

Desenvolvimento de APIs REST

03 - Mapeamento Entidades

- Java Persistence API / Hibernate
- Configuração da Aplicação - Acesso a Banco
- Mapeamento Objeto Relacional
- CRUD

Especificações Java



JSR - Java Specification Request

- Especificações JSR são documentos contendo descrições detalhadas a respeito da plataforma Java.
- Não são implementações ou frameworks em si, mas normas a serem seguidas por quem deseja fabricar um fornecedor de um recurso presente no ambiente Java.
- Máquina Virtual, Linguagem, Bibliotecas (jdbc, collections, etc) são definidas em especificações e depois são implementadas por uma ou mais empresas/organizações
- Tanto o JRE - Java Runtime Environment e o JDK - Java Development Kit são especificações que tem diversas implementações, algumas até mesmo pagas e outras Opensource

Java Standard Edition (JRE e JDK)



IBM SDK, Java Technology Edition, Version 8
The IBM® SDK, Java™ Technology Edition, Version 8™



Amazon Corretto

Distribuição gratuita, multiplataforma e pronta para produção do OpenJDK

Especificações Java



Java Enterprise Edition (JEE) - JSR 366

- Especificação “guarda-chuva” que define a plataforma Java EE.
- Não define as APIs Java EE diretamente, mas as inclui por referência a outras especificações Java
- Define como todas elas se encaixam na plataforma em geral
- Também define outros atributos da plataforma, como segurança, implantação, transações e interoperabilidade.

Uma das APIs definidas e utilizadas na JEE é a Java Persistence API (JPA).

Esta API também é utilizada pelo Spring Boot na sua camada de persistência

- **Java Persistence 2.2**



JEE 8 - 2017

Enterprise Application Technologies

- Batch Applications for the Java Platform 1.0
- Concurrency Utilities for Java EE 1.0
- Contexts and Dependency Injection for Java 2.0
- Dependency Injection for Java 1.0
- Bean Validation 2.0
- Enterprise JavaBeans 3.2
- Interceptors 1.2
- Java EE Connector Architecture 1.7
- **Java Persistence 2.2**
- Common Annotations for the Java Platform 1.3
- Java Message Service API 2.0
- Java Transaction API (JTA) 1.2
- JavaMail 1.6

Web Services Technologies

- Java API for RESTful Web Services (JAX-RS) 2.1
- Implementing Enterprise Web Services 1.3
- Web Services Metadata for the Java Platform 2.1
- Java API for XML-Based RPC (JAX-RPC) 1.1 (Optional)
- Java API for XML Registries (JAXR) 1.0 (Optional)
- Management and Security Technologies
- Java EE Security API 1.0
- Java Authentication Service Provider Interface for Containers 1.1
- Java Authorization Contract for Containers 1.5
- Java EE Application Deployment 1.2 (Optional)
- J2EE Management 1.1
- Debugging Support for Other Languages 1.0
- Java EE-related Specs in Java SE
- Java Management Extensions (JMX) 2.0
- SOAP with Attachments API for Java (SAAJ) Specification 1.3
- Streaming API for XML (StAX) 1.0
- Java API for XML Processing (JAXP) 1.6
- Java Database Connectivity 4.0
- Java Architecture for XML Binding (JAXB) 2.2
- Java API for XML-Based Web Services (JAX-WS) 2.2
- JavaBeans Activation Framework (JAF) 1.1

JPA - Java Persistence API



É uma especificação que descreve como deve ser o comportamento dos frameworks de persistência Java que desejarem implementá-la.

Os frameworks mais conhecidos que utilizam a implementação JPA são: OPenJPA, **Hibernate** e o EclipseLink.



Apache OpenJPA
Fundação
Apache



Eclipselink - Fundação Eclipse (antigo
Toplink, doado pela Oracle) Hoje é a
implementação de referência



**Hibernate, um dos principais
frameworks ORM.
Mantido pela RedHat**

JPA - Java Persistence API



Para utilizarmos JPA é necessário incluir a dependência Spring Boot Starter Data Jpa no pom.xml:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

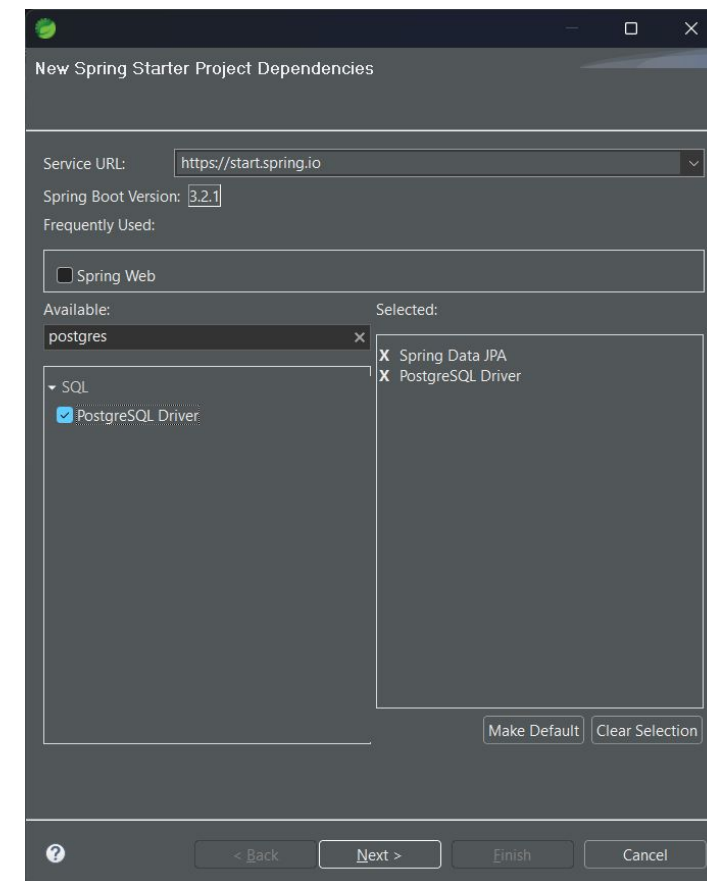
Também devemos incluir a dependência do **Driver** do banco de dados que vamos utilizar (no caso, PostgreSQL):

```
<dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.5.0</version>
</dependency>
```

Visualizando a estrutura do pom.xml temos:

- A Especificação é feita pelo Jakarta Persistence JPA.
- A implementação é feita pelo Hibernate core.

