



Universidade do Minho

UMinho

**Mestrado Engenharia Informática
Arquiteturas Aplicacionais
Sistemas Interativos Confiáveis
(2022/23)**

Solução Arquitetural



Diana Rodrigues
pg50320



Mariana Amorim
pg50623



Mariana Rodrigues
pg50622



Miguel Fernandes
pg50654

Braga, 22 de junho de 2023

Resumo

Neste documento serão apresentadas as abordagens seguidas e decisões tomadas na conceção da aplicação descrita no documento de requisitos.

Será apresentado um desenho da arquitetura, de forma a evidenciar as diferentes aplicações que compõe a aplicação final, mais concretamente as *frameworks* e tecnologias utilizadas e as suas diferentes componentes. Para além disso, é definido de que forma é feita a comunicação entre componentes dentro da mesma aplicação e entre as diferentes aplicações.

Serão também apresentados modelos PSM (Platform Specific Model), nomeadamente para *Hibernate* e *Spring*, para uma melhor compreensão sobre a persistência das entidades na base de dados e da divisão das componentes da aplicação *backend*, respetivamente.

Na parte da aplicação do *frontend* serão apresentados a estrutura dos componentes usados, nomeadamente *Vuetify* e *Vue.js* e os princípios usados para alcançar um nível de usabilidade satisfatório para os utilizadores do sistema.

Finalmente, será apresentada a metodologia escolhida para os testes de carga efetuados, bem como as ferramentas utilizadas, os resultados obtidos e uma análise sobre os mesmos.

Conteúdo

1 Solução Arquitetural	5
2 Spring Boot Backend App	6
2.1 Spring Security	6
2.2 Controller	7
2.3 Service	7
2.4 Model	7
2.5 Repository	7
2.6 Exceptions	8
3 Modelos PSM	9
3.1 Modelo PSM Hibernate	9
3.2 Anotações Hibernate	12
3.2.1 Company	12
3.2.2 SocialNetwork	12
3.2.3 User	12
3.2.4 Admin	12
3.2.5 Customer	13
3.2.6 Product	13
3.2.7 Image	13
3.2.8 TechnicalInfo	14
3.2.9 Category	14
3.2.10 SubCategory	14
3.2.11 Order	15
3.2.12 Item	15
3.2.13 OrderItem	15
3.3 Diagrama EER	15
3.4 Modelos PSM Spring	17
3.4.1 Componente <i>Company Info</i>	17
3.4.2 Componente <i>Authentication</i>	19
3.4.3 Componente <i>User</i>	21
3.4.4 Componente <i>Product</i>	23
3.4.5 Componente <i>Order</i>	25
4 Vue Frontend App	26
4.1 Facilidade de aprendizagem	27
4.2 Flexibilidade	31
4.3 Robustez	36
5 Testes	40
5.1 Teste de Carga	40

6 Conclusão **43**

7 Anexos **44**

Listas de Figuras

1.1	Solução Arquitetural	5
3.1	Modelo PSM Hibernate (I)	9
3.2	Modelo PSM Hibernate (II)	11
3.3	EER Diagram	16
3.4	Componente Company Info	18
3.5	Componente Authentication	20
3.6	Componente User	22
3.7	Componente Product	24
3.8	Componente Order	25
4.1	Mensagem de administrador removido com sucesso	27
4.2	Mensagem de erro ao tentar remover administrador	27
4.3	Página das encomendas	28
4.4	Página dos utilizadores	28
4.5	Página das informações da empresa	29
4.6	Barra de navegação de um utilizador não registado	29
4.7	Barra de navegação de um cliente	29
4.8	Barra de navegação de um cliente	29
4.9	Exemplo de procura por categoria	30
4.10	Exemplo de procura por categoria na página principal	31
4.11	Exemplo de tooltips usados	31
4.12	Página de um produto para ecrãs maiores	32
4.13	Página de um produto para ecrãs menores (1/2)	33
4.14	Página de um produto para ecrãs menores (2/2)	34
4.15	Página de um produto para ecrãs menores	35
4.16	Menu para mudar a linguagem	36
4.17	Barra de navegação (1/2)	36
4.18	Barra de navegação (2/2)	36
4.19	Formulário para editar informações da empresa	37
4.20	Paginação	37
4.21	Modal para confirmação da remoção de um administrador	38
4.22	Impedir a submissão de formulários com campos vazios	38
4.23	Mensagem de erro ao tentar introduzir NIF inválido	38
4.24	Calendário para escolher data de nascimento	39
4.25	Barra de progresso	39
5.1	Arquitetura <i>Cloud</i> da aplicação	41
5.2	Métricas para 50 utilizadores	41
5.3	Métricas para 100 utilizadores	41
5.4	Métricas para 200 utilizadores	42
5.5	Métricas para 400 utilizadores	42

5.6	Métricas para 500 utilizadores	42
7.1	Modelo PIM	44
7.2	Product Model	45
7.3	Product Controller	46
7.4	Product Repository	47
7.5	Product Service	48
7.6	Order Model	49
7.7	Order Controller	49
7.8	Order Repository	50
7.9	Order Repository	51

1. Solução Arquitetural

Para o desenvolvimento da aplicação, foram consideradas duas aplicações distintas: uma aplicação *backend* responsável por criar serviços que processam a lógica de negócios e por fazer toda a gestão da base de dados, bem como os acessos à mesma, e uma aplicação *frontend*.

A aplicação *Vue client-side* é a responsável por apresentar a interface aos utilizadores e fazer a integração com o *backend* por pedidos HTTP.

Quanto à aplicação *backend*, que é *server-side* foi utilizado o *Spring Boot*, persistindo os dados numa base de dados *MySQL*.

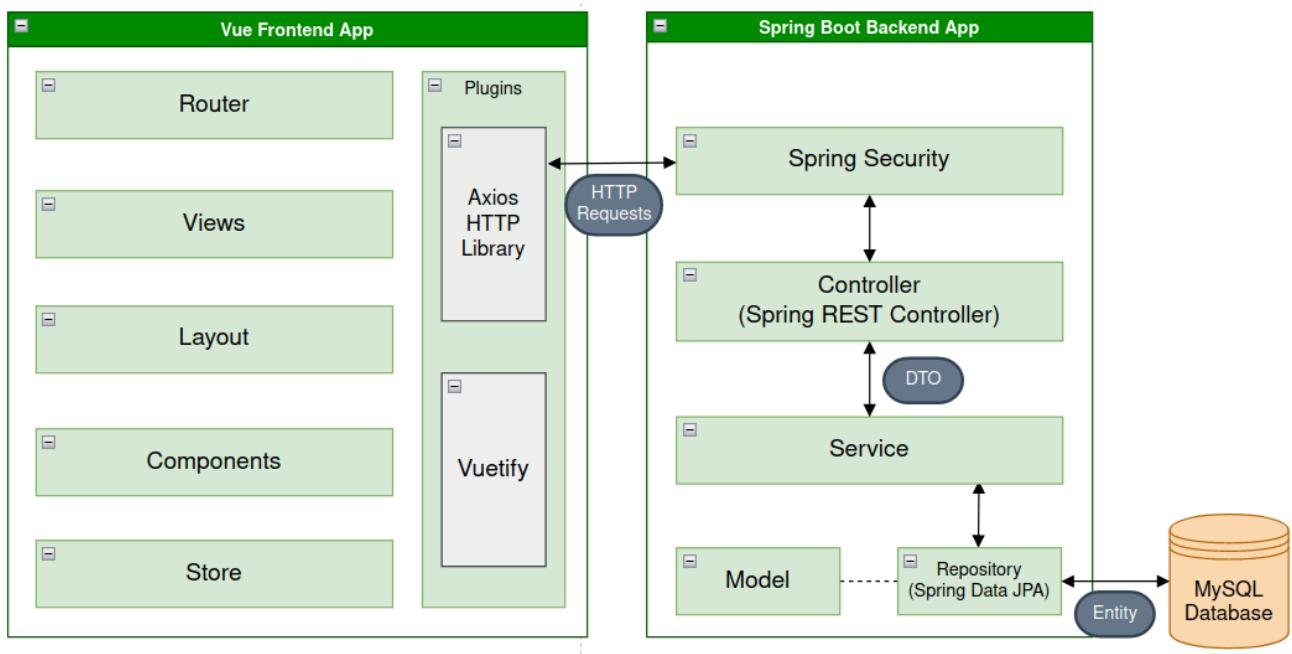


Figura 1.1: Solução Arquitetural.

2. Spring Boot Backend App

A aplicação backend foi desenvolvida com o suporte da *framework* Spring, mais concretamente a ferramenta *open-source* Spring Boot. Esta ferramenta permite a inclusão de todas as tecnologias Spring recorrendo aos *starters*, um conjunto de *dependency descriptors*. Para o desenvolvimento da aplicação foram considerados os seguintes *starters*:

- **spring-boot-starter-web** - *starter* para desenvolvimento de aplicações web, incluindo RESTful, usando Spring MVC. Usa o Tomcat como *default embedded container*.
- **spring-boot-starter-data-jpa** - *starter* para usar Spring Data JPA com Hibernate.
- **spring-boot-starter-security** - *starter* para uso de Spring Security.
- **spring-boot-starter-mail** - *starter* para usar Java Mail e a *framework* Spring de suporte ao envio de email.
- **spring-boot-starter-webflux** - *starter* para desenvolver aplicações WebFlux usando a *framework* Spring de suporte Reactive Web.

2.1 Spring Security

A *framework* Spring Security permite criar mecanismos de autenticação e autorização de uma maneira eficaz e customizável.

De maneira a cumprir os requisitos da aplicação, era necessário garantir que cada funcionalidade só deve ser implementada por utilizadores específicos, isto é, toda a lógica de gerir categorias, subcategorias, produtos e materiais deve ser exclusiva ao administrador.

No *Spring Security* é utilizado um mecanismo de *tokens* que permite tanto lidar com a parte de autenticação como de autorização. Neste *token* é encapsulado o email do utilizador, que é único e permite identificá-lo, a data de expiração e o *role* do utilizador.

Um utilizador recebe um *token* ao registar-se ou ao autenticar-se que depois é enviado em todos os seus pedidos, permitindo verificar se o *token* é válido, isto é, corresponde a um utilizador e não está expirado e com base no *role* verificar quais as funcionalidades que este tem acesso.

Para configurar a *framework* com base nos requisitos da aplicação é implementado um *filter chain* onde se trata de toda lógica de autorização com base no pedido. A aplicação desenvolvida conta com 3 tipos de utilizador:

- **Utilizador não registado** - consegue navegar pela aplicação e visualizar todas as categorias, subcategorias e respetivos produtos.
- **Cliente** - utilizador registado com acesso a todas as funcionalidades do utilizador não registado e consegue ainda realizar encomendas e fazer *reviews* a produtos.

- **Administrador** - utilizador registado com acesso a todas as funcionalidades do utilizador não registado e ainda toda a parte de gestão da aplicação, isto é, criação/remoção/atualização de categorias, subcategorias, materiais, produtos e administradores.

Com base nestes tipos de utilizadores, no caminho do pedido para o *Rest Controller* e no *token* do utilizador é implementado o *filter chain*.

Neste componente foi necessário ainda implementar algumas configurações adicionais para este estar preparado para lidar com o *CORS*, mecanismo utilizado na parte do *frontend*.

2.2 Controller

Os *Controllers* é exposta para fora a *API* criada. São recebidos pedidos HTTP e com base no seu caminho é feito o encaminhamento para o método correto. Numa aplicação Spring é utilizada a anotação *RestController* para especificar que se trata de uma classe responsável por lidar com pedidos. Foram criados diferentes *Controllers* na aplicação com o intuito de aumentar a legibilidade do código, sendo esta divisão abordada na secção 3.4.

Foram criados *Data Transfer Objects*(DTO) para facilitar a troca de dados entre diferentes componentes da aplicação. Em vez de os atributos serem passados individualmente, é criada uma classe que permite otimizar a comunicação. Além disto, os DTO's são reutilizáveis, permitindo evitar replicação de código e para além disso facilitam a manutenção e expansão da aplicação, dado que, as alterações necessárias são feitas na classe criada e não nos diferentes *endpoints* de comunicação.

