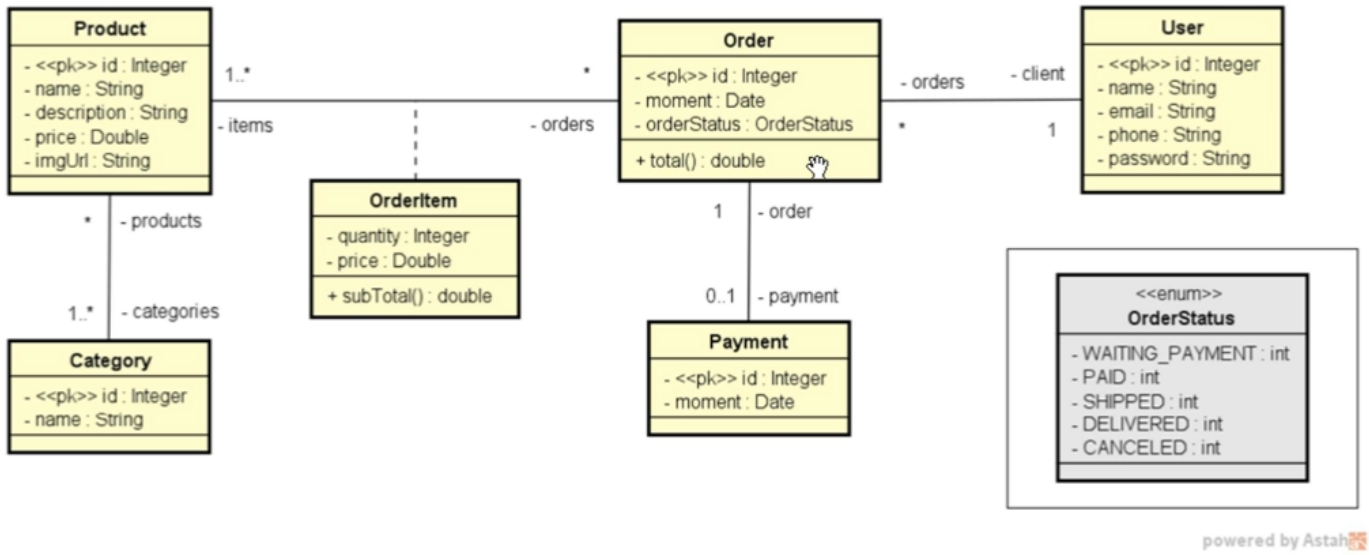
Web Service JPA/Hibernate

Bernardo Seijas Cavalcante

27/06/2025

Objetivo

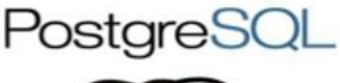


O Objetivo desta atividade é ser um primeiro contato dos alunos no curso Java COMPLETO Programação Orientada a Objetos + Projetos, do professor Nélio Alves, com as tecnologias JPA/Hibernate, Banco de Dados H2, Maven, Postman, Spring Boot e implementação de serviço online com Heroku. No entanto, como Heroku é um serviço pago atualmente e outras tecnologias populares como Render não suportam serviços Java, a API é hospedada localmente.

O Intuito deste arquivo é descrever passo-a-passo a construção do programa, seguindo o diagrama acima, enquanto observações são feitas acerca do que está sendo realizado, assim servindo de material de apoio para consultas futuras e como exercício de aprendizado.

- Criar projeto Spring Boot Java;
- Implementar modelo de domínio (representar entidades do mundo real por classes);
- Estruturar camadas lógicas: resource, service, repositor;
- Configurar Banco de Dados de teste (H2);
- Povoar o banco de dados;
- CRUD – Create, Retrieve, Update, Delete;
- Tratamento de exceções.