No STS em um projeto existente
botão direito no projeto - **Spring - Add Starters**



Hibernate



Hibernate: Framework ORM - Object Relational Mapping

(Mapeamento Objeto Relacional)

- Diminuir a complexidade na integração de aplicações Java e bancos de dados relacionais
- Abstrair diversas operações sql (INSERT, UPDATE, DELETE)
- Facilitar a realização de consultas
- Interoperabilidade entre Bancos de dados



Hibernate



Diferenças entre a sintaxe SQL

```
-- oracle
select rownum linha, a.*    from
( select * from alunos order by nome ) a
) where
  linha > 10 and -- linha inicial
  linha < 20    -- linha final

--postgres, h2, mysql, sqlite
select * from alunos order by nome
  limit 10 -- itens por pagina
  offset 10      -- linha
  inicial

--sqlServer
select * from alunos order by nome
  offset 10 rows -- linha inicial
  fetch next 10 rows only -- itens por pagina
```

Hibernate/JPA “gera” o SQL específico para o banco configurado

```
//Hibernate
Session session = sessionFactory.openSession();
Query<Aluno> query = session.createQuery("From Aluno", Aluno.class);
query.setFirstResult(20);

query.setMaxResults(10);
List<Aluno> alunosList = query.list();

//JPA
EntityManager entityManager=entityManagerFactory.createEntityManager();
TypedQuery<Aluno> query = entityManager.createQuery("From Aluno", Aluno.class);
query.setFirstResult(20);

query.setMaxResults(10);
List<Aluno> alunosList = query.getResultList();
```

Spring Data



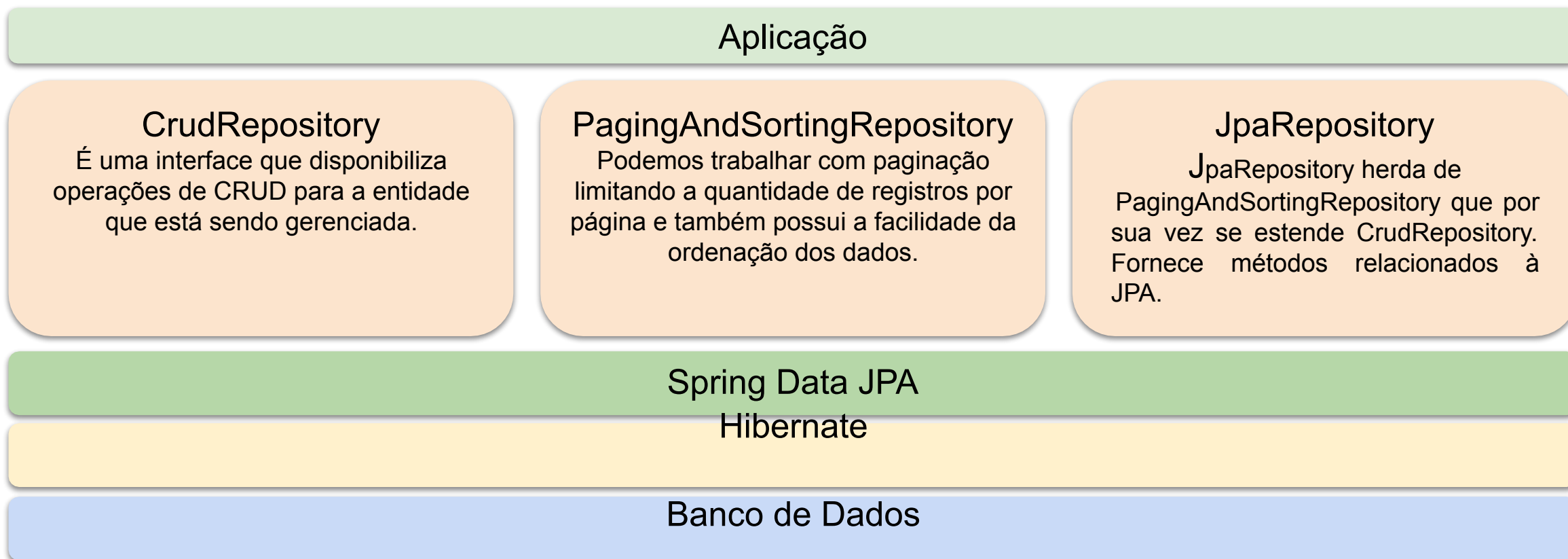
Spring Data tem por finalidade facilitar o acesso a fontes de dados no “estilo” Spring ;-)

- Spring Data JDBC
- Spring Data JPA
- Spring Data LDAP - Lightweight Directory Access Protocol (muito utilizado por serviços de autenticação e permissão)
- Spring Data MongoDB - Banco de dados NoSQL (json)
- Spring Data Redis - Banco em memória, distribuído para armazenamento de dados chave/valor
- Spring Data REST - permite consultar outros serviços REST
- Spring Data for Apache Cassandra - Banco de dados NoSQL
- Spring Data Elasticsearch - Servidor de indexação e buscas baseado no Apache Lucene

Spring Data



É um framework que facilita a implementação do DAO(Data Access Object) reduzindo a quantidade de código. Utilizando-se apenas de uma interface, o Spring irá gerar dinamicamente a implementação dos métodos de acesso a dados.



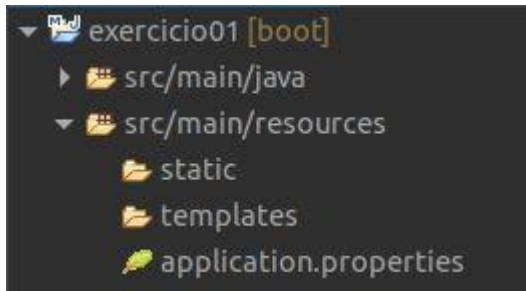
Configuração da Aplicação



O Arquivo **application.properties** do Spring Boot é onde são inseridas diversas configurações.

