @NotNull etc.
```

A validação roda no Controller quando você usa @Valid no parâmetro.

5) REPOSITÓRIO JPA: TarefaRepository.java

Local: src/main/java/com/tarefaTrabalho/demo/repositorio/TarefaRepository.java

Código:

```
package com.tarefaTrabalho.demo.repositorio;

import org.springframework.data.jpa.repository.JpaRepository;
import com.tarefaTrabalho.demo.modelo.Tarefa;

public interface TarefaRepository extends JpaRepository<Tarefa, Long> {
// Métodos CRUD já herdados: save, findById, findAll, deleteById, etc.
// Você pode criar consultas derivadas: List findByNome(String nome);
}
```

Explicações:

- É uma interface (sem implementação explícita). O Spring Data gera a implementação em runtime .

- JpaRepository já fornece CRUD completo + paginação/ordenação (Pageable/Sort).

- Injeção: o Spring vai registrar um bean “TarefaRepository” e disponibilizar para os Controllers/Ser vices.

6) CONTROLLER REST: TarefaController.java

Local: src/main/java/com/tarefaTrabalho/demo/controller/TarefaController.java

Código típico:

```
package com.tarefaTrabalho.demo.controller;

import com.tarefaTrabalho.demo.modelo.Tarefa;
import com.tarefaTrabalho.demo.repositorio.TarefaRepository;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController                      // indica que os métodos retornam JSON por padrão
@RequestMapping("/tarefas")          // prefixo comum de rota
public class TarefaController {
private final TarefaRepository repository;

// Injeção de dependência via construtor (boa prática, facilita testes)
    public TarefaController(TarefaRepository repository) {
this.repository = repository;
    }
```

```
    // CREATE – POST /tarefas
            @PostMapping
            public Tarefa criar(@RequestBody Tarefa tarefa) {
        // @RequestBody = lê JSON do corpo e converte (Jackson) para objeto Tarefa
        return repository.save(tarefa);
            }
    // READ ALL – GET /tarefas
            @GetMapping
            public List<Tarefa> listar() {
        return repository.findAll();
            }
            // READ ONE – GET /tarefas/{id}
            @GetMapping("/{id}")
            public ResponseEntity<Tarefa> buscar(@PathVariable Long id) {
        // @PathVariable = lê o {id} da URL
        return repository.findById(id)
              .map(ResponseEntity::ok)          // 200 + corpo
              .orElse(ResponseEntity.notFound().build()); // 404 sem corpo
              }
              // UPDATE – PUT /tarefas/{id}
              @PutMapping("/{id}")
              public ResponseEntity<Tarefa> atualizar(@PathVariable Long id, @RequestBody Tarefa dados) {
                  return repository.findById(id).map(registro -> {
           // atualiza apenas os campos necessários
           registro.setNome(dados.getNome());
           registro.setDataEntrega(dados.getDataEntrega());
           registro.setResponsavel(dados.getResponsavel());
           Tarefa atualizado = repository.save(registro);
           return ResponseEntity.ok(atualizado);
        }).orElse(ResponseEntity.notFound().build());
              }
              // DELETE – DELETE /tarefas/{id}
              @DeleteMapping("/{id}")
              public ResponseEntity<Void> deletar(@PathVariable Long id) {
                  return repository.findById(id).map(registro -> {
           repository.deleteById(id);
           return ResponseEntity.ok().build();
        }).orElse(ResponseEntity.notFound().build());
              }
         }
```

Anotações/Conceitos:

- @RestController: combina @Controller + @ResponseBody, tornando JSON a resposta padrão.

- @RequestMapping("/tarefas"): define o caminho base.

- @GetMapping/@PostMapping/@PutMapping/@DeleteMapping: mapeiam métodos HTTP a métodos Java.

- @RequestBody: o JSON no corpo da requisição vira um objeto Java (via Jackson).

- ResponseEntity: permite customizar status HTTP e cabeçalhos.

- Regras REST usadas:

  * POST cria (201 seria mais semântico, mas 200 OK é aceitável; pode usar “ResponseEntity.created(location).body(...)”).

  * GET lista/busca (200 ou 404).

  * PUT atualiza (200/204 ou 404).

  * DELETE remove (200/204 ou 404).

Observação sobre datas (LocalDate):

- Jackson entende “yyyy-MM-dd” por padrão. Se precisar customizar, usar @JsonFormat(pattern="dd/MM/yyyy").

7) COMO A APLICAÇÃO SOBE (CICLO DE VIDA RESUMIDO)

1. DemoApplication roda → Spring cria o ApplicationContext.

2. AutoConfig: com Spring Web no classpath, sobe o Tomcat embutido (porta 8080).

3. Com Spring Data JPA + MySQL:

   - Cria EntityManagerFactory.

   - Conecta no MySQL com as credenciais do application.properties.

   - Avalia as entidades e o ddl-auto=update para criar/alterar tabelas.

4. ComponentScan encontra @RestController/@Repository e registra os beans.

5. Endpoints ficam disponíveis: /tarefas, /tarefas/{id}, etc.

8) COMO TESTAR (POSTMAN OU cURL)

Base URL: http://localhost:8080

1) Criar (POST /tarefas)

Corpo (JSON):

```
        {
  "nome": "Estudar Spring",
  "dataEntrega": "2025-09-01",
  "responsavel": "Leonam Cassemiro RU4672144"
        }
```

Resposta: 200 (ou 201) com o objeto salvo, incluindo “id”.

2) Listar (GET /tarefas)

Resposta: 200 com array JSON de tarefas.

3) Buscar por ID (GET /tarefas/1)

Resposta: 200 com objeto, ou 404 se não existir.

4) Atualizar (PUT /tarefas/1)

Corpo (JSON):