2.3 Service

Nos *Services* é implementada toda a lógica de negócio da aplicação. Os métodos criados nestes componentes refletem todos os *use cases* especificados. Foram criados diversos *Services*, com uma lógica de divisão equivalente à utilizada nos *Controllers*.

A comunicação entre os *Controllers* e os *Services* é feita com DTO's.

2.4 Model

No *Model* está especificado o domínio da aplicação. Nestas classes foram utilizadas anotações que permitiram indicar como os objetos iriam ser persistidos na base dados, isto é, a relação entre objetos, os mecanismos de herança, entre outros.

2.5 Repository

A comunicação com a base de dados é feita nos *Repositories*, sendo utilizado um mecanismo do Spring, os *JPA Repository*, que permite simplificar a implementação, não sendo necessária a utilização de DAO's.

Para utilizar os *JPA Repositories* basta apenas criar uma interface que estende o *JpaRepository*, fornecendo automaticamente métodos CRUD para acesso aos dados. Além dos métodos fornecidos é possível ainda adicionar *queries* mais específicas à base de dados, que envolvam por exemplo outras entidades, de uma maneira simples e intuitiva.

A interface *JpaRepository* precisa de receber a entidade que vai ser acedida, isto é, uma classe do modelo e o tipo da chave primária dessa entidade.

2.6 Exceptions

Para lidar com os *Run Time errors* foram criadas um conjunto de *Exceptions* específicas para o tipo de erro, por exemplo *UserNotFoundException*, *ProductNotFoundException*, entre outras... Com isto, é possível fornecer ao *frontend* erros claros, de maneira que seja percepível a razão deste.

3. Modelos PSM

3.1 Modelo PSM Hibernate

O modelo PSM Hibernate foi construído com base no modelo abstrato PIM (ver figura 7.1). O código final utiliza anotações, sendo que, nesta secção serão explicadas algumas das decisões tomadas. As seguintes figuras ilustram o modelo *PSM Hibernate* desenvolvido.

A figura 3.1 ilustra as entidades relacionadas com a informação da empresa que devem ser persistidas. A relação entre a entidade *Company* e *SocialNetwork* é bidirecional e representa a única relação que ambas as entidades estabelecem entre o conjunto de todas as entidades.

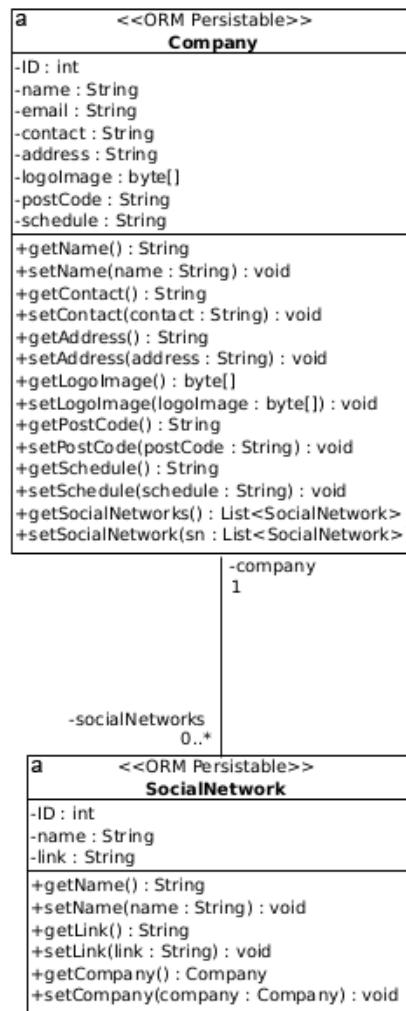


Figura 3.1: Modelo PSM Hibernate (I).

Seguindo a mesma estratégia utilizada nas entidades mencionadas anteriormente, todas as entidades que deveriam ser persistidas foram marcadas como *ORM Persistable*, foi adicionada a variável de instância ID e adicionados os métodos *get* e *set* para todas as variáveis de instância.

Relativamente às relações entre as entidades, foram definidas as multiplicidades e visibilidade (sempre *private*), bem como a naveabilidade das relações, dando especial atenção à definição da direção de cada relação.

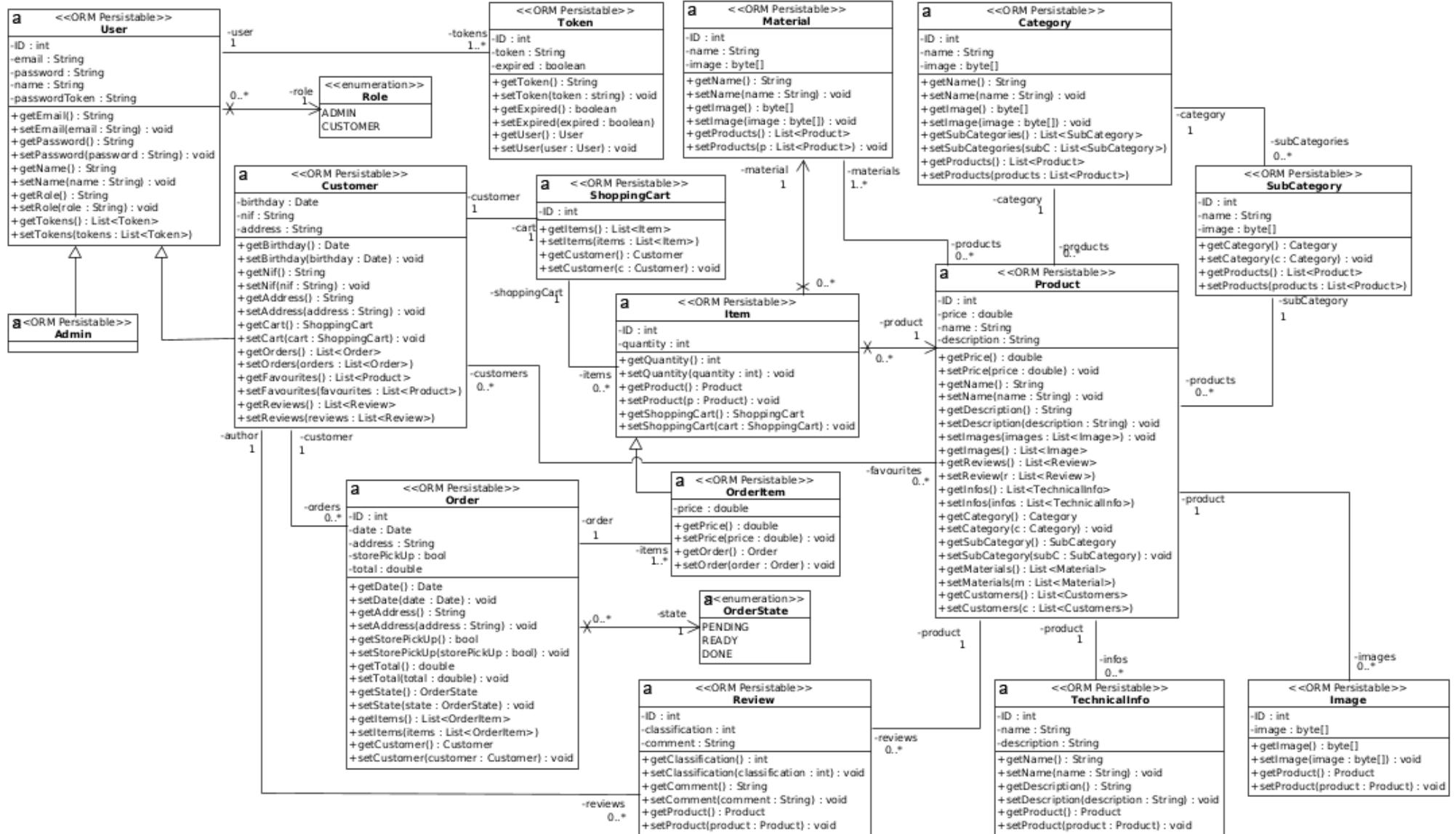


Figura 3.2: Modelo PSM Hibernate (II).

3.2 Anotações Hibernate

3.2.1 Company

Para a entidade *Company* foram usadas as seguintes anotações:

- **@Entity** na classe Company.
- **@Id** e **@GeneratedValue** para a variável ID.
- **@Column(nullable=false)** para as variáveis name, email, contact, address, postCode e schedule.
- **@Lob** e **@Column(length=20971520)** para a variável logoImage, de forma a assegurar o correto armazenamento de objetos maiores como as imagens.
- **@OneToMany(cascade = CascadeType.ALL, mappedBy="company")** para a variável socialNetworks, indicando que todas as alterações efetuadas na *Company* devem ser propagadas para as *SocialNetwork*.

3.2.2 SocialNetwork

Para a entidade *SocialNetwork* foram usadas as seguintes anotações:

- **@Entity** na classe SocialNetwork.
- **@Id** e **@GeneratedValue** para a variável ID.
- **@Column(nullable=false)** para as variáveis name, link e company.
- **@ManyToOne** para a variável company, estabelecendo a relação bidirecional.
- **@JoinColumn(name = "company_id")** para a variável company, indicando que não deve ser criada uma nova tabela para a relação, mas sim acrescentada uma coluna com o nome "company_id" na tabela as redes sociais.

3.2.3 User

Para a entidade *User* foram usadas as seguintes anotações:

- **@Entity, @Inheritance(strategy = InheritanceType.SINGLE_TABLE)** e **@DiscriminatorColumn(name = "usertype", discriminatorType = DiscriminatorType.STRING)** na classe User, de forma a persistir os dois tipos de utilizadores numa só tabela.
- **@Id** e **@GeneratedValue** para a variável ID.
- **@Column(nullable=false)** para as variáveis email, password e name.
- **@Column(unique=true)** para a variável email, garantido que é único para todos os utilizadores.
- **@OneToMany(mappedBy = "user")** para a variável tokens.
- **@Enumerated(EnumType.STRING)** para a variável role.

3.2.4 Admin

Para a entidade *Admin* foram usadas as seguintes anotações:

- **@Entity** e **@DiscriminatorValue("Admin")** para a classe Admin.

3.2.5 Customer

Para a entidade *Customer* foram usadas as seguintes anotações:

- **@Entity** e **@DiscriminatorValue("Customer")** para a classe Customer.
- **@Column(unique = true)** para a variável nif.
- **@OneToMany(cascade = CascadeType.ALL, mappedBy = "author")** para a variável reviews.
- **@OneToOne(cascade = CascadeType.ALL)** e **@JoinColumn(name = "cart_id")** para a variável cart, garantindo que as alterações no utilizador são propagadas para o seu carrinho de compras e que é adicionada uma coluna à tabela do utilizador para o "cart_id".
- **@OneToMany(cascade = CascadeType.ALL, mappedBy = "customer")** para a variável orders.
- **@ManyToMany** e **@JoinTable(name = "customer_favourites", joinColumns = @JoinColumn(name = "customer_id"), inverseJoinColumns = @JoinColumn(name = "product_id"))** para a variável favourites, definindo o nome da nova tabela e as suas colunas com base nas entidades que estabelecem essa relação de muitos para muitos.

3.2.6 Product

Para a entidade *Product* foram usadas as seguintes anotações:

- **@Entity** para a classe Product.
- **@Id** e **@GeneratedValue** para a variável ID.
- **@Column(nullable = false)** para as variáveis price e name.
- **@ManyToMany** para a variável materials.
- **@OneToMany(cascade = CascadeType.ALL, mappedBy = "product")** para as variáveis infos, reviews e images.
- **@ManyToOne** para as variáveis category e subCategory.
- **@JoinColumn(name = "category_id")** para a variável category.
- **@JoinColumn(name = "subcategory_id")** para a variável subCategory.
- **@ManyToMany(mappedBy = "favourites")** para a variável customers.

3.2.7 Image

Para a entidade *Image* foram usadas as seguintes anotações:

- **@Entity** para a classe Image.
- **@Table(uniqueConstraints = @UniqueConstraint(columnNames = "image", "product_id"))** para a classe Image, garantindo que cada imagem está associada a um produto apenas uma vez.
- **@Id** e **@GeneratedValue** para a variável ID.