Propriedades comuns do Spring Boot:

server.port	Porta do servidor (8080)
spring.datasource.url	URL jdbc do banco
spring.datasource.username	Usuário para acesso ao banco
spring.datasource.password	Senha para acesso ao banco



```
Exercicio01Application [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (2 de set de 2022 18:41:16) [pid: 1116124]

:: Spring Boot :: (v2.7.3)

2022-09-02 18:41:18.107 INFO 1116124 --- [main] o.s.b.e.Exercicio01Application : Starting Exercicio01Application using Java 11.0.16 on bula-idk-notebook v
2022-09-02 18:41:18.111 INFO 1116124 --- [main] o.s.b.e.Exercicio01Application : No active profile set, falling back to: default profile: "default"
2022-09-02 18:41:18.932 INFO 1116124 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s) 9090 (http)
2022-09-02 18:41:18.943 INFO 1116124 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-09-02 18:41:18.943 INFO 1116124 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.65]
2022-09-02 18:41:19.025 INFO 1116124 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-09-02 18:41:19.025 INFO 1116124 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 860 ms
2022-09-02 18:41:23.945 INFO 1116124 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s) 9090 (http) with context path ''
2022-09-02 18:41:24.255 INFO 1116124 --- [main] o.s.b.e.Exercicio01Application : Started Exercicio01Application in 6.946 seconds (JVM running for 7.567)
```

Configuração da Aplicação



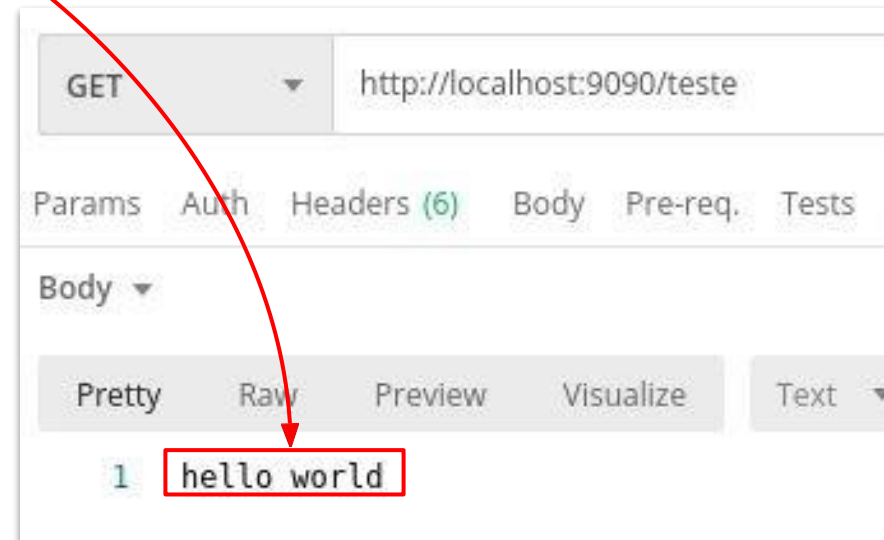
Propriedades customizadas

```
server.port=9090  
minha-propriedade=hello world
```

```
@Value("${minha-propriedade}")  
private String propriedadeCustomizada;  
  
@GetMapping("/teste")  
public String retornaMinhaPropriedade() {  
    return propriedadeCustomizada;  
}
```

IMPORTANTE

O nome da propriedade deve estar entre \${..} na anotação @Value.



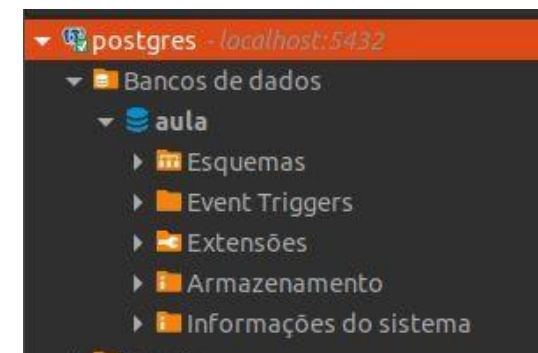
Configuração da Aplicação - Banco de Dados



Adicionar as linhas para conexão com o banco no arquivo
application.properties

```
application.properties x
1 server.port=9090
2
3 spring.datasource.url=jdbc:postgresql://localhost:5432/aula
4 spring.datasource.username=postgres
5 spring.datasource.password=postgres
6 spring.jpa.show-sql=true
7 spring.jpa.hibernate.ddl-auto=update
8
9
```

Criar a base de dados aula utilizando o PGAdmin ou o DBeaver



Executar a aplicação

```
2022-09-03 10:42:17.935 INFO 1156219 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2022-09-03 10:42:17.936 INFO 1156219 --- [main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.65]
2022-09-03 10:42:18.014 INFO 1156219 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2022-09-03 10:42:18.015 INFO 1156219 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1106 ms
2022-09-03 10:42:18.276 INFO 1156219 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [name: default]
2022-09-03 10:42:18.314 INFO 1156219 --- [main] org.hibernate.Version : HHH000412: Hibernate ORM core version 5.6.10.Final
2022-09-03 10:42:18.561 INFO 1156219 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {5.1.2.Final}
2022-09-03 10:42:18.681 INFO 1156219 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2022-09-03 10:42:19.081 INFO 1156219 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2022-09-03 10:42:19.109 INFO 1156219 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.PostgreSQL10Dialect
2022-09-03 10:42:20.038 INFO 1156219 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2022-09-03 10:42:20.050 INFO 1156219 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2022-09-03 10:42:20.157 WARN 1156219 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may
2022-09-03 10:42:20.536 INFO 1156219 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : tomcat started on port(s): 9090 (http) with context path
2022-09-03 10:42:20.557 INFO 1156219 --- [main] o.s.b.e.Exercicio01Application : Started Exercicio01Application in 4.037 seconds (JVM running for 4.389)
```


Configuração da Aplicação - Banco de Dados



Duas propriedades referentes a DDL (Data Definition Language do SQL - CREATE, DROP, ALTER ...)

- **spring.jpa.generate-ddl** - indica se o spring deve ser responsável pela ddl (true ou false)
- **spring.jpa.hibernate.ddl-auto** - como o hibernate irá se comportar:
 - **none**: não realiza nenhuma operação
 - **validate**: valida a estrutura do banco com as entidades mapeadas, se houver diferenças
 - **update**: atualiza a estrutura do banco de acordo com as entidades (não exclui colunas, por segurança)
 - **create**: recria a estrutura do banco sempre
 - **create-drop**: cria a estrutura e a apaga ao final da sessão

O Spring tem um comportamento padrão diferente de acordo com o tipo de banco de dados.

- Bancos “embedded” (H2, HSQLDB e Derby) considera um ambiente de dev e usa create-drop por padrão
- Para outros bancos ele considera none

Configuração da Aplicação - Banco de Dados



Outras propriedades relacionadas ao banco de dados

- **spring.datasource.driverClassName** - classe do driver jdbc (org.h2.Driver ou org.postgres.Driver) - não é necessária pois o hibernate consegue identificar qual driver a partir da url, mas para alguns bancos ou situações específicas pode ser necessário configurar. Sempre é necessário incluir a dependência do driver do banco de dados no pom.xml.
- **spring.jpa.database-platform** - indica a classe do hibernate responsável por tratar o “dialeto” sql do banco
 - org.hibernate.dialect.PostgreSQLDialect
 - org.hibernate.dialect.H2Dialect
- **spring.jpa.properties.hibernate.format_sql** - se ao exibir o sql no console/log, ele deve estar formatado (identado) - (true ou false)

Mapeamento Objeto Relacional



ORM(Object Relaton Mapping)

O mapeamento objeto relacional é a representação de uma tabela de um banco de dados através de classes.