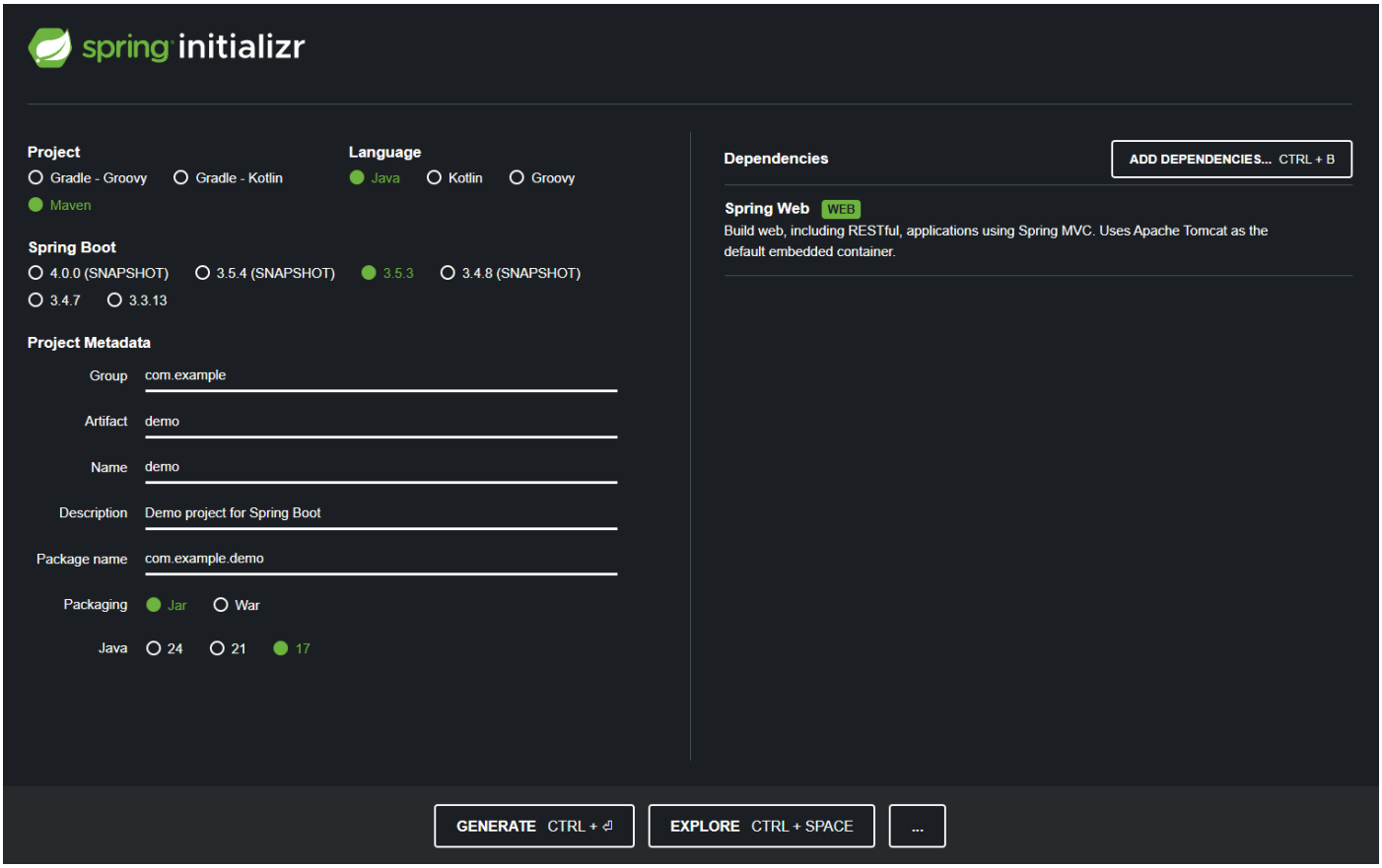
Tecnologias Utilizadas



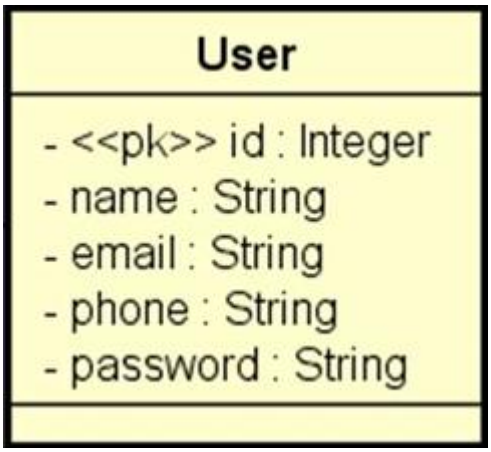
- **Spring Boot:** É um framework muito utilizado atualmente para desenvolvimento de APIs e Web Services em Java pela sua gama de funções prontas para uso e recursos pré-instalados como TomCat para aplicações desse tipo. Um exemplo da sua praticidade está na criação de Rest APIs que com anotações como @RestController e @GetMapping reduzem a quantidade de código necessária para execução de certas tarefas fundamentais para serviços web.
- **Apache Tomcat:** É o servidor embutido para rodar aplicações Java na internet. Quando você configura a aplicação no Spring Initializr com a dependência Spring Web, o Apache Tomcat é instalado junto com Spring MVC (controladores REST) e Jackson (serialização Json). Tanto quando a aplicação é hospedada localmente quanto quando o é em serviços online como Heroku e Render o Apache Tomcat é utilizado para transformar classes Java em servidores Web (servlet), escutando requisições e processando dados.
- **Maven:** É uma ferramenta de automação para desenvolvimento de aplicações Java. Ele ajuda a gerenciar dependências baixando-as do repositório online (Maven central) e a compilar e criar executáveis (.jar e .war) da sua aplicação.
- **Banco de Dados H2:** É um banco de dados salvo em memória feito totalmente em Java, consistindo em linguagem SQL, para execução de aplicações simples e rápidas que não demandem bancos de dados robustos.
- **Postman:** Ferramenta utilizada para fazer testes de requisições HTTP em APIs e Web Services.

Primeiros Passos

1. **Gere o arquivo da aplicação pelo Spring Initializr adicionando como dependência inicial o Spring Web. Depois, coloque o arquivo na pasta que lhe for conveniente à inicialização do Spring Tool Suite.**



2. **Crie a entidade User no pacote entities conforme diagrama e checklist do que tem que ser feito:**



Basic entity checklist:

- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

<NA PRÓXIMA PÁGINA TEM A DEMONSTRAÇÃO DO CÓDIGO>

```
//imports
@Entity //Instruir o sistema que este objeto deve ser convertido para seu modelo relacional (uma tabela)
@Table(name = "tb_user")
public class User implements Serializable{

	//É necessário serializar a classe para que ela trafegue em rede e seja salva em cache. Ela será transformada em uma sequência de bytes
	private static final long serialVersionUID = 1L;

	@Id//Definir atributo como primary key
	@GeneratedValue(strategy = GenerationType.IDENTITY) //Definir que o atributo é auto incrementável
	private Long id;
	private String name;
	private String email;
	private String phone;
	private String password;


	//Construtores

	//Getters and Setters

	//hashCode and equals

}
```

Observações:

@Entity: Indica para Hibernate e ao Spring Boot que a classe em questão é uma entidade JPA, ou seja, deve ser convertida para uma tabela no banco de dados e cada uma das suas instâncias representa um registro da tabela.

@Table: A anotação Table serve para definir o nome da tabela no banco de dados, definir esquemas e restrições às colunas da tabela.

```
@Table(name = "usuarios", schema = "admin")
```

Esquemas: Os esquemas são moldes diferentes para trabalhar com o mesmo banco de dados. Ao acessar um schema, você tem acesso a tabelas com regras de negócio que não necessariamente são iguais as outros esquemas. O esquema padrão é o public, que é o utilizado quando você não define um schema na anotação table.

```
@Table(
   name = "usuarios",
   uniqueConstraints = {
      @UniqueConstraint(columnNames = "email")
   }
)
```

Restrições: As restrições são meios pelos quais as colunas da tabela terão configurações para evitar que os registros das colunas se repitam, ou seja, cada registro terá um dado único na sua coluna com restrição.

Implements Serializable: Serve para a classe ser transformada em uma sequência de bytes, permitindo que ela trafegue em rede (vire dado em trânsito) e seja salva em cache (armazenamento rápido e temporário utilizado para agilizar a consulta de dados já consultados).

---

3. **Crie um UserResource, no pacote resources, para executar comandos em função de requisições HTTP. Essa é a interface de configuração dos endpoints da sua aplicação, ela é o controlador REST que acessa a Service Layer dos usuários. Ela responde aos endpoints GET, PUT, DELETE, POST configurados em seu interior.**

```
@RestController //Falar que essa classe é um recurso Web implementado por um controlador rest
@RequestMapping(value = "/users")//Indica que todos os métodos no interior desta classe responderão a url
/users
public class UserResource {

	@Autowired
	private UserService service;

	@GetMapping//Indica que o método abaixo responde a requisições GET em /users
	public ResponseEntity<List<User>> findAll(){
		List<User> list = service.findAll();
		return ResponseEntity.ok().body(list);//Retorna um HTTP 200 ok com um objeto java u em um
corpo JSON.
	}

	@GetMapping(value = "/{id}")
	public ResponseEntity<User> findById(@PathVariable Long id){
		User obj = service.findById(id);
		return ResponseEntity.ok().body(obj);
	}
}
```

Observações:

@RestController: Diz que essa classe responde a requisições REST (GET, PUT, DELETE, POST, ...) e devolve dados.