- `@Lob` e `@Column(nullable = false, length = 20971520)` para image.
- `@ManyToOne` e `@JoinColumn(name = "product_id", nullable = false)` para product.

3.2.8 TechnicalInfo

Para a entidade `TechnicalInfo` foram usadas as seguintes anotações:

- `@Entity` para a classe `TechnicalInfo`.
- `@Table(uniqueConstraints = @UniqueConstraint(columnNames = "name", "product_id"))` para a classe `TechnicalInfo`, garantindo que cada par (tipo de informação, produto) é único.
- `@Id` e `@GeneratedValue` para a variável ID.
- `@Column(nullable = false)` para as variáveis name e product.
- `@ManyToOne` e `@JoinColumn(name = "product_id")` para product.

3.2.9 Category

Para a entidade `Category` foram usadas as seguintes anotações:

- `@Entity` para a classe `Category`.
- `@Id` e `@GeneratedValue` para a variável ID.
- `@Column(unique = true, nullable = false)` para a variáveis name.
- `@Lob` e `@Column(length = 20971520)` para image.
- `@OneToMany(cascade = CascadeType.ALL, mappedBy = "category")` para subCategories e products.

3.2.10 SubCategory

Para a entidade `SubCategory` foram usadas as seguintes anotações:

- `@Entity` para a classe `SubCategory`.
- `@Table(uniqueConstraints = @UniqueConstraint(columnNames = "id", "category_id"))` para a classe `SubCategory`, definindo que cada par subCaterogy e category deve ser único, ou seja, uma subcategoria só pode ser associada a uma categoria.
- `@Id` e `@GeneratedValue` para a variável ID.
- `@Lob` e `@Column(length = 20971520)` para image.
- `@ManyToOne` e `@JoinColumn(name = "category_id", nullable = false)` para category.
- `@OneToMany(mappedBy = "subCategory", cascade = CascadeType.PERSIST, CascadeType.MERGE, CascadeType.DETACH, CascadeType.REFRESH)` para products, definindo que todas as alterações nas subcategorias devem ser propagadas para os produtos, exceto a operação REMOVE, ou seja, quando uma subcategoria é removida, os seus produtos não são eliminados, apenas ficam associados à uma categoria.

3.2.11 Order

Para a entidade *Order* foram usadas as seguintes anotações:

- **@Entity** e **@Table(name = "_order")** para a classe Order, definindo o nome da tabela como `_order` para evitar conflitos com a palavra reservada ORDER do MySQL.
- **@Id** e **@GeneratedValue** para a variável ID.
- **@Column(nullable = false)** para as variáveis date, address, storePickUp, total, state e customer.
- **@OneToMany(cascade = CascadeType.ALL, mappedBy = "order")** para a variável items.
- **@Enumerated(EnumType.ORDINAL)** para a variável state.
- **@ManyToOne e @JoinColumn(name = "customer_id")** para a variável customer.

3.2.12 Item

Para a entidade *Item* foram usadas as seguintes anotações:

- **@Entity, @Inheritance(strategy = InheritanceType.SINGLE_TABLE), @DiscriminatorColumn(name = "itemtype", discriminatorType = DiscriminatorType.STRING) e @DiscriminatorValue("item")** para a classe Item, de forma a persistir os dois tipos de itens numa só tabela.
- **@Id e @GeneratedValue** para a variável ID.
- **@Column(nullable = false)** para as variáveis quantity, material e product.
- **@ManyToOne** para material, product e shoppingCart.
- **@JoinColumn(name = "material_id")** para material.
- **@JoinColumn(name = "product_id")** para product.
- **@JoinColumn(name = "cart_id")** para shoppingCart.

3.2.13 OrderItem

Para a entidade *Item* foram usadas as seguintes anotações:

- **@Entity e @DiscriminatorValue("orderitem")** para a classe OrderItem.
- **@Id e @GeneratedValue** para a variável ID.
- **@ManyToOne** para order.
- **@JoinColumn(name = "order_id")** para order.

Nas classes não mencionadas, são utilizadas apenas variações de todas as anotações acima definidas.

3.3 Diagrama EER

As anotações definidas na secção anterior permitiram gerar a base de dados que apresentamos recorrendo ao seguinte diagrama:

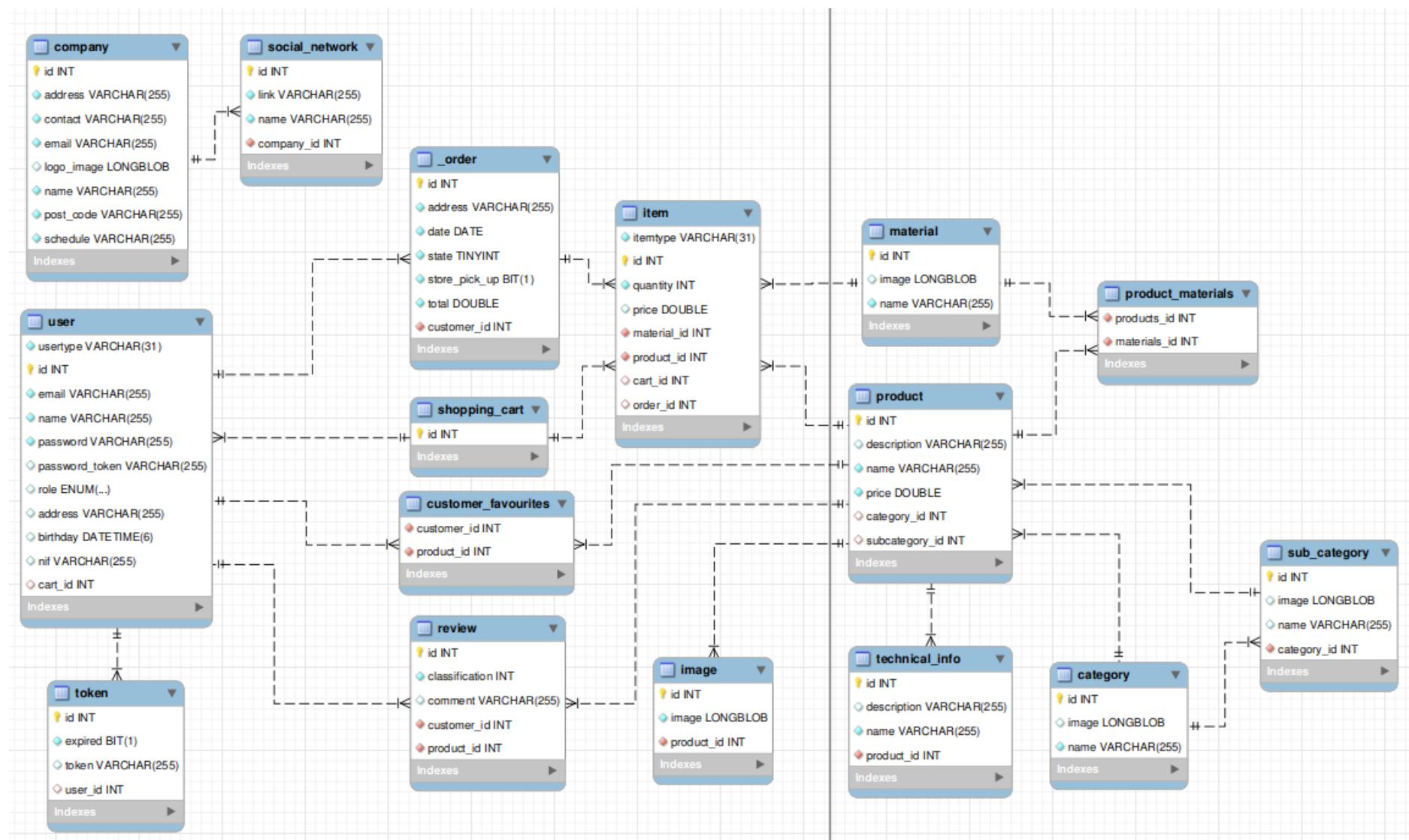


Figura 3.3: EER Diagram.

3.4 Modelos PSM Spring

Os modelos PSM Spring são essenciais para definir a divisão de componentes de forma a evitar dependências desnecessárias, além disso foram definidas as classes necessárias, bem como os seus métodos.

A divisão dos componentes foi feita com o objetivo de separar partes independentes, com esse objetivo foram criados 5 componentes, que serão abordados ao pormenor de seguida.

3.4.1 Componente *Company Info*

O componente *Company Info* é responsável pela informação da *company*, nomeadamente todas as operações CRUD sobre esta.

O *InfoController* recebe pedidos para obter informações concretas da *Company*, que estarão disponíveis para todos os utilizadores. Os pedidos de atualização são exclusivos ao administrador.

Na lógica de negócio implementada no *InfoService* é necessário garantir que existe apenas uma única empresa. Assim sendo, o grupo decidiu tirar proveito do padrão de criação Singleton para garantir o cumprimento desta restrição. Para isso, foi acrescentada a variável de instância *instance* à classe *Company*, bem como o método estático *getInstance()*. Antes de adicionar uma empresa é invocado esse método e uma empresa só é adicionada caso o seu retorno seja *null*.

No *Model* são definidas as entidades referentes à *Company* e às suas *Social Networks*, tendo nos *Repositories* as interfaces responsáveis por persisti-las e realizar *queries* sobre estas.

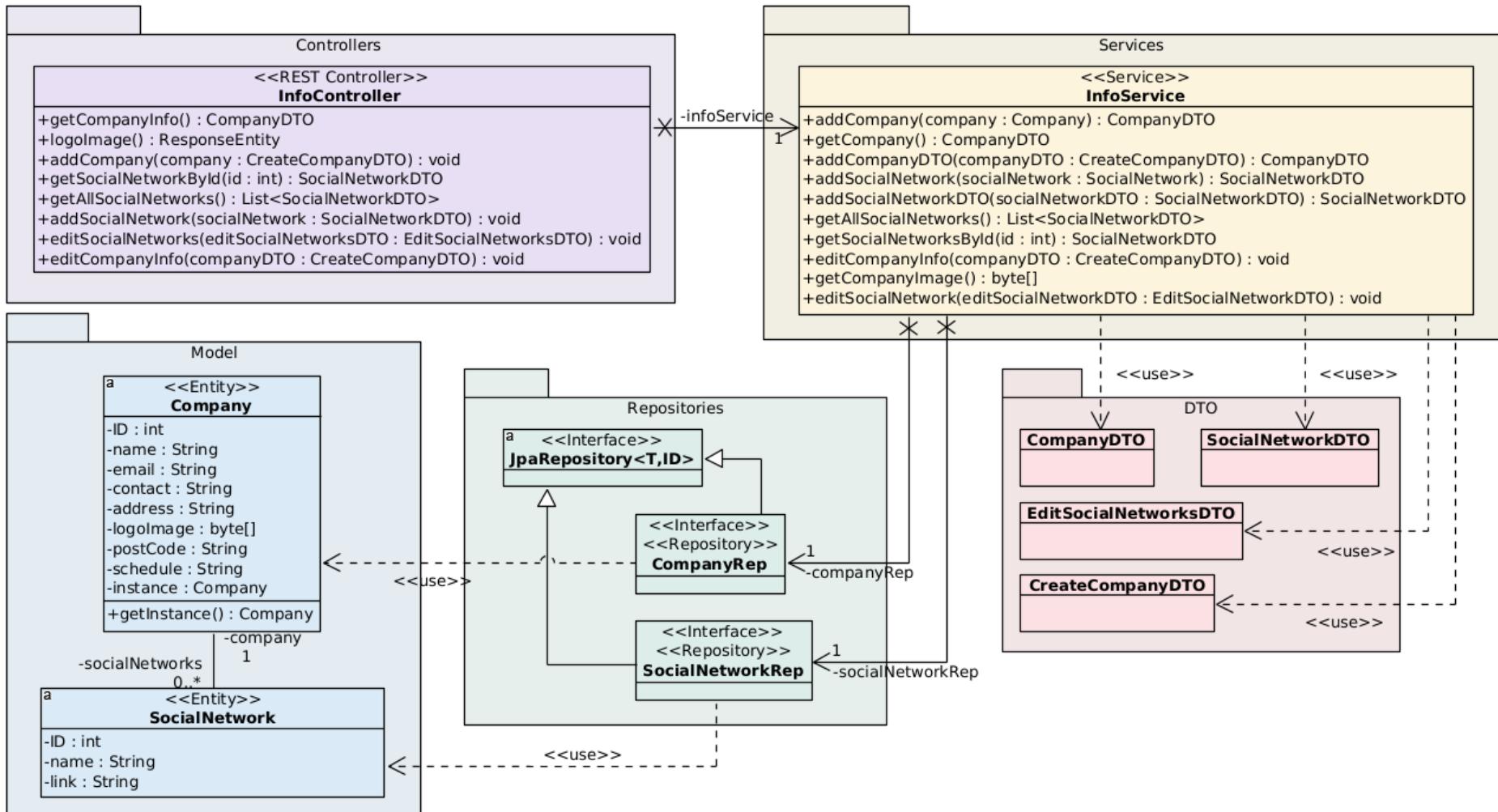


Figura 3.4: Componente Company Info

3.4.2 Componente *Authentication*

Neste componente é onde está implementada toda a lógica de autenticação do utilizador. Tal como referido anteriormente, a autenticação é feita com base em *tokens*, estes são recebidos aquando do registo ou login de um utilizador.