Representação		
Tabela	→	Classe
Coluna	→	Atributo
Registro	→	Instância da Classe (Objeto)

Anotações

Elas ajudam na configuração dos mapeamentos necessários para as classes. As principais anotações usadas no mapeamento relacional são:

- **@Entity**
- **@Table**
- **@Id**
- **@Column**

Domain



- **@Entity** - identifica que a classe é uma entidade do banco
- **@Table** - indica o nome da tabela
- **@Id** - indica que o atributo será mapeado como chave primária
- **@GeneratedValue** - Permite usar valores gerados automaticamente, o atributo **strategy** pode ter os seguintes valores:
 - **GenerationType.IDENTITY** - utiliza uma coluna própria do banco de dados que faça o auto incremento dos valores (no Postgres seria uma coluna do tipo Serial)
 - **GenerationType.SEQUENCE** - utiliza uma **sequence** no banco requer uma anotação adicional para definir a **sequence**
 - **GenerationType.AUTO** - tende variar de acordo com a implementação e até com relação a versão, por exemplo, no Hibernate 5.0 houve uma alteração e ele passou a usar TABLE quando se coloca AUTO.
- **@Column** - Anotação opcional, indica que o atributo é uma coluna da tabela. Possui diversos parâmetros
 - **name** - nome do campo
 - **nullable** - indica se pode receber nulo
 - **length** - tamanho do campo
 - **unique** - se o valor do campo é único
 - **scale** e **precision** - utilizados para definição de tamanho de números
- **@Temporal** - Indica campos do tipo data. Sempre associada a um atributo do tipo java.util.Date. Usa as constantes de TemporalType para indicar o tipo SQL:
 - **TemporalType.DATE** para SQL DATE
 - **TemporalType.TIME** para SQL
 - **TemporalType.TIMESTAMP** para SQL TIMESTAMP

Criar a Classe/Entidade Produto no pacote domain

```
@Entity
@Table(name = "produto")
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id_produto")
    private Long id;
    @Column(name="descricao", nullable=false, length=40)
    private String descricao;
    @Column
    private BigDecimal valor;
    @Column(name="data_cadastro")
    @Temporal(TemporalType.DATE)
    private Date dataCadastro;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    // continua gets sets ...
}
```

Pacote utilizado para a importação das anotações: **jakarta.persistence**

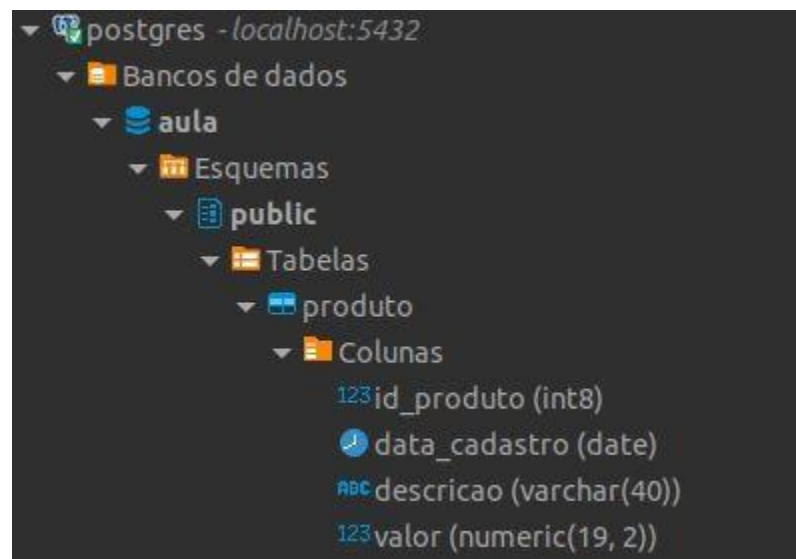
- Inserir **Getters/Setters**
- **Equals** e **HashCode** para o id do **Produto**

Domain



O Hibernate fez a criação automatica da tabela no banco de dados caso a propriedade **spring.jpa.hibernate.ddl-auto** esteja definida para **create**. Recomendação para ambiente de trabalho: deixar opções em false e none e, ou usar um script manual de banco de dados ou utilizar frameworks de gerenciamento de versão de banco, como o **liquibase** ou o **flyway**

```
2022-09-03 11:07:13.760 INFO 1165474 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2022-09-03 11:07:14.096 INFO 1165474 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2022-09-03 11:07:14.120 INFO 1165474 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.PostgreSQL10Dialect
Hibernate: create table produto (id_produto int8 generated by default as identity, data_cadastro date, descricao varchar(40) not null, valor numeric(19, 2), primary key (id_produto))
2022-09-03 11:07:15.832 INFO 1165474 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HHH000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]
2022-09-03 11:07:15.851 INFO 1165474 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2022-09-03 11:07:15.903 WARN 1165474 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. This is not recommended. Use 'spring.jpa.open-in-view=false' to suppress this message.
```



Repositório



Para criar um repositório, basta criar uma interface e estender uma das interfaces:

- CrudRepository<T,ID> - operações de crud
- PagingAndSortingRepository<T,ID> - operações de paginação
- **JpaRepository<T,ID>** - operações específicas associadas a JPA



Repositório - Métodos Providos



CrudRepository

- save - Salva o objeto (INSERT) e retorna o objeto salvo (já com id gerada)
- findOne - retorna um objeto, recebendo o ID como parâmetro (SELECT * WHERE ID=)
- findAll() - retorna todos objetos da tabela (um objeto Iterable) (SELECT *)
- count() - retorna o total de registros na tabela (SELECT COUNT)
- delete - exclui o registro da tabela
- exists - retorna true se o ID existir na tabela (SELECT (COUNT(*)>0) WHERE ID=)

PagingAndSortingRepository

- findAll(Sort) - retorna todos os objetos (um objeto Iterable), mas com suporte a ordenação
- findAll(Pageable) - retorna um objeto Page contendo uma “página” de dados, com base nos valores do objeto Pageable passado

```
Sort sort = Sort.by(Direction.ASC, "lastName");  
Pageable pageable = PageRequest.of(0,5,sort);
```

Repositório - Métodos Providos



JpaRepository

- `findAll()` - retorna uma lista (List) contendo todos os objetos
- `findAll(Sort)` - retorna todos os objetos numa lista (List), mas com suporte a ordenação
- `save(Iterable)` - salva um conjunto de objetos passados num objeto Iterable (uma collection que implemente esta interface), retorna uma lista com os objetos inseridos (salvamento em lote)
- `flush` - JPA pode agrupar operações antes de enviá-las ao servidor de banco de dados, este método “força” que estas sejam enviadas
- `saveAndFlush` - faz as duas operações num único método ;-)
- `deleteInBatch(Iterable)` - recebe um conjunto de objetos passados num objeto Iterable e exclui todos do banco