@RequestMapping: Serve para definir a URL de acesso aos recursos do Resource. Além disso, ele também pode definir o tipo de requisição de uma função (GET, PUT, POST, DELETE) definida no interior do controlador e a URL de acesso a ela. E também segue o padrão de configuração abaixo:

```
@RequestMapping(
   value = "/users",
   method = RequestMethod.POST,
   consumes = "application/json",
   produces = "application/json",
   headers = "X-Custom-Header=myHeader"
)
public User insert(@RequestBody User user) { ...
}
```

Value: define a URL de acesso ao método.

Method: o tipo de requisição do endpoint.

Consumes: define o tipo de dado aceito como entrada.

Produces: define o tipo de dado aceito como resposta.

Headers: serve para definir um cabeçalho específico de acesso ao endpoint. Se a requisição do client não tiver um dos cabeçalhos definidos, o controlador retornará 404 – Not Found.

@Autowired: Cria uma instância automaticamente para o componente em que for utilizada (variável, parâmetro, etc). É o mesmo que usar "new class_name();". Só funciona com componentes Spring, ou seja, que sejam assinados com anotações @Service, @Repository, @Component ou que implementem a interface JpaRepository.

@GetMapping: Define que o método responde a requisições GET. Opcionalmente, pode conter como parâmetro o endpoint utilizado para obter-se a resposta do método em questão.

ResponseEntity: É uma classe que representa uma resposta HTTP, contendo cabeçalho, status, corpo e etc.

---

4. **Adicione as seguintes dependências ao projeto:**

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

SpringBoot-Starter-Data-JPA: Faz anotações como @Entity, @Id, @OneToMany funcionarem; Adiciona o Hibernate e a abstração de repositórios (JpaRepository) à aplicação.

H2: Adiciona o banco de dados H2 à aplicação.

5. **Abra o arquivo application.properties no caminho src/main/resources e salve ele com esse código:**

```
spring.profiles.active=test
spring.jpa.open-in-view=true
```

Spring.profiles.active: serve para definir o perfil atual como o perfil de teste, ou seja, as configurações de acesso ao banco de dados que serão utilizadas estarão no arquivo application-test.properties (nesse caso), com os critérios de acesso ao database que forem especificados para esse perfil de teste.

Spring. Jpa.open-in-view: Libera a comunicação entre o Jackson (serializador JSON) e o JPA, possibilitando que o serializador consulte o banco de dados para atender a requisições.

6. **Crie o File application-test.properties e coloque o seguinte código dentro dele:**

```
# DATASOURCE

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.url=jdbc:h2:mem:testdb

spring.datasource.username=sa

spring.datasource.password=

# H2 CLIENT

spring.h2.console.enabled=true

spring.h2.console.path=/h2-console

# JPA, SQL

spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.jpa.defer-datasource-initialization=true

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.format_sql=true
```

driverClassName: define o driver JDBC que será utilizado.

Console.enable e .path: Ativa o console padrão do h2 e o configura para ser aberto no endpoint /h2-console.

Database-platform: o Hibernate precisa saber qual é o dialeto que será utilizado para comunicação com o banco.

Datasource-inicialization: Garante que os scripts SQL só sejam executados depois que o banco de dados estiver inicializado e pronto.

Show-sql: Exibe os comandos SQL no console.

Format_sql: Formata a exibição dos comandos SQL no console.

Observação: Do jeito que está até agora com a anotação @Entity no topo da classe User a tabela tb_user já foi criada no banco de dados H2. Para visualizar, basta acessar o caminho: "localhost:8080/h2-console". Mas observe que como o UserService ainda não foi criado, o código de UserResource não estará funcionando.

7. **Crie um repositório UserRepository no pacote repositories para executar operações com a entidade User relacionadas ao seu comportamento no banco de dados, executando INSERTs, DELETEs, UPDATEs, etc.**

```
public interface UserRepository extends JpaRepository<User, Long>{
}
```

JpaRepository<*entidade mapeada, *chave da entidade>

8. **Crie a classe TestConfig para executar testes de configuração do banco de dados com o perfil de teste.**

```
@Configuration
@Profile("test")
public class TestConfig implements CommandLineRunner { //CommandLineRunner para executar o método run assim que o programa começar

	@Autowired //Instancia o objeto userRepository automaticamente
	private UserRepository userRepository;

	@Override
	public void run(String... args) throws Exception {

		User u1 = new User(null, "Maria Brown", "maria@gmail.com", "988888888", "123456");
		User u2 = new User(null, "Alex Green", "alex@gmail.com", "977777777", "123456");

		userRepository.saveAll(Arrays.asList(u1, u2)); //Insere os users na tabela tb_user

	}
}
```

Observações:

Implements CommandLineRunner: interface para executar comandos assim que a aplicação é iniciada. Tudo o que está dentro do método contratado run será executado assim que o programa rodar.

@Configuration: Diz ao Spring que a classe é de configuração e que o que está no interior dela deve ser executado junto com a aplicação. Sem essa anotação, o método run do CommandLineRunner nunca será executado.

@Profile("test"): Indica ao Spring que as configurações do interior da classe só serão executadas se a aplicação estiver rodando no perfil de teste.

9. **Crie a classe UserService, no pacote services, para estabelecer a comunicação entre o RestController UserResource e o UserRepository. É importante criar esse meio de campo entre as duas camadas para que todas as regras de negócio fiquem reservadas ao service, enquanto o restController encarrega-se somente de receber as requisições HTTP nos endpoints especificados e o repositório de executar as operações da entidade com o banco.**

```
@Service
public class UserService {

	@Autowired
	private UserRepository repository;

	public List<User> findAll(){
		return repository.findAll();
	}

	public User findById(Long id) {
		Optional<User> obj = repository.findById(id);
		return obj.get();
	}
}
```

Observações:

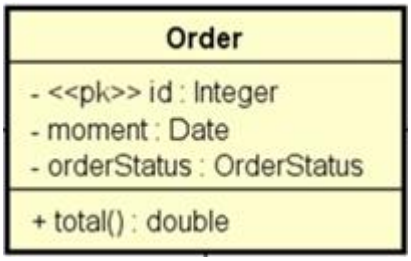
@Service: É importante especificar o tipo de componente Spring da classe para que na injeção de dependência dessas classes em outras no projeto a anotação @Autowired instancie as suas respectivas variáveis corretamente.