A criação do *token* gera uma chave encriptada que reflete o email do utilizador, de maneira a associar um *token* ao seu dono, a *role* do utilizador para tratar da autorização nos pedidos posteriores e uma data de expiração.

O registo presente no *AuthenticationController* é apenas para clientes, dado que o registo de administradores têm de ser feito por um administrador autenticado. O pedido de *logout* elimina o atual *token* do utilizador.

Na parte das configurações é onde está inserida a lógica de segurança do Spring, todos os pedidos recebidos são passadas por uma *filter chain* para tratar da parte de autorização com base no pedido feito e no tipo de utilizador.

Quanto às entidades persistidas, existe uma superclasse *User* com informação comum entre o *Admin* e *Customer*. Cada *User* terá associado a si uma lista de *tokens* que indicam se estão válidos.

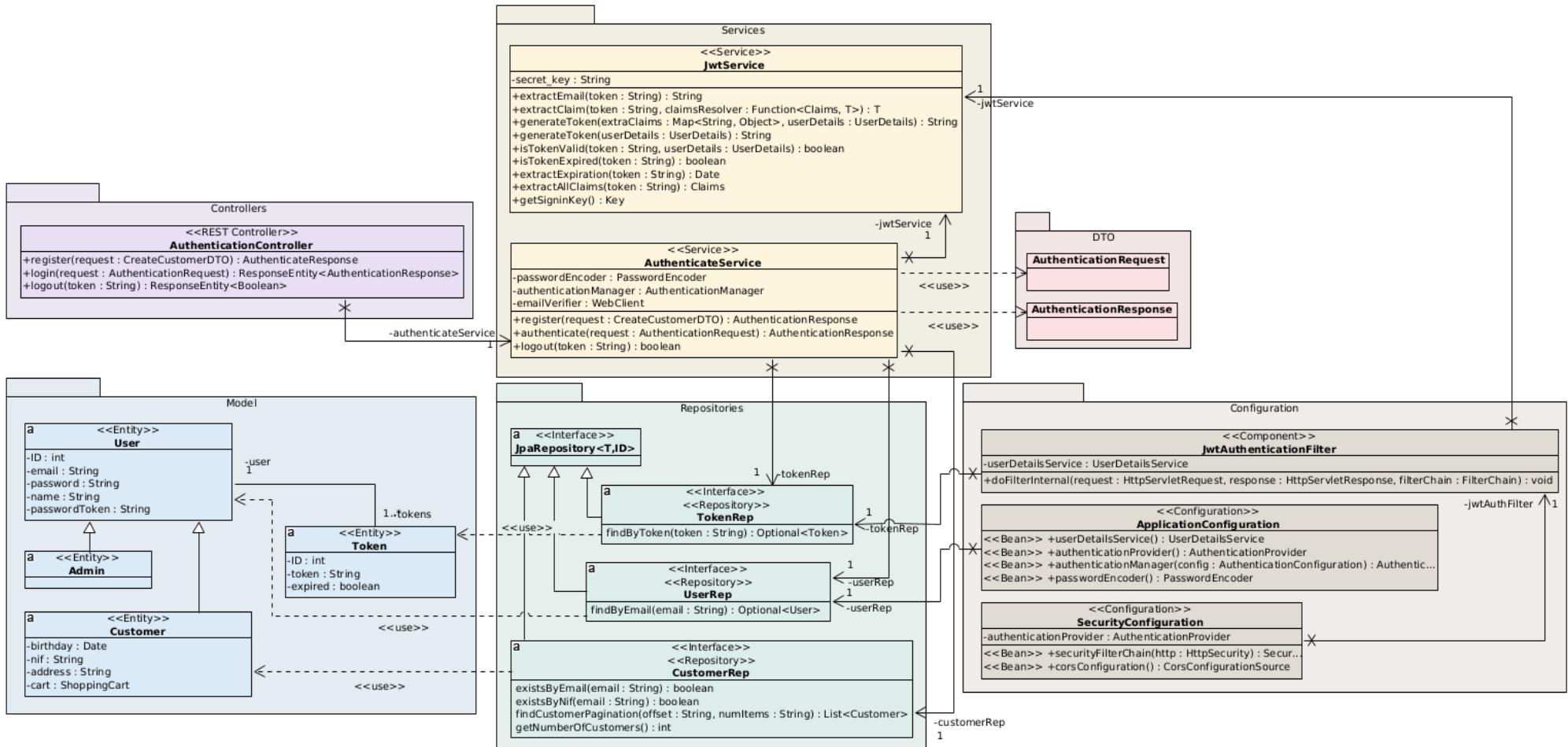


Figura 3.5: Componente Authentication

3.4.3 Componente *User*

A lógica de ambos utilizadores está presente no Componente *User*, isto é, todos os pedidos referentes a obter/atualizar/remover informação dos utilizadores. Sendo que os pedidos de administrador e cliente necessitam de autorizações diferentes esta divisão facilita na distinção dos pedidos.

Nesta componente foi implementado uma lógica de *Publisher/Subscriber*, maneira mais próxima de implementar o padrão *Observer* no Spring. Os *Customers* são os *Subscribers* na aplicação, estes são os que vão reagir a eventos do seu interesse, esses eventos são do tipo *Email Event* e desencadeiam o envio de um email. O envio do email é feito pelo *Notification Service*.

Foi feita ainda a integração neste componente com uma API externa, esta será responsável por verificar que os email recebidos no registo são válidos. Para integrar esta API externa, foi adicionado uma configuração *WebClientConfiguration* com o método *emailVerifier* onde é definido o *url* externo e os *headers* necessários para realizar os pedidos.

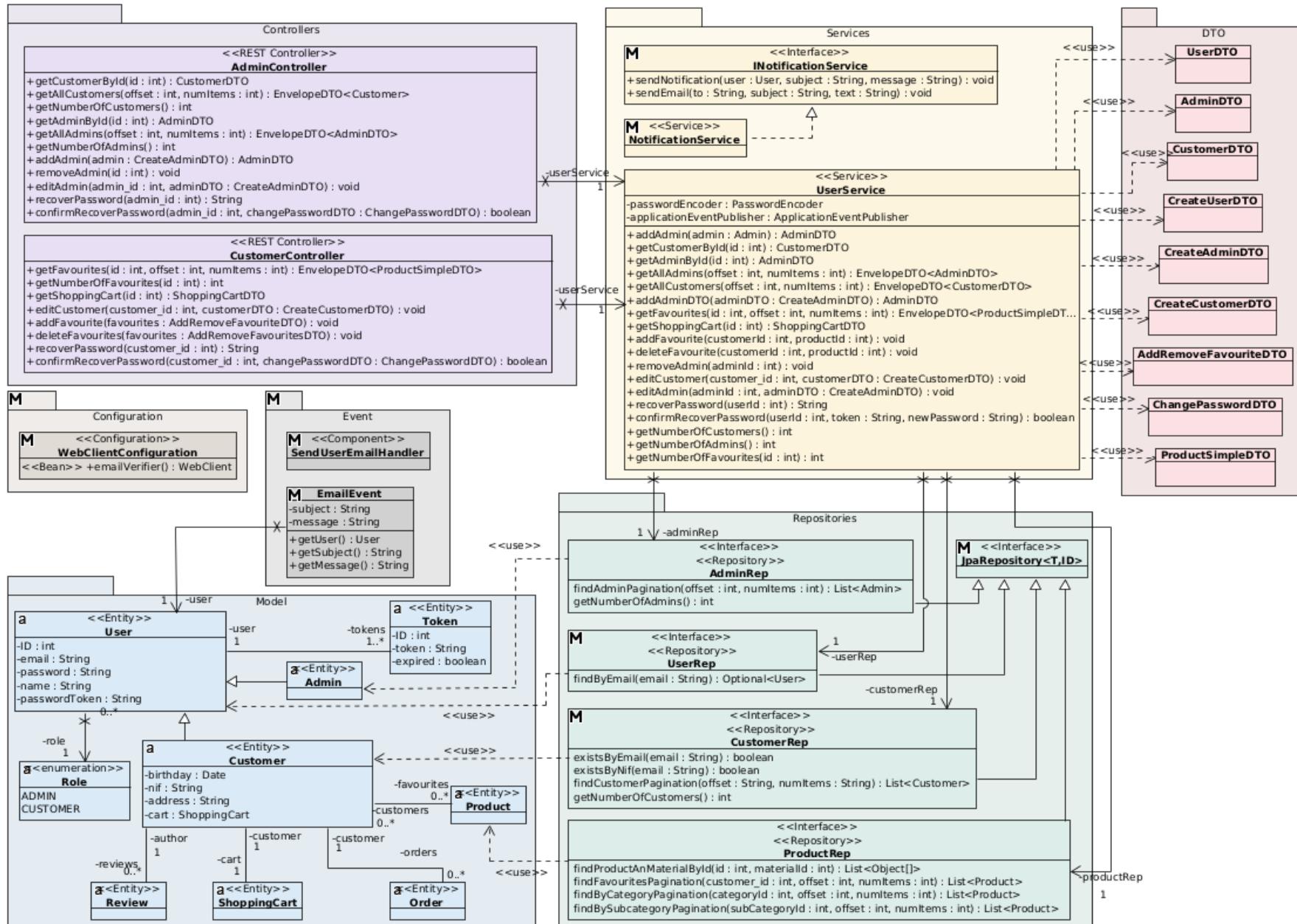


Figura 3.6: Componente User

3.4.4 Componente *Product*

Na componente *Product* estão as entidades relacionadas com os produtos, ou seja, as categorias, subcategorias e materiais. O *ProductController* é acedido por todos os utilizadores, sendo que os clientes registados e não registados irão fazer, maioritariamente pedidos de leitura para obter informações sobre os produtos e têm a opção de realizar *reviews* aos produtos. Já o administrador é responsável por adicionar/remover/atualizar todas estas entidades.

No *ProductService* é injetada a dependência *ApplicationEventPublisher* que é a interface utilizada para gerar eventos. Quando um produto altera o seu preço é necessário informar todos os clientes que o têm como favorito, dessa forma é criado um *EmailEvent* que leva os campos da notificação, este evento será posteriormente tratado pelos clientes interessados, tal como explicado na Componente *User*.