Repository - Produto



New Java Interface

Create a new Java interface.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ package ☐ private ☐ protected

Extended interfaces:

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

1. Vamos criar nossa interface (File > New > Interface) no pacote **repository** com o nome **ProdutoRepository**
2. Incluir a anotação **@Repository**, que serve para informar que vamos trabalhar na camada de persistência
3. Herdar **JpaRepository** do Spring JPA, informando a classe da entidade e a classe da chave primária na definição dos tipos de generics.

```
@Repository
public interface ProdutoRepository extends JpaRepository<Produto, Long>{

}
```

Controller - Produto



Criar a classe responsável por receber as requisições REST **ProdutoController** e o método para listar todos produtos

```
@RestController
@RequestMapping("/produtos")
public class ProdutoController {

    @Autowired
    private ProdutoRepository produtoRepository;

    @GetMapping
    public List<Produto> listar() {
        return produtoRepository.findAll();
    }
}
```

A anotação **@Autowired** faz a injeção de dependência o que significa que a gerência sobre os objetos será efetuada pelo Spring.

Testando



Insira os registro abaixo no banco de dados, pelo PGAdmin ou pelo DBeaver

```
insert into produto (data_cadastro, descricao, valor)
values ('2021-03-15', 'Celular', 1500);

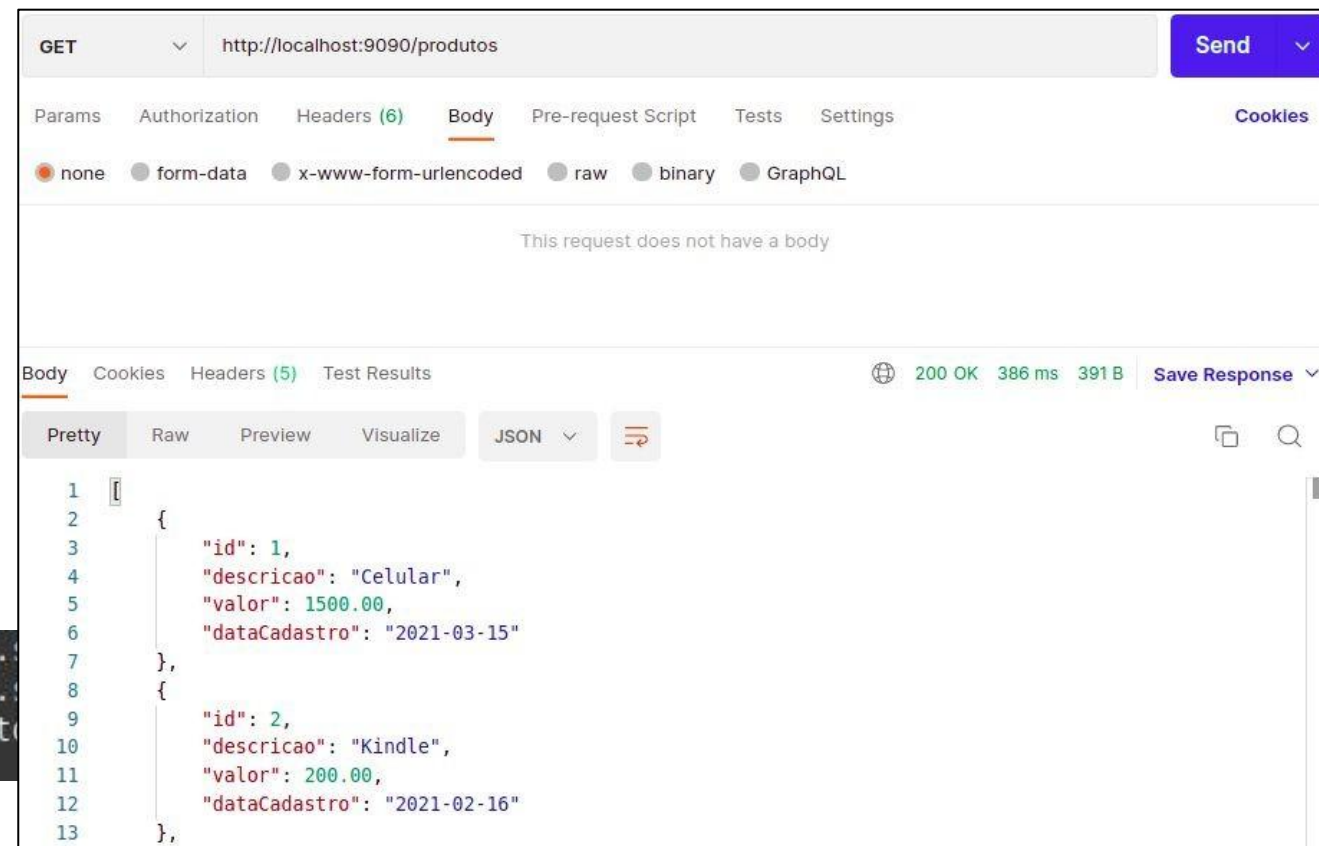
insert into produto (data_cadastro, descricao, valor)
values ('2021-02-16', 'Kindle', 200);

insert into produto (data_cadastro, descricao, valor)
values ('2021-02-22', 'Computador', 2200);
```

No console é possível ver o código SQL gerado pelo JPA

```
2022-09-03 11:51:02.447 INFO 1182128 --- [nio-9090-exec-2] o..
2022-09-03 11:51:02.449 INFO 1182128 --- [nio-9090-exec-2] o..
Hibernate: select produto0_.id_produto as id_produ1_0_, produto
```

Execute a aplicação e acesse o endpoint **<http://localhost:9090/produtos>** no postman (verificar se a porta está configurada no application.properties)