<Agora a aplicação pode ser iniciada, pois tem seu princípio de funcionamento concluído>

<Tem mais na próxima página!>

Aprofundamento

10. Crie a classe Order no pacote entities e configure-a de acordo com o seguinte:



Basic entity checklist:

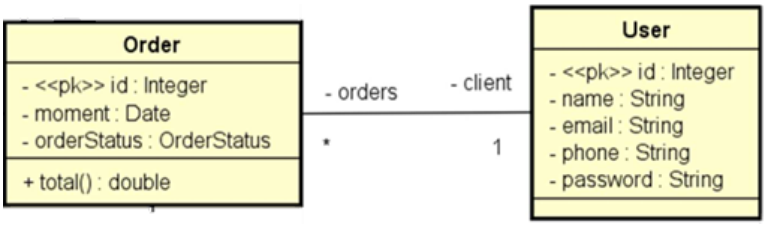
- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

```
@Entity
@Table(name = "tb_order")
public class Order implements Serializable{

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;

	@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd'T'HH:mm:ss'Z'", timezone = "GMT")
	private Instant moment;

	@ManyToOne
	@JoinColumn(name = "client_id")
	private User client;

	//Construtores

	//Getters and Setters

	//hashCode and equals

}
```

Observações:

@JsonFormat: Define um padrão de formatação para a variável moment ser armazenada no banco de dados. Sem isso, o Instant poderia ser serializado em formato numérico. @ManyToOne: Serve para informar ao JPA/Hibernate que há uma relação de muitos para um entre a entidade Order e a entidade User através da variável client. Sendo assim, diz que a variável client será uma chave estrangeira na tabela tb_order, em que um order contém um único user, mas um user pode conter vários orders. Sem essa anotação, o JPA não estabelece a relação entre o campo client_id da tabela tb_order e o Id da tb_user e, por isso, a variável client será ignorada ou mal gerada quando a tabela de Order for criada.

@JoinColumn(name="client_id"): Define o nome de colunas estrangeiras e, nesse caso, define o nome da coluna como "cliente_id". Mas como essa anotação só define o nome de colunas estrangeiras, é importante conhecer a @Column(name = "..."), que altera o nome de colunas convencionais.



11. **Agora que o Order tem sua relação ManyToOne definida, é necessário especificar qual atributo de User (que não existe ainda) possui a relação OneToMany. Para isso, adicione às variáveis da classe User o seguinte código:**

```
@JsonIgnore
@OneToMany(mappedBy = "client")
private List<Order> orders = new ArrayList<>();

[...]

//getOrders

[...]
```

@OneToMany: Ao declarar essa anotação sobre um atributo de uma entidade, você está dizendo ao JPA: "não crie um campo para esta variável na tabela, pois esse fardo já está procedido na outra classe que contém a relação (mappedBy = "client"). E ao mesmo tempo, diz ao Jackson (serializador JSON) que deve-se exibir a variável orders no corpo da resposta JSON.

@JsonIgnore: Diz ao Jackson (serializador JSON) que ele não deve serializar o objeto junto com a variável orders. Isso é importante nesse caso, pois se o serializador transformasse o corpo do User contendo uma lista de Order e, consequentemente, o Order que contém um User, isso geraria um loop de serializações lançando um erro. Nesse sentido, apenas a classe pedido exibe um usuário, mas o usuário não vem junto com seus pedidos.

12. **Crie o OrderRepository, o OrderService e o OrderResource exatamente como nessas definições de User só que trocando a classe User no código por Order (não esqueça de trocar o endpoint do restController de Order para "/orders").**
13. **Faça o teste dos recursos de Order no TestConfig. Adicione o seguinte código ao método run:**

```
Order o1 = new Order(null, Instant.parse("2019-06-20T19:53:07Z"), u1);
Order o2 = new Order(null, Instant.parse("2019-07-21T03:42:10Z"), u2);
Order o3 = new Order(null, Instant.parse("2019-07-22T15:21:22Z"), u1);
orderRepository.saveAll(Arrays.asList(o1,o2,o3));
```

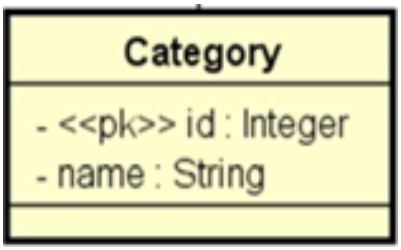
14. **Crie o tipo enumerado OrderStatus, no pacote entities.enum, como uma variável extra de Order:**

```
public enum OrderStatus {
	WAITING_PAYMENT(1),
	PAID(2),
	SHIPPED(3),
	DELIVERED(4),
	CANCELED(5);

	private int code;
	private OrderStatus(int code) {
		this.code = code;
	}
	public int getCode() {
		return code;
	}
	public static OrderStatus valueOf(int code) {
		for(OrderStatus value : OrderStatus.values()) {
			if(value.getCode() == code) {
				return value;
			}
		}
		throw new IllegalArgumentException("Invalid OrderStatus code");
	}

}
```

15. Adicione o atributo OrderStatus na classe Order:

```
[...]
private Integer orderStatus;
[...]
public Order(..., OrderStatus orderStatus, ...){ //Construtor
        [...]
        this.orderStatus = orderStatus.getCode();
        [...]
}
[...]
public OrderStatus getOrderStatus() {
        return OrderStatus.valueOf(orderStatus);
}

public void setOrderStatus(OrderStatus orderStatus) {
        if(orderStatus != null) {
                this.orderStatus = orderStatus.getCode();
        }
}
[...]
```

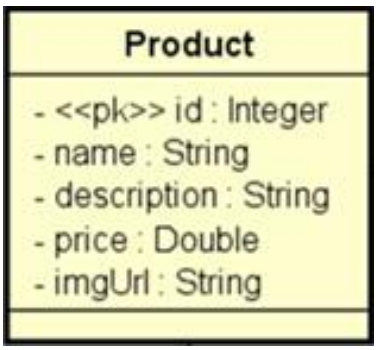
16. Crie a entidade Category, no pacote entities. Crie o CategoryRepository, o CategoryService e o CategoryResource exatamente como nessas definições de User só que trocando a classe User no código por Category (não esqueça de trocar o endpoint do restController de Category para "/categories").