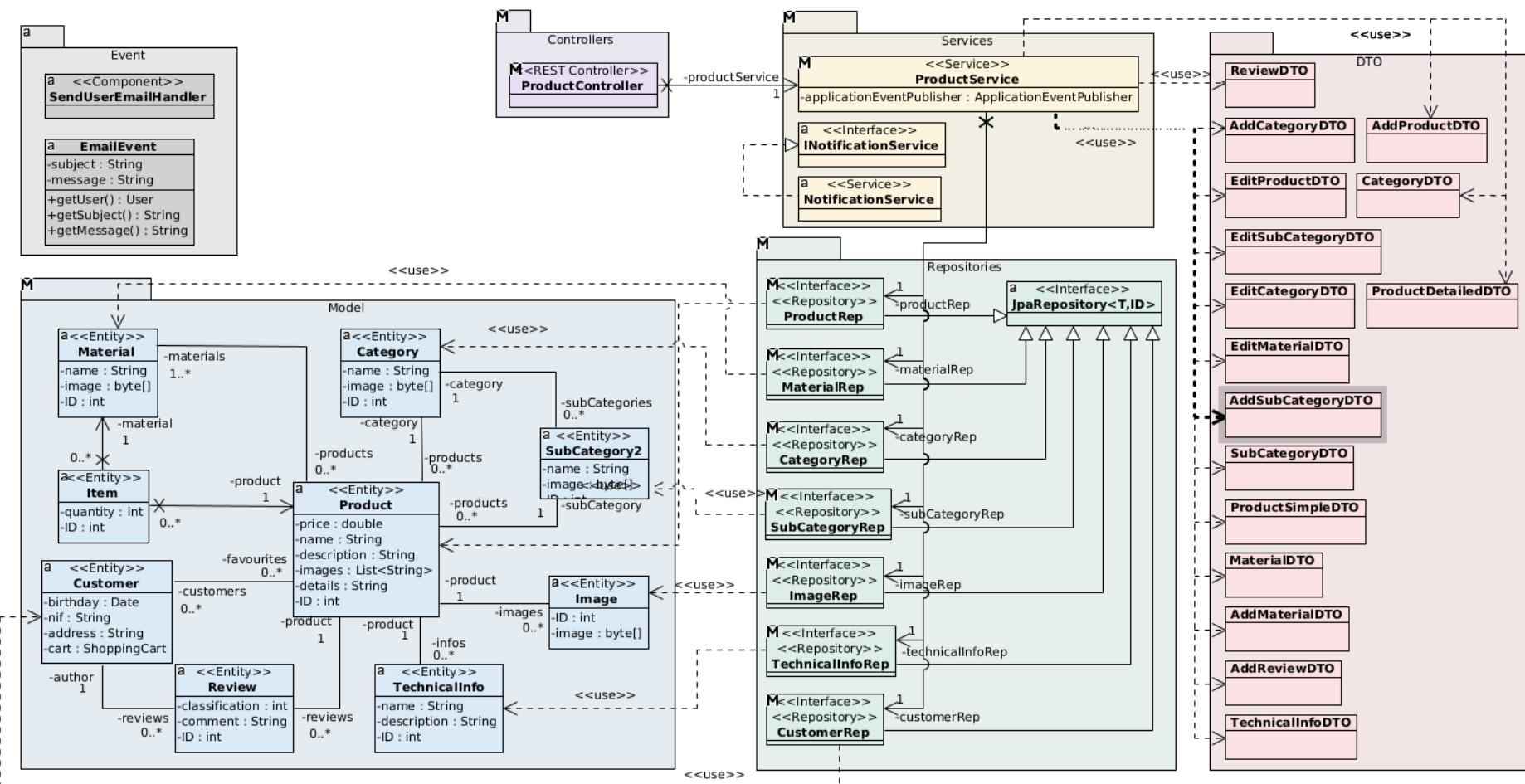


Figura 3.7: Componente Product

3.4.5 Componente *Order*

As funcionalidades necessárias para a criação de um encomenda estão todas presentes neste componente, desde a adição dos produtos ao *Shopping Cart* até à finalização do processo da encomenda.

Quando um produto é adicionado ao *Shopping Cart* é criado um *Item* que mantém uma referência para o produto selecionado. Quando a *Order* é criada, é necessário persistir o preço o preço a que foi comprado, criando assim as *OrderItem*. Só os clientes registado conseguem realizar e aceder ás encomendas, sendo o administrador responsável por gerir o estado destas.

O *Order Service* tal como o *Product Service* também gera notificações aos clientes, dado que o cliente é informado sobre alterações de estado nas suas encomendas.

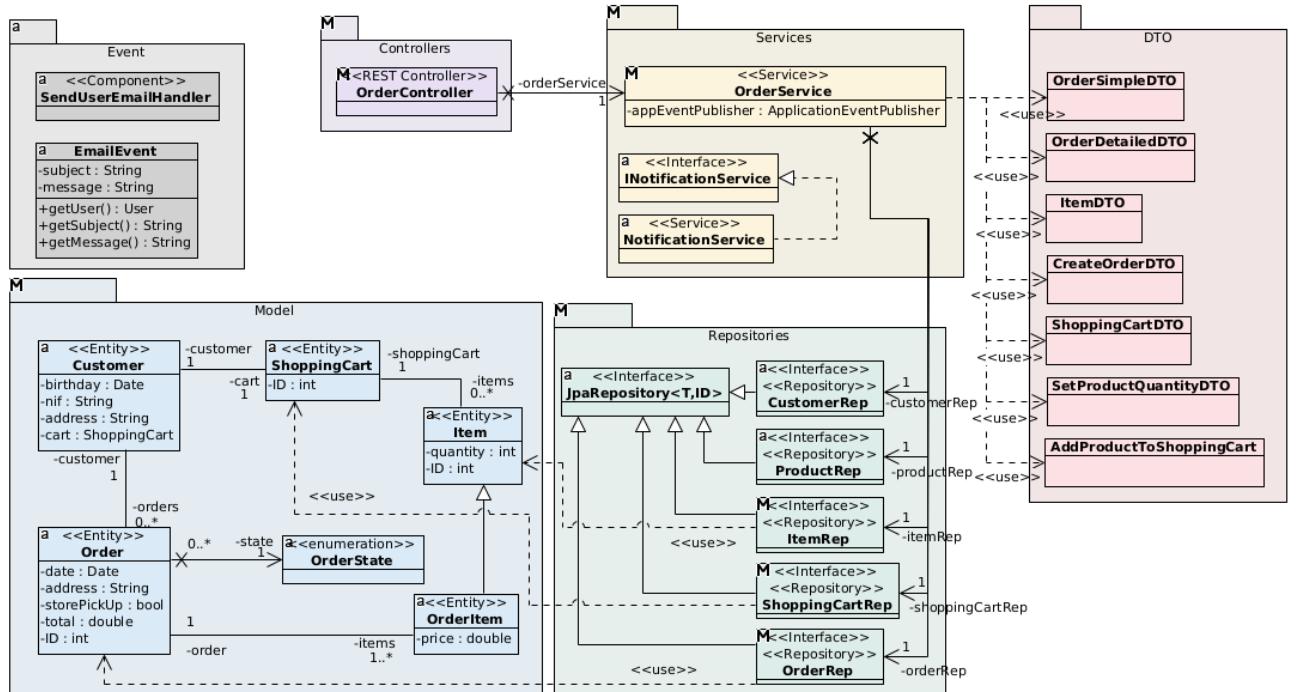


Figura 3.8: Componente *Order*

4. Vue Frontend App

A aplicação de *frontend* foi desenvolvida usando o *framework* Vue.js com o suporto do Vuetify, uma biblioteca de componentes UI baseada em Material Design.

A estrutura do projeto foi organizada seguindo o padrão **Atomic Design**, que é uma abordagem de *design* de componentes que divide a *interface* do utilizador em diferentes níveis de componentes reutilizáveis. Isto permitiu-nos também manter a consistência visual e funcional em toda a aplicação, além de facilitar a reutilização e manutenção dos componentes.

A estrutura do projeto é a seguinte:

- **appTypes** : é aqui que estão definidos todos os tipos de dados utilizados na aplicação.
- **router** : responsável pela definição das rotas da aplicação. Aqui encontram-se os URLs definidos e os seus componentes correspondentes que serão renderizados em cada rota.
- **locales** : Aqui, estão os ficheiros *JSON* utilizados para a tradução da aplicação. Este contém as *strings* traduzidas em dois idiomas (português e inglês). No caso de ser necessário adicionar uma nova linguagem, bastaria adicionar aqui um ficheiro **JSON**.
- **plugins** : Nesta pasta estão localizados todos os *plugins* utilizados na aplicação. Os *plugins* utilizados foram:
 - **funcionalidades do Vuetify** como a definição do tema global da aplicação. Aqui, encontram-se configuradas as cores da aplicação.
 - **Axios**: Nesta pasta encontram-se todos os ficheiros relacionados às chamadas de API do *backend* utilizando o Axios.
- **store** : Nesta pasta, utilizamos o **Pinia** para gerir o estado global da aplicação. O **Pinia** é uma biblioteca para gestão de estados voltada para o Vue.js.

Dentro desta pasta, podemos encontrar:

- **index.ts** : responsável por criar a instância do Pinia e exportá-la para uso em toda a aplicação.
- outros ficheiros **.ts**, onde são definidos cada módulo do Pinia. Um módulo é uma parte do estado global da aplicação que pode ser acessada e modificada pelos componentes. Cada módulo representa uma área ou recurso em específico da aplicação, tal como as categorias, notificações, etc...
- **components** : Aqui encontram-se armazenados todos os componentes da aplicação. Estes encontram-se divididos em três subdiretórias cruciais: **atoms**, **molecules** e **organism**, representando os diferentes níveis de componentes do **Atomic Design**.
 - **Atoms** : são os componentes mais básicos e reutilizáveis, como botões e tipografia.
 - **Molecules** : são combinações de **atoms** e representam componentes mais complexos, como botões com títulos.

- **Organisms** : são componentes completos que podem ser usados como páginas ou partes de páginas.
- **layouts** : Nesta pasta encontram-se definidos todos os *layouts* que são usados nos componentes da aplicação. Estes definem a estrutura visual básica das páginas.
- **views** : Aqui estão localizadas as páginas da aplicação. Cada ficheiro representa uma página da aplicação e contém toda a lógica e os componentes necessários para essa página em específico.

Esta estrutura do projeto facilitou a organização e reutilização de componentes, conseguindo com isto abstrair todos os componentes das suas responsabilidades. Fora isso, permitiu um desenvolvimento mais modular e escalável da aplicação, facilitando a manutenção e o trabalho em equipa.

4.1 Facilidade de aprendizagem

Considerámos importante que o utilizador tivesse *feedback* do sistema de modo a perceber que efeito as suas ações passadas tiveram sobre o mesmo. Para isso, utilizamos mensagens simples que surgem no topo do ecrã informando o utilizador sobre o efeito das suas ações. Por exemplo, ao tentar remover um administrador irá surgir no ecrã do utilizador uma das duas seguintes mensagens:



Figura 4.1: Mensagem de administrador removido com sucesso.

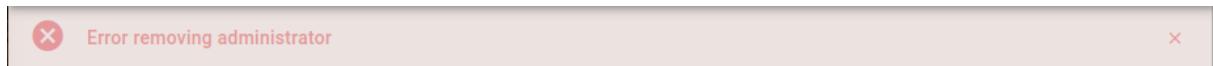


Figura 4.2: Mensagem de erro ao tentar remover administrador.

Utilizámos a mesma estratégia para remoção, edição e adição de outras entidades como produtos, materiais, categorias, etc e, até mesmo na mudança de estado das encomendas.

Para além disso, optámos por criar páginas parecidas de modo a que os utilizadores conseguissem aplicar o conhecimento de interação de uma página a outras e também garantir consistência entre as diferentes páginas. Por exemplo, as páginas onde o administrador pode consultar encomendas, utilizadores e informação sobre a empresa seguem o mesmo *layout* trazendo uma sensação de familiaridade ao utilizador:

The screenshot shows a web application interface for managing orders. At the top, there are two tabs: "All orders" on the left and "Pending orders" on the right. On the left side, there is a sidebar with three filter options: "Pending" (radio button selected), "Ready", and "Done". Below the sidebar is a search bar with a magnifying glass icon. The main content area displays two orders. Each order card includes the order number, a "CHANGE TO READY" button, a "VIEW DETAILS +" button, and a dropdown menu. Order #1 is associated with "Client #1", was requested on "2021-05-05", and has a total of "500€". Order #2 is associated with "Client #2".

Figura 4.3: Página das encomendas.

The screenshot shows a web application interface for managing users. At the top, there are two tabs: "All Users" on the left and "Admins" on the right. On the left side, there is a sidebar with three options: "My profile" (radio button selected), "Admins", and "Clients". Below the sidebar is a large "ADD ADMIN" button. The main content area displays a list of three admins. Each admin card includes the admin number, name, email, and a "REMOVE ADMIN" button. Admin #33 is named "Admin" and has the email "admin@admin.com". Admin #40 is named "Admin Lee" and has the email "admin7@example.com". Admin #41 is partially visible with the name "Admin Scott".