DevTools



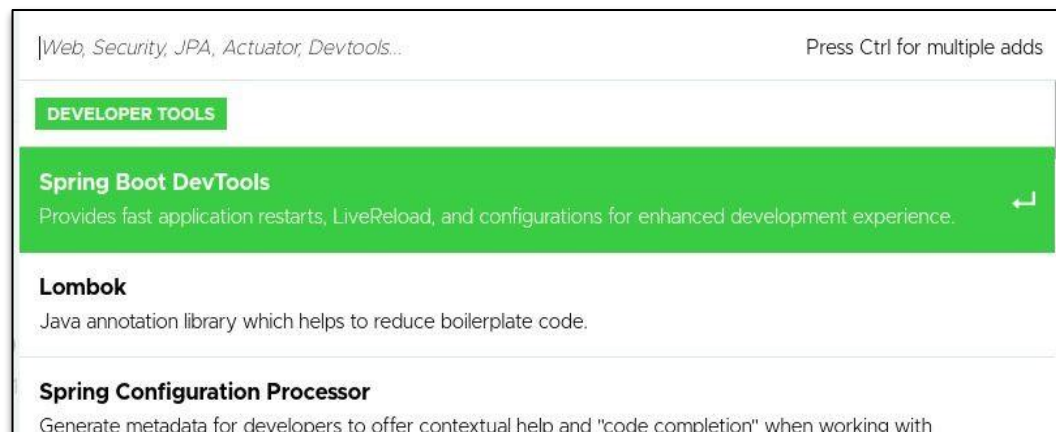
DevTools é uma feature do Spring Boot com o foco em aumentar a produtividade durante o desenvolvimento de aplicações. Para utilização adicionamos a dependência abaixo, ou conforme imagem ao lado na criação do projeto.

<dependency>

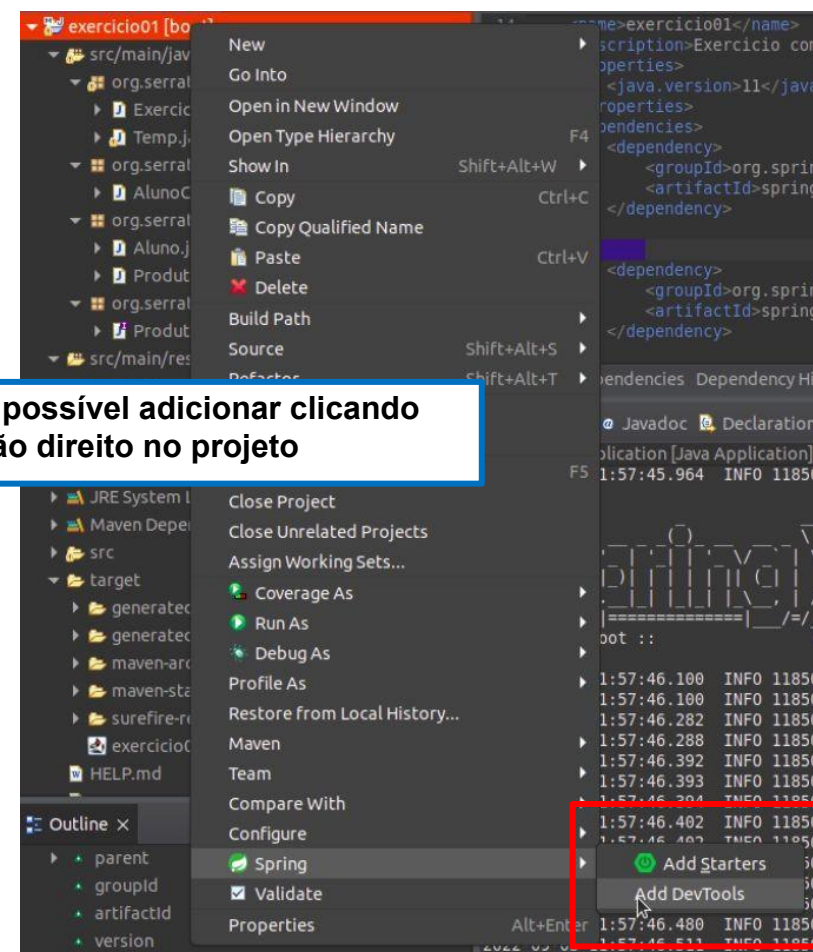
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>



Também é possível adicionar clicando com o botão direito no projeto





Diversas propriedades são incluídas em tempo de execução para acelerar o desenvolvimento

- Desabilitar cache de recursos estáticos (páginas html, imagens, etc...)

Restart automático - Após iniciar a aplicação, ao alterar alguma classe ou arquivo o Spring Boot irá reinicializar seu servidor interno recarregando a aplicação

Livereload - Para aplicações com páginas web, ele pode forçar o browser a recarregar uma página (necessita de extensão no browser - na ChromeWebStore: RemoteLiveReload)

Mais informações <https://www.baeldung.com/spring-boot-devtools>

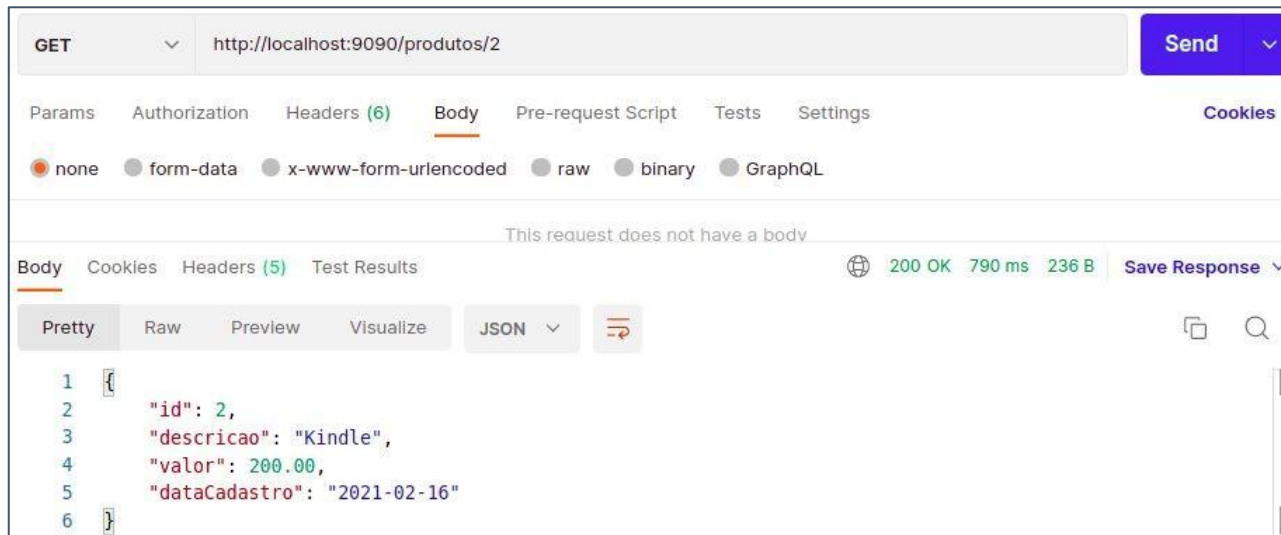
Controller - ResponseEntity



ResponseEntity é uma classe utilitária do Spring Web para auxiliar na resposta HTTP. Representa o tipo de resposta que vai ser retornada.

```
@GetMapping("/{id}")
public ResponseEntity<Produto> pesquisar(@PathVariable Long id) {
    Optional<Produto> produto = produtoRepository.findById(id);
    if (produto.isPresent()) {
        return ResponseEntity.ok(produto.get());
    }
    return ResponseEntity.notFound().build();
}
```