Basic entity checklist:

- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

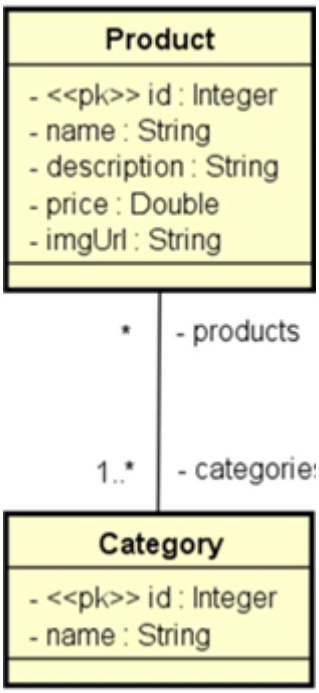
17. Crie a entidade Product, no pacote entities. Crie o ProductRepository, o ProductService e o ProductResource exatamente como nessas definições de User só que trocando a classe User no código por Product (não esqueça de trocar o endpoint do restController de Product para "/products").



Basic entity checklist:

- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

18. Adicione a associação existente entre Product e Category, conforme diagrama e código abaixo:



Insira dentro de Product a variável Set categories:

```
[...]
@ManyToMany
@JoinTable(name = "tb_product_category",
joinColumns = @JoinColumn(name="product_id"),
inverseJoinColumns = @JoinColumn(name="category_id"))
private Set<Category> categories = new HashSet<>();
[...]

//getCategories

[...]
```

Observações:

@ManyToMany: Indica ao JPA que ele terá que criar uma tabela de junção para sustentar a relação muitos para muitos, ou seja, uma tabela que contenha os ids de Category e Product.

@JoinTable: Define o nome da tabela de junção, o nome da coluna do lado da relação em que você está escrevendo o código (joinColumns), e o nome da coluna do outro lado da relação (inverseJoinColumns).

Insira dentro de Category a variável Set products:

```
[...]
@JsonIgnore
@ManyToMany(mappedBy = "categories")
private Set<Product> products= new HashSet<>();
[...]

//getProducts

[...]
```

Observação: Recomenda-se utilizar Set em relações muitos para muitos.

<Tem mais na próxima página!>

19. **Adicione ao TestConfig o seguinte código (não esqueça de fazer a injeção de dependência com o ProductRepository e o CategoryRepository:**

```
		Category cat1 = new Category(null, "Electronics");
		Category cat2 = new Category(null, "Books");
		Category cat3 = new Category(null, "Computers");

		categoryRepository.saveAll(Arrays.asList(cat1, cat2, cat3));

		Product p1 = new Product(null, "The Lord of the Rings", "Lorem ipsum dolor sit amet,
consectetur.", 90.5, "");
		Product p2 = new Product(null, "Smart TV", "Nulla eu imperdiet purus. Maecenas ante.",
2190.0, "");
		Product p3 = new Product(null, "Macbook Pro", "Nam eleifend maximus tortor, at mollis.",
1250.0, "");
		Product p4 = new Product(null, "PC Gamer", "Donec aliquet odio ac rhoncus cursus.", 1200.0,
"");
		Product p5 = new Product(null, "Rails for Dummies", "Cras fringilla convallis sem vel faucibus.",
100.99, "");

		productRepository.saveAll(Arrays.asList(p1,p2,p3,p4,p5));

		p1.getCategories().add(cat2);
		p2.getCategories().add(cat1);
		p2.getCategories().add(cat3);
		p3.getCategories().add(cat3);
		p4.getCategories().add(cat3);
		p5.getCategories().add(cat2);

		productRepository.saveAll(Arrays.asList(p1,p2,p3,p4,p5));
```

<Pode testar>

---

20. **Crie a classe OrderItemPK, no pacote entities.pk, para servir de chave primária composta para a futura classe que será criada: OrderItem. OrderItemPK armazenará as duas chaves estrangeiras de product e order.**

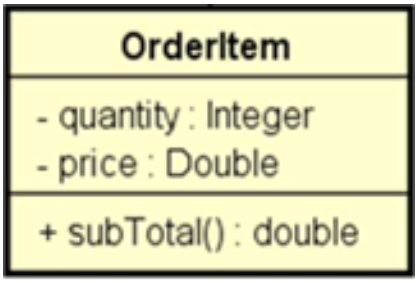
```
@Embeddable
public class OrderItemPK {

	@ManyToOne
	@JoinColumn(name = "order_id")
	private Order order;

	@ManyToOne
	@JoinColumn(name = "product_id")
	private Product product;

	//Getters and Setters

	//subTotal

	//hashCode and equals
}
```

Observações:

@Embeddable: Indica ao JPA que a classe atual não é um entidade para se gerar uma tabela, mas sim um dado que deve ser incorporado a uma entidade, seja como chave primária ou como qualquer outro dado composto.

@ManyToOne: Nesse caso, a anotação em questão está sendo usada para se estabelecer uma relação entre um dado que supostamente estaria na classe principal (OrderItem), ou seja, uma relação entre OrderItem, Order e Product.

21. Crie a classe OrderItem, no pacote entities, para servir de tabela de junção com dados extras no banco de dados para a relação entre as entidades Product e Order.



Basic entity checklist:

- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

```
@Entity
@Table(name = "tb_order_item")
public class OrderItem implements Serializable {

	private static final long serialVersionUID = 1L;

	@EmbeddedId
	private OrderItemPK id = new OrderItemPK();
	private Integer quantity;
	private Double price;

	public OrderItem() {

	}



	public OrderItem(Order order, Product product, Integer quantity, Double price) {
		id.setOrder(order);
		id.setProduct(product);
		this.quantity = quantity;
		this.price = price;
	}

	public Order getOrder() {
		return id.getOrder();
	}

	public void setOrder(Order order) {
		id.setOrder(order);
	}

	public Product getProduct() {
		return id.getProduct();
	}

	public void setProduct(Product product) {
		id.setProduct(product);
	}

	//Getters and Setters (menos do Id)

	//hashCode and equals
}
```

@EmbeddedId: Para declarar que o Id composto OrderItemPK é um dado composto.