Figura 4.4: Página dos utilizadores.

The screenshot shows a user interface for managing company information. On the left, there's a sidebar with three options: 'Company' (selected), 'Categories', and 'Materials'. The main area is titled 'Company' and contains fields for address, postal code, schedule, phone number, and email. Below this is a section titled 'Network Links' with fields for Facebook and Instagram URLs.

Address:	Rua do Carvalhal, nº 18, Vila Nova de Famalicão, Portugal
Postal code:	4770-847
Schedule:	Monday - Friday 9:00 AM - 12:30 PM 2:00 PM - 7:00 PM Saturday CLOSED Sunday CLOSED
Phone number:	252 993 990
Email:	geral@rodrigues-moveis.pt

Facebook:	https://www.facebook.com/MoveisRodrigues
Instagram:	https://instagram.com/azevedorodriguesmoveis?igshid=NTc4MTIwNjQ2YQ==

Figura 4.5: Página das informações da empresa.

De modo a tornar o acesso rápido às páginas que consideramos mais importantes criamos uma barra de navegação que se adaptada a cada tipo de utilizador.

Para todos os utilizadores foram mantidos alguns componentes em comum: o ícone para modificar a linguagem em que o *website* é apresentado, a barra de pesquisa, o *drawer* que contém ligações rápidas para todas as categorias e o logo serve de *link* para a página inicial (*design pattern home link*).

Para além disso, na barra de navegação de um utilizador não registado apenas existe um *icon* que o redireciona para a página de *login*:



Figura 4.6: Barra de navegação de um utilizador não registado.

Por outro lado, a barra de navegação de um cliente contém ligações para o seu perfil (as suas informações pessoais e encomendas), para o seu carrinho de compras e para a sua lista de favoritos, assim como um *icon* para terminar a sessão:



Figura 4.7: Barra de navegação de um cliente.

Por sua vez, a barra de navegação de um administrador tem ligações para a lista de encomendas, a lista de utilizadores registados do sistema, para a informação da empresa (dados, materiais e categorias) e, também, o *icon* para terminar sessão:

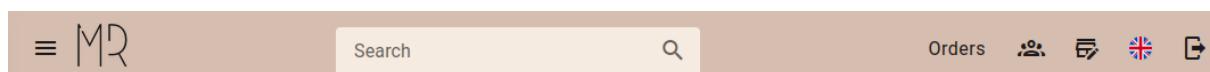


Figura 4.8: Barra de navegação de um cliente.

Com o intuito de facilitar a procura do utilizador quer por categoria ou por subcategoria criou-se um *drawer* e uma barra de pesquisa.

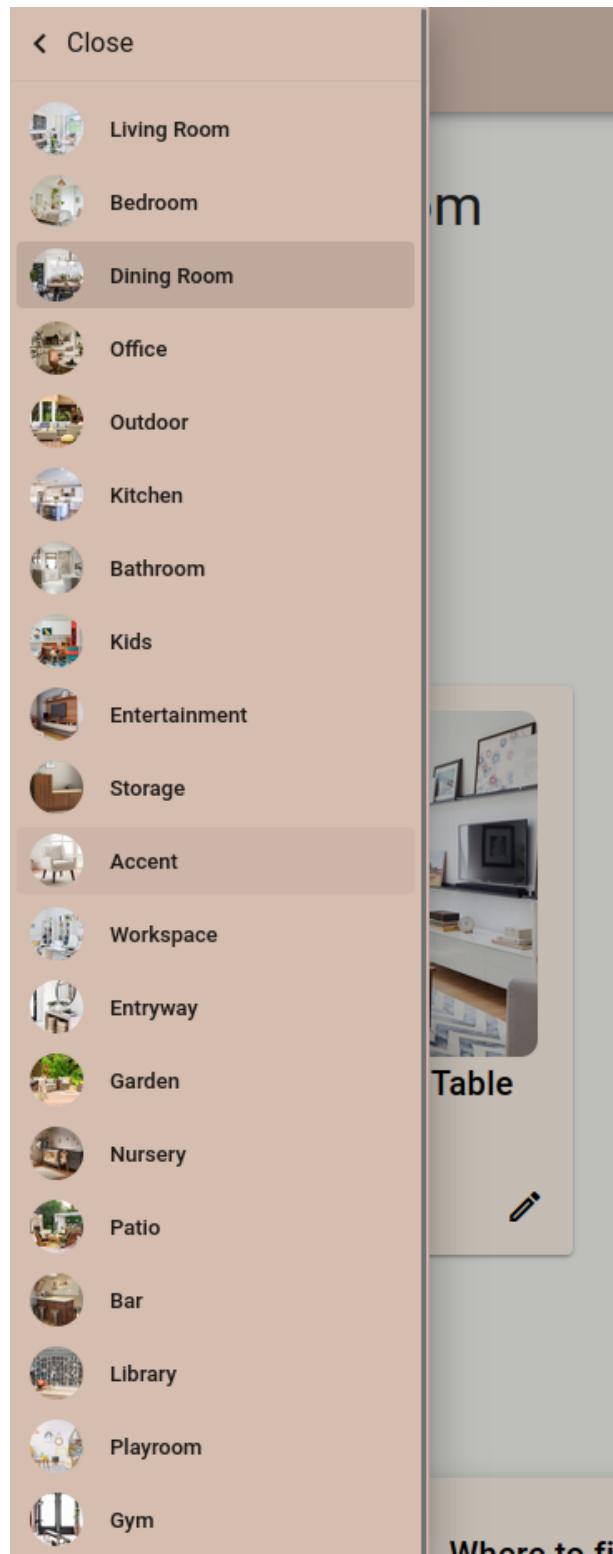


Figura 4.9: Exemplo de procura por categoria

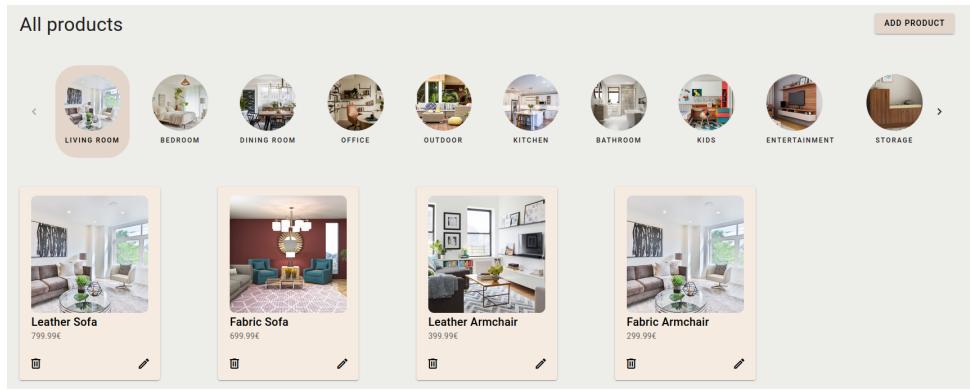


Figura 4.10: Exemplo de procura por categoria na página principal

De modo ao utilizador tomar uma decisão mais informada ao selecionar um material de um dado produto ou de um administrador conseguir visualizar as subcategorias de uma categoria, recorremos ao *Tooltips* para indicar o nome do material ou o nome das subcategorias.

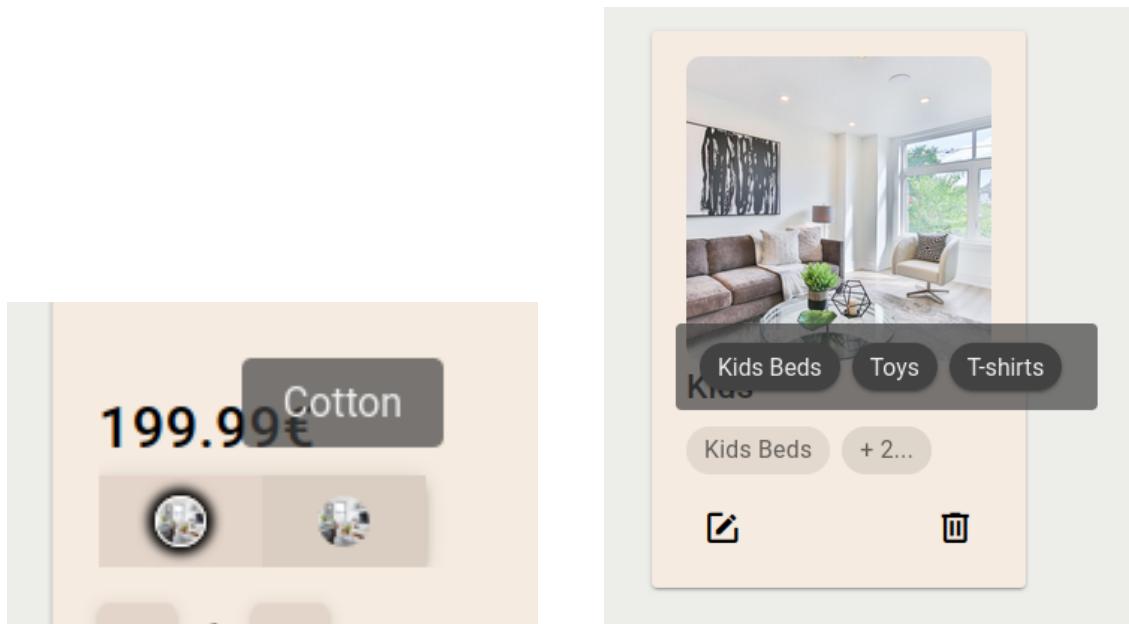


Figura 4.11: Exemplo de tooltips usados

4.2 Flexibilidade

Para tornar a utilização do *website* agradável em qualquer dispositivo tornamos o mesmo adaptável ao tamanho do ecrã disponível (*adaptivity*). Por exemplo, em ecrãs maiores a página de um produto específico tem o seguinte aspeto:

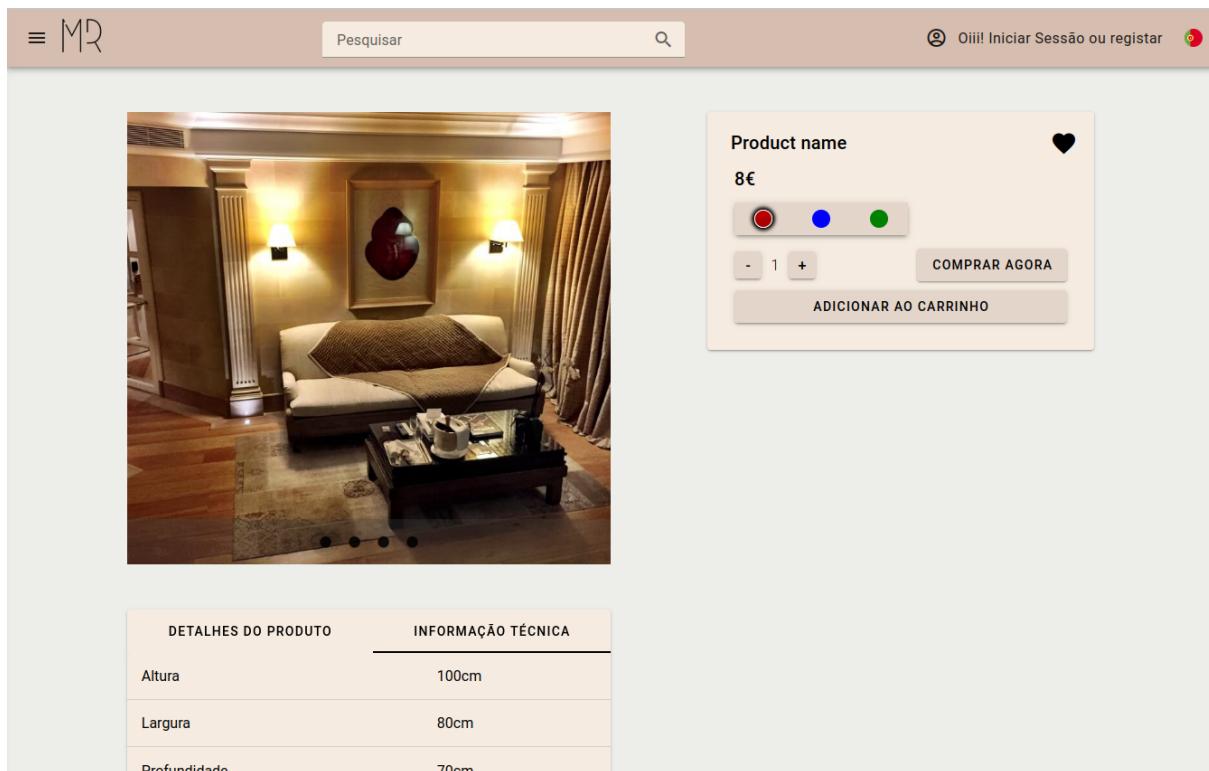


Figura 4.12: Página de um produto para ecrãs maiores.