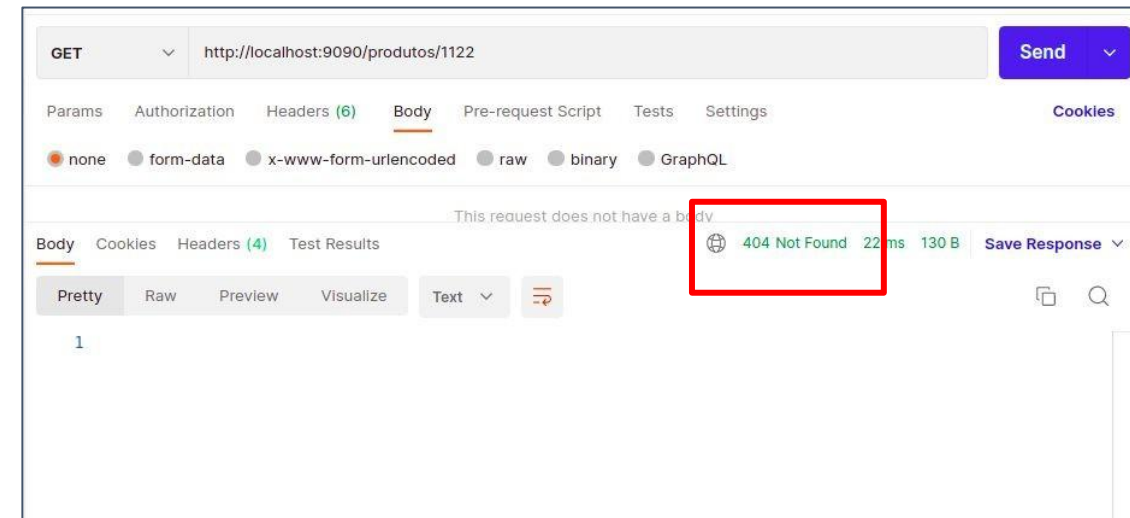
Testando no Postman com um código existente



Método para buscar um Produto

Se existir produto vamos retornar usando o método **get** que busca o que está armazenado no **Optional**, caso não existir retorna 404, o método build retorna um **ResponseEntity**.

Testando no Postman com um código inexistente



ResponseEntity



Código ResponseEntity Method

Padrões	<code>new ResponseEntity<>(HttpStatus.<STATUS>) ResponseEntity.status(HttpStatus.CREATED).build() new ResponseEntity<>(objeto, HttpStatus.<STATUS>) ResponseEntity.status(HttpStatus.CREATED).body(objeto) ResponseEntity.ok().build() ResponseEntity.ok(objeto)</code>
200 - Ok	
201 - Criado HttpStatus.CREATED	<i>ResponseEntity.created(uri).body(objeto) *</i>
204 - Sem Conteúdo	<code>ResponseEntity.noContent().build()</code>
400 - Bad Request	<code>ResponseEntity.badRequest().build() ResponseEntity.badRequest().body(objeto)</code>
404 - não encontrado	<code>ResponseEntity.notFound().build()</code>

Método `created` com parâmetro `uri` deve ser usado com *HATEOAS* tutorial aqui
<https://www.baeldung.com/spring-hateoas-tutorial> Mais informações sobre os códigos http
<https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>

Controller - Inserindo Registro



```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Produto inserir(@RequestBody Produto produto) {
    return produtoRepository.save(produto);
}
```

Adicionado um Produto no banco de dados.
A anotação `@RequestBody` indica que o JSON que vier no corpo do request será transformado em um objeto `Produto`

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** http://localhost:9090/produtos
- Body Tab:** Selected, showing the request body as JSON:

```
{
  "descricao": "Pen Drive",
  "dataCadastro": "2021-03-24",
  "valor": 55.2
}
```
- Response Tab:** Selected, showing the response body as JSON:

```
{
  "id": 4,
  "descricao": "Pen Drive",
  "valor": 55.2,
  "dataCadastro": "2021-03-24T00:00:00.000+00:00"
}
```
- Status:** 201 Created
- Duration:** 420 ms
- Size:** 261 B
- Buttons:** Send, Beautify, Save Response

Controller - Atualizando Registro



```
@PostMapping("/{id}")
public ResponseEntity<Produto> atualizar(@RequestBody Produto produto, @PathVariable Long id) {
    if (!produtoRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    produto = produtoRepository.save(produto);
    return ResponseEntity.ok(produto);
}
```

Inserir o método no **ProdutoController**

Após o teste no Postman para alterar o produto de código 4 tivemos como retorno o código **200 OK**, mas quando conferimos na tabela o registro não foi alterado mas inserido. Por quê?

produto		Enter a SQL expression to filter results (use Ctrl+Space)			
		id produto	data cadastro	descricao	valor
Grade	1	1	2021-03-15	Celular	1.500
	2	2	2021-02-16	Kindle	200
	3	3	2021-02-22	Computador	2.200
Texto	4	4	2021-04-24	Pen Drive	55,2
	5	5	2021-10-20	Cartão de Memória	55,15

PUT localhost:9090/produtos/4

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "descricao": "Cartão de Memória",
3   "dataCadastro": "2021-10-21",
4   "valor": 55.15
5 }
6 }
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 254 ms Size: 248 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 5,
3   "descricao": "Cartão de Memória",
4   "valor": 55.15,
5   "dataCadastro": "2021-10-21"
6 }
```

Controller - Atualizando Registro



```
@PutMapping("/{id}")
public ResponseEntity<Produto> atualizar(@RequestBody Produto produto, @PathVariable Long id) {
    if (!produtoRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    produto.setId(id);
    produto = produtoRepository.save(produto);
    return ResponseEntity.ok(produto);
}
```

Alterando um Produto
no banco de dados.

Atribuímos o **id** ao **setId** pois senão seria criado um novo registro na tabela porque o id vem nulo, ele não é definido no corpo da requisição. No retorno do método **save** vamos atribuir o valor à variável produto.

The screenshot shows a REST client interface with a PUT request to `http://localhost:9090/produtos/4`. The request body is a JSON object: `{ "descricao": "Cartão de Memória", "dataCadastro": "2021-10-21", "valor": 55.15 }`. The response is a 200 OK status with a JSON body: `{ "id": 4, "descricao": "Cartão de Memória", "valor": 55.15, "dataCadastro": "2021-10-21T00:00:00.000+00:00" }`.

Controller - Atualizando Registro



Uma alternativa é carregar o objeto do banco de dados, realizar as alterações necessárias a partir do objeto recebido como parâmetro e salvar o objeto novamente no banco de dados.