@JsonIgnore: Necessário para que quando ocorrer a serialização de Product, que posteriormente irá

<Tem mais na próxima página!>

22. Edite a classe Order adicionando o seguinte atributo:

```
[...]
@OneToMany(mappedBy = "id.order")
private Set<OrderItem> items = new HashSet<>();
[...]
//getItems
[...]
```

Observações:

@OneToMany: Perceba que a anotação OneToMany vai buscar dentro do OrderItem seu id e depois acessar seu order. Por padrão, é importante ter em mente que todas as anotações fazem consulta dos seus respectivos atributos da classe através dos métodos get, ou seja, a serialização do Order, por exemplo, vai consultar o método getOrder de cada OrderItem de items, porém a relação entre o campo Order de "OrderItem" (OrderItemPK) e o atributo Id de Order será dada pelas anotações @ManyToOne em OrderItemPK e @OneToMany em Order (Essa informação será importante para entender o próximo passo).

---

23. Coloque a anotação @JsonIgnore acima do método getOrder da classe OrderItem. É importante destacar que o Jackson não serializa a classe Entity pelos atributos, mas sim pelos métodos get... O texto que vem depois do prefixo get e o retorno dos getters dão nome e valor aos atributos no arquivo JSON.

24. É necessário fazer com que o objeto Product venha com uma lista de Orders aos quais ele está associada. Nesse sentido, faz-se necessário terminar a relação entre OrderItemPK e Product e criar um método getOrders na classe dos produtos para que sejam exibidos os pedidos no JSON.

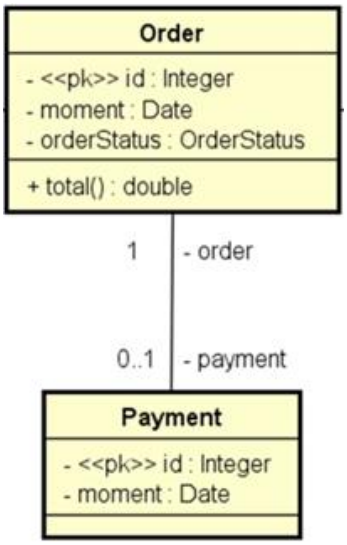
```
[...]
@OneToMany(mappedBy = "id.product")
private Set<OrderItem> items = new HashSet<>();
[...]
@JsonIgnore
public Set<Order> getOrders(){
	Set<Order> set = new HashSet<>();
	for(OrderItem x : items) {
		set.add(x.getOrder());
	}
	return set;
}
[...]
```

25. Adicione o seguinte código ao TestConfig:

```
	OrderItem oi1 = new OrderItem(o1, p1, 2, p1.getPrice());
	OrderItem oi2 = new OrderItem(o1, p3, 1, p3.getPrice());
	OrderItem oi3 = new OrderItem(o2, p3, 2, p3.getPrice());
	OrderItem oi4 = new OrderItem(o3, p5, 2, p5.getPrice());

	orderItemRepository.saveAll(Arrays.asList(oi1,oi2,oi3,oi4));
```

<Tem mais na próxima página!>

26. Crie a classe Payment, no pacote entities, para ser veiculada ao Order.



Basic entity checklist:

- **Basic attributes**
- **Associations (instantiate collections)**
- **Constructors**
- **Getters & Setters (collections: only get)**
- **hashCode & equals**
- **Serializable**

```
@Entity
@Table(name = "tb_payment")
public class Payment implements Serializable{

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private Instant moment;

	@JsonIgnore
	@OneToOne
	@MapsId
	private Order order;

	//Construtores

	//Getters and Setters

	//hashCode and equals

}
```

Observações:

@OneToOne: Indica ao JPA que a relação entre Order e Payment é de um para um.

@MapsId: Indica que o Payment recebe um Id igual ao do seu Order associado. Para cada Payment há um Order, forçando uma dependência forte entre os objetos tanto para o Banco de Dados quanto para o Java.

<Tem mais na próxima página!>

27. Adicione o atributo Payment ao Order e seus Getters and Setters:

```
[...]
@OneToOne(mappedBy = "order", cascade = CascadeType.ALL)
private Payment payment;
[...]
//getPayment
//setPayment
[...]
```

Observações:

Cascade = CascadeType.ALL: Essa variável da anotação OneToOne indica que todas as operações feitas com Order deverão ser igualmente realizadas com a respectiva entidade Payment no outro banco de dados, ou seja, UPDATE, DELETE, etc, serão realizadas em cascata com todo Payment relacionado a um Order que, nesse caso, é apenas um.

---

28. Insira o método getSubTotal na classe OrderItem:

```
public Double getSubTotal() {

        return price * quantity;

}
```

29. Insira o método getTotal na classe Order:

```
public Double getTotal() {
        double sum = 0.0;
        for(OrderItem x : items) {
                sum += x.getSubTotal();
        }
        return sum;
}
```

Finalizando

30. Adicione o método insert ao UserService:

```
public User insert(User obj) {

        return repository.save(obj);

}
```

31. Adicione o método insert ao UserResource:

```
@PostMapping
public ResponseEntity<User> insert(@RequestBody User obj){
        obj = service.insert(obj);
        URI uri =
ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(obj.getId()).toUri();
        return ResponseEntity.created(uri).body(obj);
}
```

Observações:

@PostMapping: Indica que o método atual responde a requisições de POST.

@RequestBody: Indica que o corpo da resposta do PostMapping deve ser desserializado como um objeto User.

ServletUriComponentsBuilder: Classe necessária para a construção da URI.

fromCurrentRequest(): Resgata a URI da requisição atual que, nesse caso, é localhost:8080/users.

Path("/{id}"): Adiciona o caminho "/{id}" ao final da URI atual.

buildAndExpand(): Substitui o valor de {id} pelo id do obj.

toUri(): Converte todo o processo para um objeto URI.

---

32. Adicione o método delete ao UserService:

```
public void delete(Long id) {
        repository.deleteById(id);
}
```

33. Adicione o método delete ao UserResource:

```
@DeleteMapping(value = "/{id}")
public ResponseEntity<Void> delete(@PathVariable Long id){
        service.delete(id);
        return ResponseEntity.noContent().build();
}
```

Observações:

@DeleteMapping: Indica que o método atual responde a requisições de DELETE.

@PathVariable: Indica que o corpo da URL digitada contém a variável que deve ser passada como parâmetro do método.

---

34. Adicione o método update ao UserService:

```
public User update(Long id, User obj) {
        User entity = repository.getReferenceById(id);
        updateData(entity, obj);
        return repository.save(entity);
}

private void updateData(User entity, User obj) {
        entity.setName(obj.getName());
        entity.setEmail(obj.getEmail());
        entity.setPhone(obj.getPhone());
}
```

Observações:

getReferenceById: Cria a instância de um objeto User mapeado com o banco de dados. Diferentemente do findViewById, getReferenceById não se dá o trabalho de buscar o objeto no banco, ele apenas faz referência a ele. Ou seja, cria um proxy (um objeto fantasma) que não contém imediatamente os dados reais do objeto. Uma observação importante, é que quando você faz referência aos atributos do objeto, é nesse momento que o JPA faz a consulta dos dados no banco de dados, ou seja, a instância é carregada por completo.