Num ecrã menor os componentes adotarão a seguinte disposição:



DETALHES DO PRODUTO		INFORMAÇÃO TÉCNICA
Altura		100cm
Largura		80cm
Profundidade		70cm

Figura 4.13: Página de um produto para ecrãs menores (1/2).

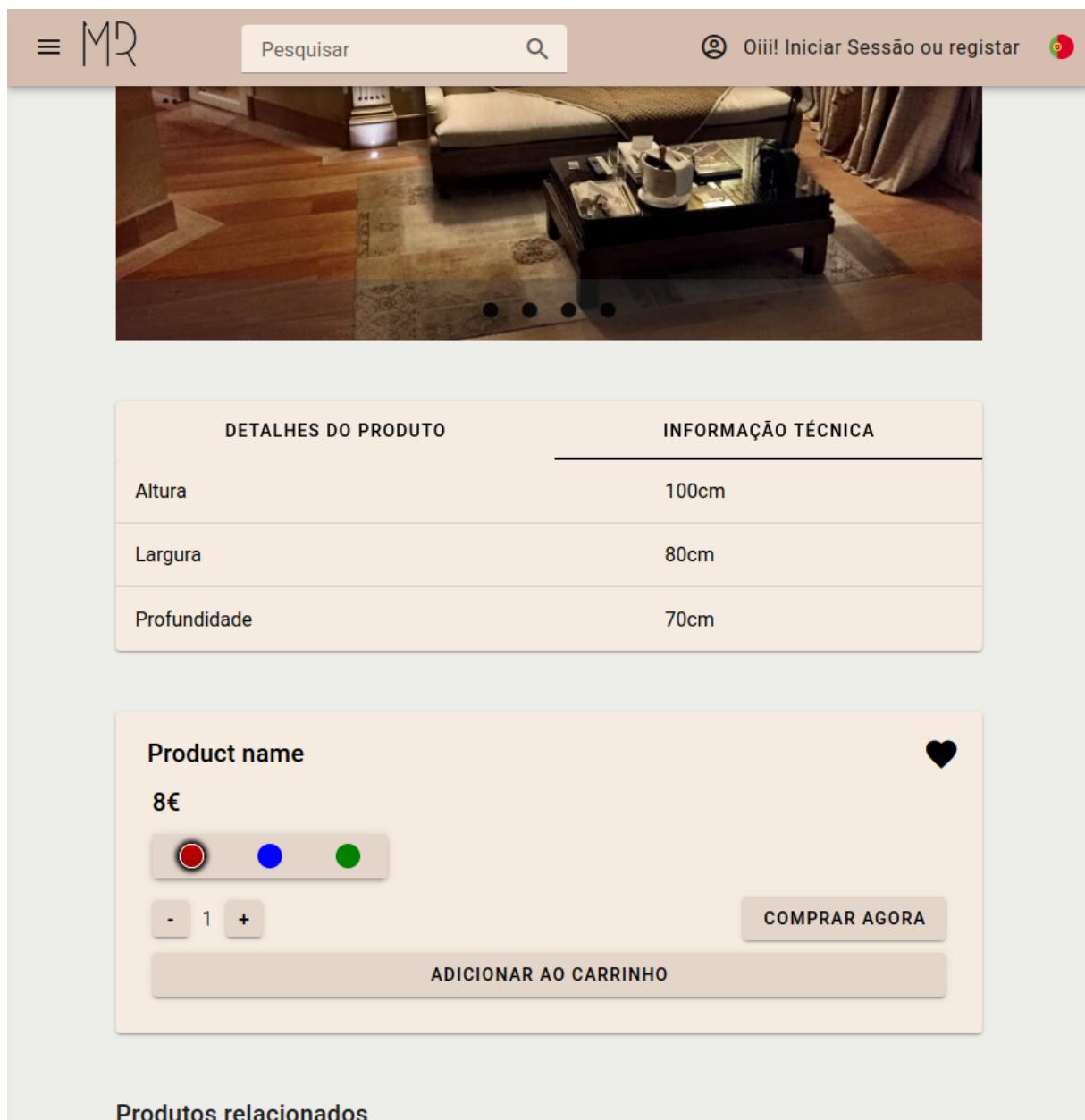


Figura 4.14: Página de um produto para ecrãs menores (2/2).

Num ecrã ainda mais pequeno o aspeto será o seguinte:

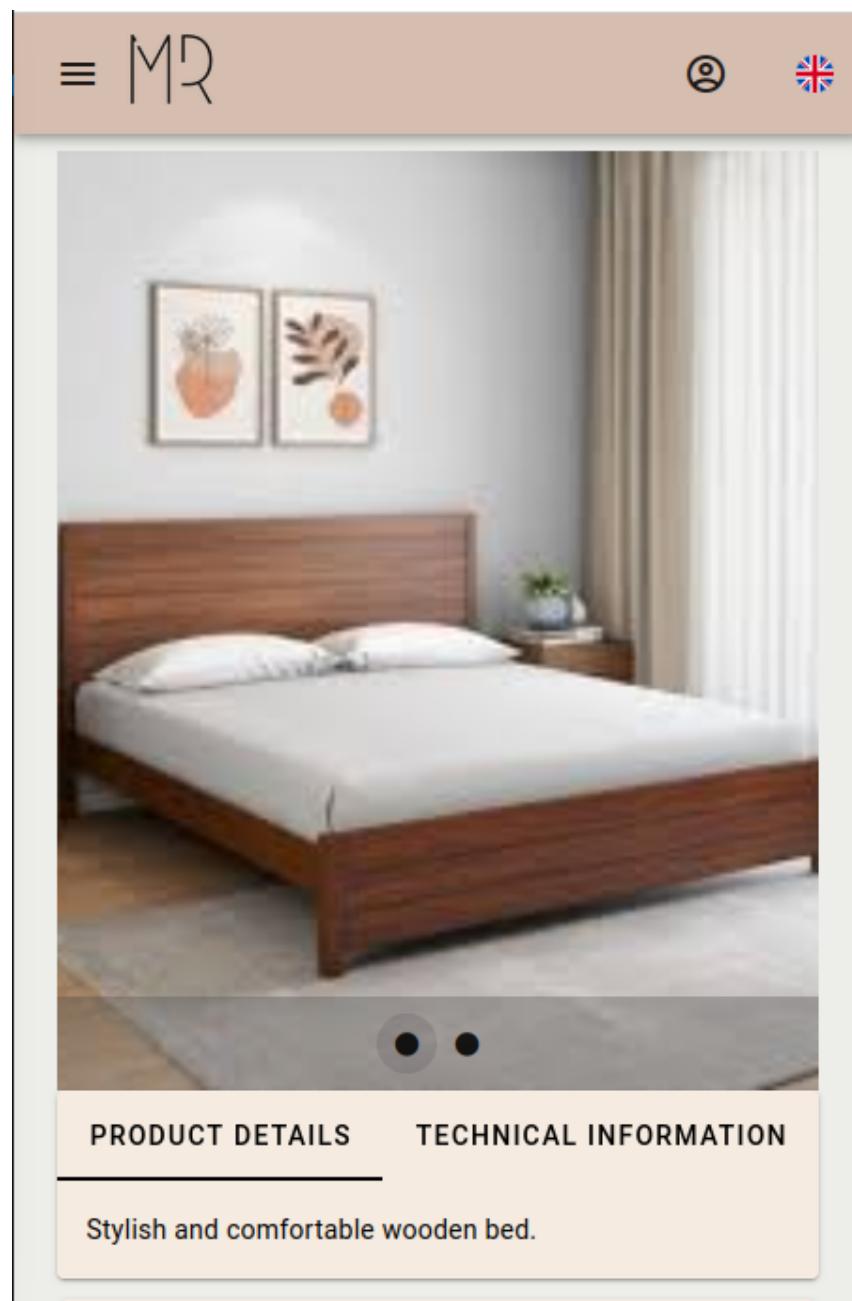


Figura 4.15: Página de um produto para ecrãs menores.

Introduzimos também a possibilidade de um utilizador mudar a linguagem na qual a informação lhe é apresentada, tendo como opções inglês e português.

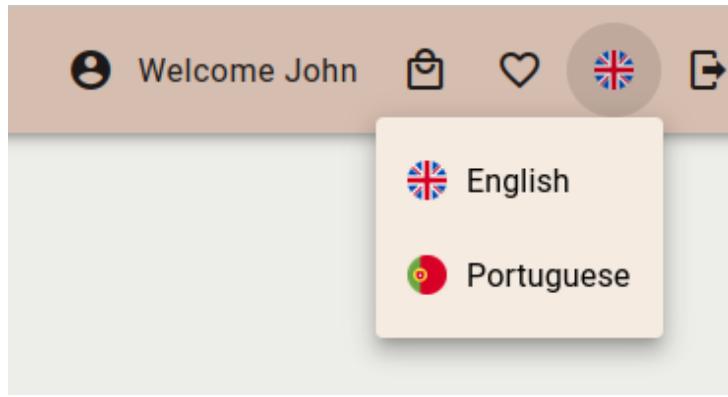


Figura 4.16: Menu para mudar a linguagem.

Para além disso, utilizamos barras de pesquisa pelo nome para produtos, administradores e clientes. Estas barras de pesquisa tornam mais fácil de um utilizador encontrar o produto que deseja. Pode ver-se nas imagens a baixo o exemplo da barra de pesquisa para os produtos:



Figura 4.17: Barra de navegação (1/2).

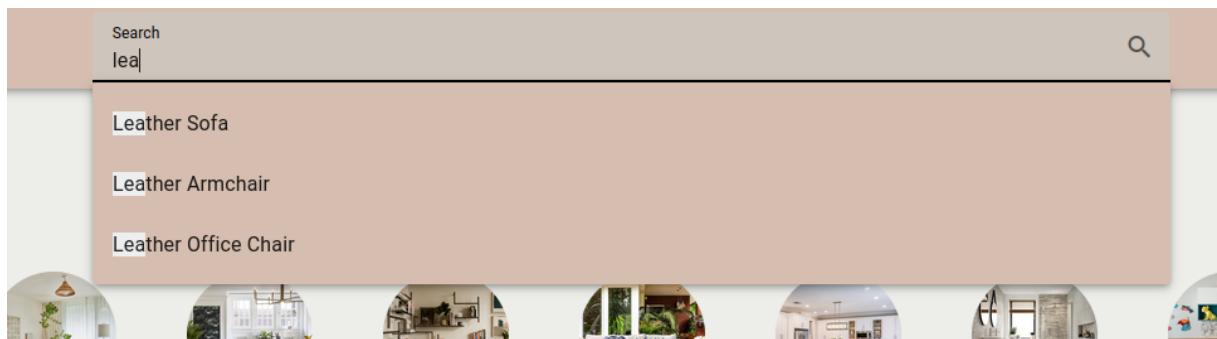


Figura 4.18: Barra de navegação (2/2).