```
@PutMapping("/{id}")
public ResponseEntity<Produto> atualizar(@RequestBody Produto produto, @PathVariable Long id) {
    Optional<Produto> produtoOptional = produtoRepository.findById(id);
    if (!produtoOptional.isPresent()) {
        return ResponseEntity.notFound().build();
    }

    Produto produtoDB = produtoOptional.get();
    produtoDB.setDescricao(produto.getDescricao());
    produtoDB.setDataCadastro(produto.getDataCadastro());
    produtoDB.setValor(produto.getValor());

    produto = produtoRepository.save(produtoDB);
    return ResponseEntity.ok(produto);
}
```

Esta alternativa é útil quando alteramos mais de um dado no banco de dados (ex: transferência entre contas, onde se debita de uma conta e credita em outra, atualizando dois registros ao mesmo tempo)

Controller - Apagando Registro



Excluindo um Produto no banco de dados.

```
@DeleteMapping("/{id}")
public ResponseEntity<Void> remover(@PathVariable Long id) {
    if (!produtoRepository.existsById(id)) {
        return ResponseEntity.notFound().build();
    }
    produtoRepository.deleteById(id);
    return ResponseEntity.noContent().build();
}
```

Ao tentarmos excluir novamente o id 4 qual código deverá ser retornado?

DELETE http://localhost:9090/produtos/4

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies Beautify

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   ... "descricao": "Cartão de Memória",
3   ... "dataCadastro": "2021-10-21",
4   ... "valor": 55.15
5 }
```

Body Cookies Headers (3) Test Results 204 No Content 138 ms 112 B Save Response

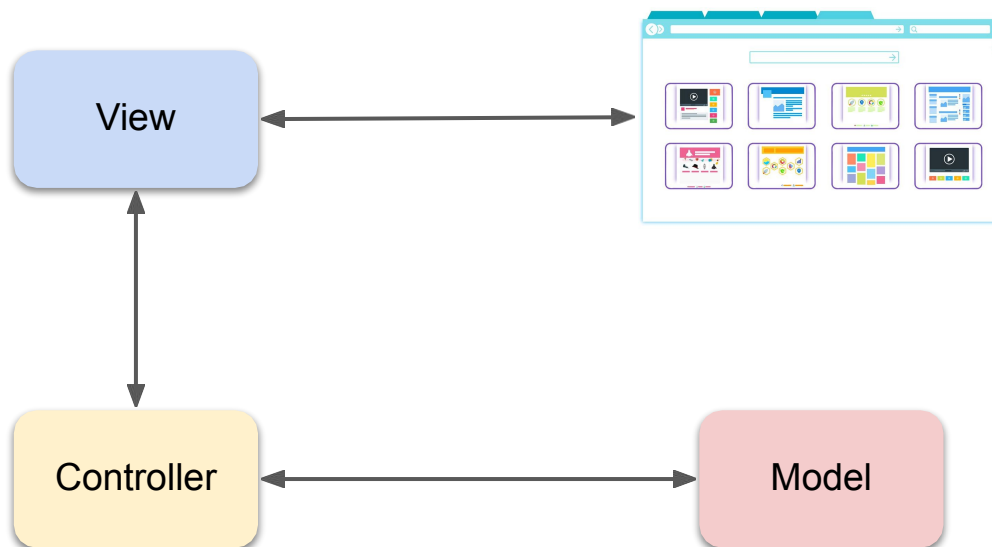
Pretty Raw Preview Visualize Text

Não retornamos corpo só indicamos que deu certo com o código **204** que é um código mais específico para esta situação

MVC

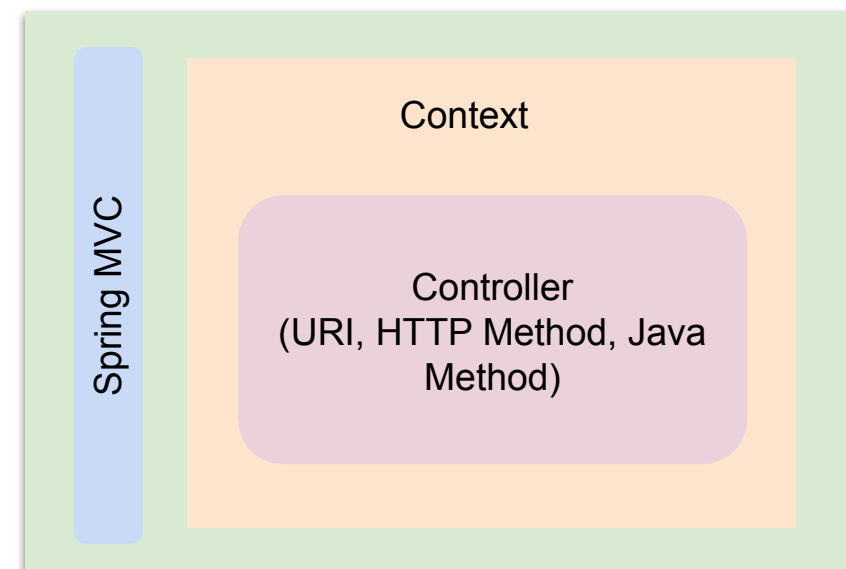


MVC - Model View Controller - Padrão de projeto



- Model - Classes Java - Entidades / Banco
- Controller - Recebe requisições, tratamento de erros HTTP, etc...
- View - Json, XML, Html, PDF, CSV, XLS, Imagen, etc....

Spring MVC - implementação do padrão MVC



Mais informações sobre MVC <https://tableless.com.br/mvc-afinal-e-o-que>

Exercício



Crie uma tabela no banco de dados com o nome **cliente**: **Obs: Não utilizar a criação automática do hibernate.**

Campos da tabela

- Id_cliente - bigint
- nome - varchar (60)
- cpf - varchar (11)
- email - varchar(50)
- data_nascimento
- **Objeto Cliente**
 - id – Long
 - nome – String
 - cpf – String
 - email – String
 - dataNascimento LocalDate
- **API**
 - GET /clientes – lista todos clientes
 - GET /clientes/<id> - retorna o cliente com o id específico.
 - POST /clientes - insere um novo cliente
 - PUT /clientes/<id> - atualiza o cliente (o id não pode ser atualizado)
 - DELETE / clientes /<id> - remove um cliente cujo id foi passado
- Testar no Postman