35. **Adicione o método update ao UserResource:**

```
	@PutMapping(value = "/{id}")
	public ResponseEntity<User> update(@PathVariable Long id, @RequestBody User obj){

		obj = service.update(id, obj);
		return ResponseEntity.ok().body(obj);

	}
```

Observações:

@PutMapping: Indica que o método atual responde a requisições de PUT.

---

36. **Crie a exceção ResourceNotFoundException, no pacote services.exception, para que a camada de serviços tenha sua própria exceção personalizada para o caso de não ser encontrado ao recurso solicitado do banco de dados.**

```
public class ResourceNotFoundException extends RuntimeException{

	private static final long serialVersionUID = 1L;

	public ResourceNotFoundException(Object id) {
		super("Resource not found. Id " + id);
	}
}
```

37. **Crie a classe StandardError, no pacote resource.exceptions para representar o corpo de uma exceção que será lançada como corpo da resposta HTTP, caso ocorra um erro.**

```
public class StandardError implements Serializable{

	private static final long serialVersionUID = 1L;

	private Instant timestamp;
	private Integer status;
	private String error;
	private String message;
	private String path;

	//Construtores

	//Getters and Setters

}
```

38. **Crie a classe ResourceExceptionHandler, no pacote resource.exceptions, para executar ações quando uma exceção for resgatada.**

```
@ControllerAdvice
public class ResourceExceptionHandler {

	@ExceptionHandler(ResourceNotFoundException.class)
	public ResponseEntity<StandardError> resourceNotFound(ResourceNotFoundException e,
HttpServletRequest request){
		String error = "Resource not found.";
		HttpStatus status = HttpStatus.NOT_FOUND;
		StandardError err = new StandardError(Instant.now(), status.value(), error, e.getMessage(),
request.getRequestURI());
		return ResponseEntity.status(status).body(err);
	}
}
```

Observações:

@ControllerAdvice: Serve para informar ao sistema que a classe em questão é um componente global que captura exceções de controllers ou restControllers.

@ExceptionHandler: Especifica o tipo de exceção que está sendo tratada pelo método a seguir.

HttpServletRequest: Trata-se de um objeto que representa a requisição HTTP feita pelo cliente. Dentro dele, pode-se resgatar o tipo de requisição que foi feita (POST, GET, PUT,...), o cabeçalho da requisição, o IP do cliente, o corpo da requisição, parâmetros da URL, etc.

HttpStatus.NOT_FOUND: Código de erro da requisição padrão para requisições que buscam algo que não foi possível encontrar

---

39. Altere o retorno da função findById de UserService para o seguinte comando:

```
	public User findById(Long id) {

		Optional<User> obj = repository.findById(id);
		return obj.orElseThrow(() -> new ResourceNotFoundException(id));

	}
```

Observações:

orElseThrow: Método da classe Optional que lança um objeto User (nesse caso) ou quando for nulo, uma exceção.

---

40. Faça o tratamento de exceção para o método delete em UserService para o caso de não ter sido encontrado o cadastro de um determinado usuário ao se realizar a requisição.

```
	public void delete(Long id) {
		try {
			repository.deleteById(id);
		}catch(EmptyResultDataAccessException e) {
			throw new ResourceNotFoundException(id);
		}catch(DataIntegrityViolationException e) {
			throw new DatabaseException(e.getMessage());
		}

	}
```

Observações:

EmptyResultDataAccessException: Exceção lançada quando o Id de usuário não retorna nenhum registro do banco de dados.

DataIntegrityViolationException: Exceção lançada quando tenta-se deletar um usuário que tem relação com registros de outras tabelas que, nesse caso, trata-se de um usuário que tem pedidos associados a ele.

DatabaseException: Exceção personalizada que será criada adiante.

<Tem mais na próxima página!>

41. **Crie a exceção DatabaseException, no pacote services.exceptions, para representar os erros de consistência com o banco de dados.**

```
public class DatabaseException extends RuntimeException{

	private static final long serialVersionUID = 1L;

	public DatabaseException(String msg) {
		super(msg);
	}

}
```

42. **Adicione um ExceptionHandler ao ResourceExceptionHandler para capturar a DatabaseException:**

```
	@ExceptionHandler(DatabaseException.class)
	public ResponseEntity<StandardError> database(DatabaseException e, HttpServletRequest request){

		String error = "Database error";
		HttpStatus status = HttpStatus.BAD_REQUEST;
		StandardError err = new StandardError(Instant.now(), status.value(), error, e.getMessage(),
request.getRequestURI());

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

	}
```

Observações:

HttpStatus.BAD_REQUEST: Quando a requisição é reconhecida, porém os dados informados estão inválidos ou faltam dados.

---

43. **Por fim, trate o erro no método update de não encontrar um usuário por um determinado Id ao qual não corresponda nenhum registro no banco de dados.**

```
	public User update(Long id, User obj) {
		try {
			User entity = repository.getReferenceById(id);
			updateData(entity, obj);
			return repository.save(entity);
		}catch(EntityNotFoundException e) {
			throw new ResourceNotFoundException(id);
		}
	}
```

Observações:

EntityNotFoundException: A exceção que é lançada quando o Id inserido na url da requisição update de UserResource não retorna um objeto usuário.