```
        {
  "nome": "Estudar Spring (atualizado)",
  "dataEntrega": "2025-09-10",
  "responsavel": "Leonam"
        }
```

Resposta: 200 com tarefa atualizada, ou 404 se ID não existir.

5) Excluir (DELETE /tarefas/1)

Resposta: 200/204, ou 404 se não existir.

9) ERROS COMUNS E COMO RESOLVER

A) “interface TarefaRepository is public, should be declared in a file named TarefaRepository.java”

→ O nome do arquivo .java deve ser idêntico ao nome da interface/classe pública.

→ Corrija o nome do arquivo OU o nome da interface.

B) “package ... does not exist” / “wrong package name”

→ A declaração “package com.x.y;” deve refletir a pasta real do arquivo.

→ Pasta: src/main/java/com/x/y/Arquivo.java

→ Primeira linha: package com.x.y;

C) “Cannot determine embedded database driver class” ou erro de conexão

→ Verifique o driver do MySQL, URL, usuário/senha, se o serviço MySQL está rodando.

→ Ex.: jdbc:mysql://localhost:3306/tarefasdb?createDatabaseIfNotExist=true

→ Adicione &allowPublicKeyRetrieval;=true&useSSL;=false se necessário.

D) “Failed to configure a DataSource: 'url' attribute is not specified”

→ Faltou configurar spring.datasource.url (no application.properties).

E) “Whitelabel Error Page” / 404

→ Endpoint digitado errado? Porta correta? Controller foi escaneado?

→ O pacote da classe principal deve “abranger” os demais pacotes (ou ajuste o ComponentScan).

F) “ClassNotFoundException: jakarta…”

→ Confirme que está usando Spring Boot 3.x (migrou para “jakarta.*”) e JDK compatível.

G) Datas em JSON não parseiam

→ Use “yyyy-MM-dd” para LocalDate ou anote com @JsonFormat(pattern="dd/MM/yyyy").

10) (OPCIONAL) CAMADA DE SERVIÇO E VALIDAÇÃO

Se quiser seguir as aulas 4 e 5 mais de perto, pode interpor uma camada de serviço e validação:

- TarefaService (interface) e TarefaServiceImpl (@Service @Transactional):
  * Intermedia Controller ↔ Repository.
  * Onde você aplica regras de negócio.

- Validação (pom: spring-boot-starter-validation):
  * Na entidade:

```
@NotBlank(message="nome é obrigatório")
@Size(min=2, max=60)
@NotNull(message="data é obrigatória")
```

  * No Controller:

```
public String salvar(@Valid @RequestBody Tarefa t, BindingResult result) { ... }
```

  * Para REST, o tratamento de erros fica melhor com um @RestControllerAdvice
    para padronizar mensagens 400 (Bad Request).

11) (OPCIONAL) MELHORIAS DE ARQUITETURA

- DTOs (Data Transfer Objects): separe a entidade do que expõe na API.
- Mapper (MapStruct): converte DTO ↔ Entidade.
- Paginação: use Page findAll(Pageable p) no Repository/Controller.
- Documentação: springdoc-openapi-starter-webmvc-ui para Swagger UI (rota /swagger-ui.html).
- Testes:
  * @SpringBootTest para integração.
  * @WebMvcTest para camada web.
  * @DataJpaTest para repositórios.

12) RESUMO FINAL (O QUE CADA PARTE FAZ)

- DemoApplication: inicia o Spring, servidor, escaneia componentes e sobe a app.
- application.properties: ensina a app a falar com o MySQL e como o JPA deve agir.
- Tarefa (entidade): define a estrutura de dados persistida (tabela) com JPA.
- TarefaRepository: fornece CRUD pronto para a entidade Tarefa.
- TarefaController: recebe requisições HTTP e usa o repository para executar operações,
  retornando respostas JSON padronizadas.

Com isso você tem o panorama completo: do hello world do Spring Boot até o CRUD completo, incluindo detalhes sobre cada anotação, propriedade e fluxo interno.

13) INFORMAÇÃO COMPLEMENTAR (SEU CASO ESPECÍFICO)

No seu projeto, você está usando os seguintes nomes de pacotes/pastas:

- Pacote raiz: com.tarefaTrabalho.demo
- Subpacotes:
  * controller → onde fica TarefaController.java
  * modelo → onde fica Tarefa.java

* repositorio → onde fica TarefaRepository.java

Portanto, as primeiras linhas de cada arquivo devem ser:

1) Tarefa.java (entidade)

```
package com.tarefaTrabalho.demo.modelo;

@Entity
public class Tarefa {
...
}
```

2) TarefaRepository.java (repositório)

```
package com.tarefaTrabalho.demo.repositorio;

public interface TarefaRepository extends JpaRepository<Tarefa, Long> {
}
```

3) TarefaController.java (controller REST)

```
package com.tarefaTrabalho.demo.controller;

@RestController
@RequestMapping("/tarefas")
public class TarefaController {
...
}
```

4) DemoApplication.java (classe principal)

```
package com.tarefaTrabalho.demo;

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
SpringApplication.run(DemoApplication.class, args);
    }
}
```

IMPORTANTE:

Sempre que criar uma classe/interface pública, o nome do ARQUIVO deve ser idêntico ao nome d a classe/interface.

Exemplo: `public interface TarefaRepository {}` → arquivo deve se chamar **TarefaRepository.java **.

Essa padronização evita erros de compilação como o que você encontrou.