<Tem mais na próxima página!>

44. Atualize o TestConfig para que ele fique da seguinte forma:

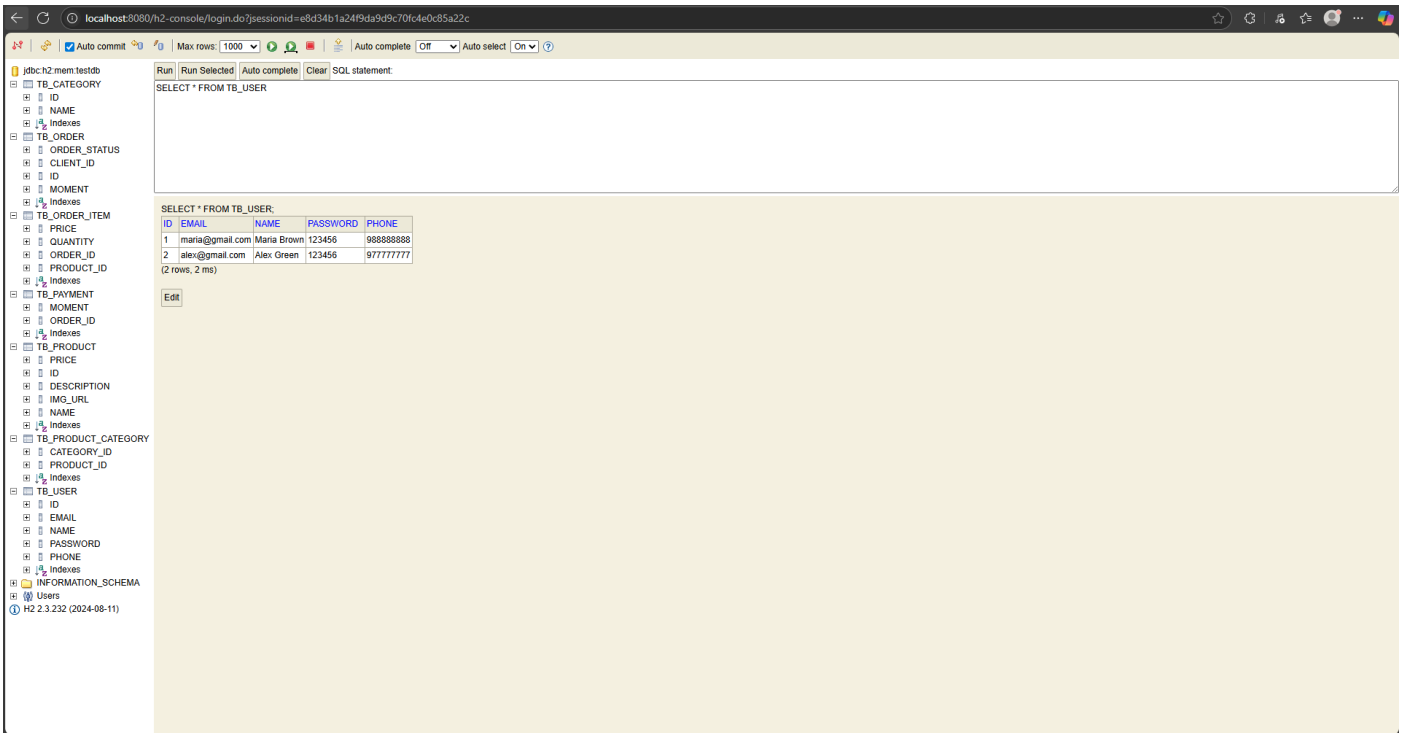
```
	@Override
	public void run(String... args) throws Exception {

		Category cat1 = new Category(null, "Electronics");
		Category cat2 = new Category(null, "Books");
		Category cat3 = new Category(null, "Computers");

		categoryRepository.saveAll(Arrays.asList(cat1, cat2, cat3));

		Product p1 = new Product(null, "The Lord of the Rings", "Lorem ipsum dolor sit amet, consectetur.", 90.5, "");
		Product p2 = new Product(null, "Smart TV", "Nulla eu imperdiet purus. Maecenas ante.", 2190.0, "");
		Product p3 = new Product(null, "Macbook Pro", "Nam eleifend maximus tortor, at mollis.", 1250.0, "");
		Product p4 = new Product(null, "PC Gamer", "Donec aliquet odio ac rhoncus cursus.", 1200.0, "");
		Product p5 = new Product(null, "Rails for Dummies", "Cras fringilla convallis sem vel faucibus.", 100.99, "");

		productRepository.saveAll(Arrays.asList(p1,p2,p3,p4,p5));

		p1.getCategories().add(cat2);
		p2.getCategories().add(cat1);
		p2.getCategories().add(cat3);
		p3.getCategories().add(cat3);
		p4.getCategories().add(cat3);
		p5.getCategories().add(cat2);

		productRepository.saveAll(Arrays.asList(p1,p2,p3,p4,p5));

		User u1 = new User(null, "Maria Brown", "maria@gmail.com", "988888888", "123456");
		User u2 = new User(null, "Alex Green", "alex@gmail.com", "977777777", "123456");

		Order o1 = new Order(null, Instant.parse("2019-06-20T19:53:07Z"), u1, OrderStatus.PAID);
		Order o2 = new Order(null, Instant.parse("2019-07-21T03:42:10Z"), u2, OrderStatus.WAITING_PAYMENT);
		Order o3 = new Order(null, Instant.parse("2019-07-22T15:21:22Z"), u1, OrderStatus.WAITING_PAYMENT);

		userRepository.saveAll(Arrays.asList(u1, u2));
		orderRepository.saveAll(Arrays.asList(o1,o2,o3));

		OrderItem oi1 = new OrderItem(o1, p1, 2, p1.getPrice());
		OrderItem oi2 = new OrderItem(o1, p3, 1, p3.getPrice());
		OrderItem oi3 = new OrderItem(o2, p3, 2, p3.getPrice());
		OrderItem oi4 = new OrderItem(o3, p5, 2, p5.getPrice());

		orderItemRepository.saveAll(Arrays.asList(oi1,oi2,oi3,oi4));

		Payment pay1 = new Payment(null, Instant.parse("2019-06-20T20:53:07Z"), o1);
		o1.setPayment(pay1);

		orderRepository.save(o1);
	}
```

Observações:

Repare que ao salvar um pedido, o pagamento é salvo automaticamente, por isso não é necessário criar um PaymentRepository.

<Tem mais na próxima página!>

Resultado



Um banco de dados completo, automatizado, gerado automaticamente, com restrições e consistências entre os dados. Além de um servidor web que recebe requisições HTTP do tipo GET, POST, UPDATE, PUT e DELETE e que envia e processa dados, serializando objetos JAVA para corpos JSON e desserializando mensagens para objetos JAVA.

Um projeto 100% comentado em uma documentação consistente a um passo-a-passo com observações sobre os temas mais difíceis e menos comuns abordados nos códigos.

Muito obrigado por acompanhar até e/ou explorar o meu material!

Observações Extras:

@Transient: Notação que não foi utilizada, mas que pode ser muito útil. Quando colocada acima de uma variável, ela indica ao JPA que essa variável deve ser desconsiderada, ou seja, não deve ter um campo correspondente no banco de dados.

@Embedded: O mesmo que @EmbeddedId, porém para declarar que a variável em questão é um dado composto em geral e não uma primary key.