4.3 Robustez

Quisemos também garantir que estava sempre visível para o utilizador as operações que o mesmo poderia executar. Por exemplo, nos formulários para edição optámos por colocar os botões de "guardar alterações" sempre no topo do formulário em vez de no final, pois caso o

formulário fosse grande o suficiente esse botão poderia não ficar visível no ecrã se se encontrasse no fim do formulário:

The screenshot shows a web-based form for editing company details. At the top left is a back arrow labeled '← Empresa'. At the top right is a button labeled 'GUARDAR ALTERAÇÕES' (Save changes). The form fields and their values are:

- Nome:** Móveis Rodrigues
- Morada:** Rua do Carvalhal, nº 18, Vila Nova de Famalicão, Portugal
- Código postal:** 4770-847
- Horário:** Monday - Friday 9:00 AM - 12:30 PM | 2:00 PM - 7:00 PMSa
- Número de telemóvel:** 252993990
- Email:** geral@rodrigues-moveis.pt

Figura 4.19: Formulário para editar informações da empresa.

Para além disso, utilizámos paginação em todas as páginas onde podem ser apresentados vários itens com o objetivo de dar a entender ao utilizador de que os produtos existentes numa dada categoria não são apenas os que estão a ser apresentados no ecrã nesse momento:

The screenshot shows a list of clients with two entries visible:

Client #19	VIEW DETAILS +
Name: Alexander Wilson Email: customer19@example.com	▼

Client #20	VIEW DETAILS +
Name: Mia Johnson Email: customer20@example.com	▼

At the bottom, there is a navigation bar with page numbers: < 1 2 3 4 ... 27 >

Figura 4.20: Paginação.

Também foram implementados mecanismos de *forward recovery* em todas as operações que considerámos que exigiriam um esforço maior para voltar ao estado anterior: terminar sessão, remoção de produtos, administradores, etc. Neste caso recorremos ao *modal design pattern* uma vez que, nesses casos, queremos que o utilizador foque toda a sua atenção na confirmação (ou não) da ação passada:

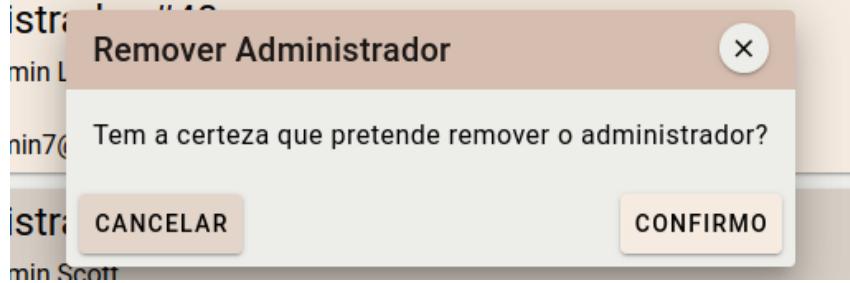


Figura 4.21: Modal para confirmação da remoção de um administrador.

No preenchimento ou edição de formulários, de modo a evitar erros não é permitido a submissão dos mesmos sem todas as informações preenchidas tal como podemos verificar de seguida:

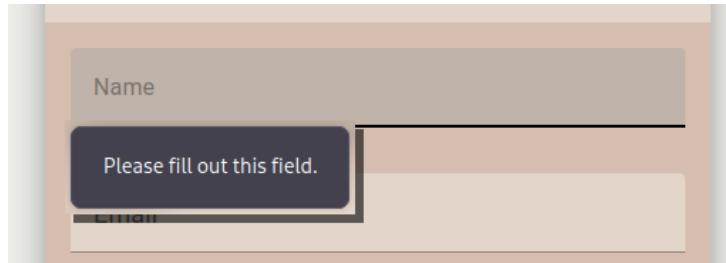


Figura 4.22: Impedir a submissão de formulários com campos vazios.

Também utilizamos *feedback* sobre o *input* de cada campo nos formulários (*design pattern input feedback*) de modo que o utilizador saiba a causa exata pela qual não é possível submeter os dados que introduziu. Assim, por baixo do campo onde foi inserido um valor considerado inválido apresentamos sempre uma mensagem de erro. Por exemplo, se um utilizador tentar introduzir um valor no campo NIF que não seja constituído por 9 números (nada mais, nada menos), irá surgir a seguinte mensagem de erro:

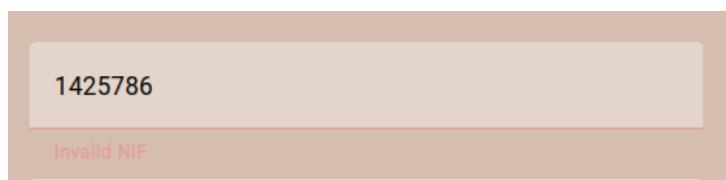


Figura 4.23: Mensagem de erro ao tentar introduzir NIF inválido.

Recorremos também ao *design pattern calendar picker* para os clientes introduzirem a sua data de nascimento aquando do registo:



Figura 4.24: Calendário para escolher data de nascimento.

Por fim, aplicamos também nas páginas que podem estar sujeitas a maior demora no carregamento do conteúdo uma barra de progresso. Isto permite indicar ao utilizador que o sistema tá em atividade quando não é possível dar uma resposta curta ou instantânea:



Figura 4.25: Barra de progresso.

5. Testes

Ao longo do desenvolvimento da aplicação é importante a realização de testes para verificar o bom funcionamento da aplicação. Por essa razão, foi adotada uma metodologia em que, a cada funcionalidade nova adicionada, esta era submetida a um conjunto de testes.

À medida que a aplicação foi sendo desenvolvida, existiu a necessidade de criar um *script* para popular a base de dados, de forma a agilizar a realização de testes e evitar a repetir os mesmos passos ao longo do tempo. Este *script* popula a base de dados com diferentes clientes e administradores, cria um conjunto de categorias e subcategorias com diferentes produtos e materiais. Além disso neste *script* são realizadas algumas operações como adicionar produtos aos favoritos de clientes aleatórios, e realizar encomendas com um conjunto de produtos aleatórios.

Os testes foram feitos com a ferramenta **Postman**, desta forma era possível fazer os pedidos diretamente aos *Controllers* e visualizar as respostas obtidas.

5.1 Teste de Carga

Os testes de carga numa aplicação são úteis para compreender o comportamento da aplicação com um elevado stress, isto é, submeter a aplicação a uma carga de trabalho muito alta e analisar métricas de performance. Nesta secção será abordado os testes de carga realizados bem como a arquitetura utilizada.

Quando se realizam testes a uma aplicação, é importante garantir que esses testes são fidedignos e que se assemelham ao comportamento esperado dos utilizadores. Tendo isso em atenção e com a utilização da ferramenta **JMeter** foi desenvolvido um teste que procura simular o comportamento de um cliente na aplicação:

1. O cliente realiza o login
2. O cliente obtém a lista de todas as categorias
3. Seleciona uma categoria aleatória e obtém a lista de produtos associados a essa categoria
4. Escolhe um produto aleatório
5. Escolhe um material aleatório desse produto e adiciona ao carrinho
6. Realiza uma encomenda

O **JMeter** permite que este teste seja executado concorrentemente por um número configurável de clientes. Para realizar estes testes foi desenhada uma arquitetura estática na *google cloud* com mostra a figura 5.1.

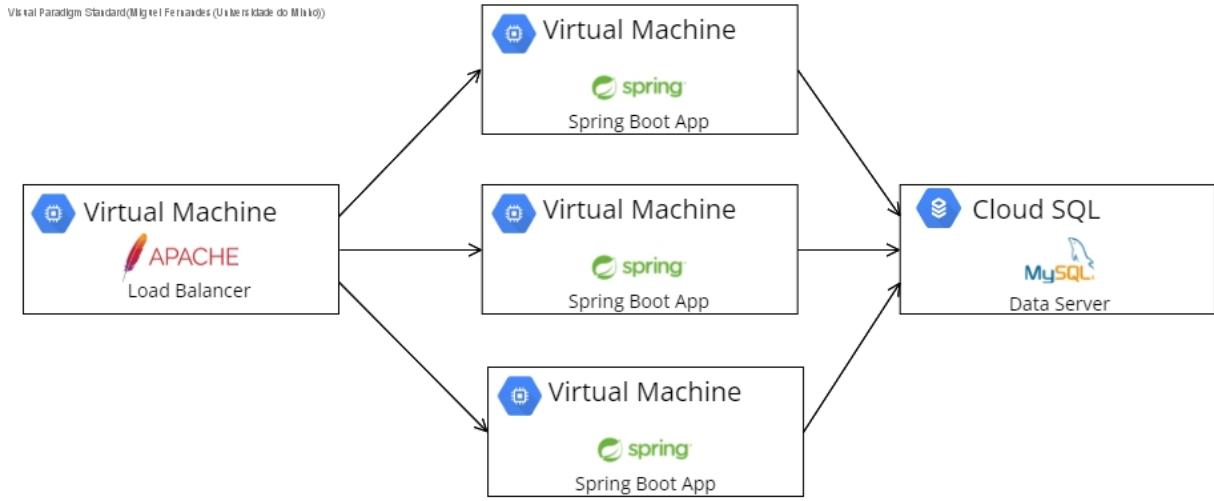


Figura 5.1: Arquitetura *Cloud* da aplicação

A aplicação *Spring* é lançada em 3 máquinas virtuais diferentes, tendo uma máquina virtual dedicada apenas um *Apache Web Server* com o módulo *mod_proxy* responsável por fazer o balanceamento de carga entre elas. Todas as aplicações *Spring* estão ligadas à mesma base de dados, em que foi utilizado o *Cloud SQL* que trata da administração de base dados e oferece uma alta disponibilidade.

Idealmente o número de máquinas virtuais não deveria ser estático, uma alternativa era utilizar os *load balancers* fornecidos pela *Google Cloud* que permite lançar/remover dinamicamente máquinas virtuais com base na carga recebida. Além disso, todas as aplicações estarem ligadas à mesma base de dados, acaba por gerar um *bottleneck* de desempenho. Poderiam ter sido consideradas técnicas de replicação de base de dados e balanceamento de pedidos, mais concretamente, a criação de réplicas de leitura e balancear os pedidos pelas réplicas.

Com esta arquitetura e com o teste de carga referido anteriormente, foram realizadas várias iterações com diferente números de clientes em simultâneo. Os seguintes testes, foram gerados a partir da ferramenta *JMeter* e demonstram os resultados obtidos com 50, 100, 200, 400 e 500 clientes:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
login	50	695	415	1190	228.28	0.00%	16.4/sec	12.04	3.99	752.4
Get all categories	50	2589	746	3145	567.80	0.00%	9.1/sec	17.19	1.39	1937.0
Get products of ...	50	546	251	857	143.08	0.00%	9.5/sec	5.00	1.57	538.5
Get product Det...	50	560	254	893	182.02	0.00%	9.7/sec	6.48	1.31	682.6
Add product to S...	50	601	241	1184	328.68	0.00%	9.9/sec	3.90	4.83	405.0
Create Order	50	450	235	1108	313.79	0.00%	9.5/sec	3.76	4.46	405.0
TOTAL	300	907	235	3145	822.76	0.00%	35.7/sec	27.43	9.85	786.8

Figura 5.2: Métricas para 50 utilizadores

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
login	100	1432	1050	1752	179.93	0.00%	56.5/sec	41.53	13.78	752.8
Get all categories	100	6863	5230	7675	440.66	0.00%	12.0/sec	22.70	1.84	1937.0
Get products of ...	100	1064	573	1852	338.71	0.00%	21.8/sec	11.46	3.59	538.1
Get product Det...	100	1044	669	1801	313.77	0.00%	23.7/sec	15.79	3.18	681.8
Add product to S...	100	850	151	1332	235.00	1.00%	26.0/sec	10.67	12.59	420.8
Create Order	100	632	29	1472	355.89	4.00%	24.3/sec	11.13	10.97	468.2
TOTAL	600	1981	29	7675	2220.04	0.83%	44.2/sec	34.51	12.02	799.8

Figura 5.3: Métricas para 100 utilizadores

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
login	200	1651	458	3125	638.58	0.00%	49.2/sec	36.23	12.05	754.4
Get all categories	200	12541	8489	18982	2892.55	0.00%	9.4/sec	17.86	1.45	1937.7
Get products of a category	200	1883	241	3556	854.21	0.00%	15.1/sec	7.96	2.49	538.2
Get product Details	200	1900	241	4087	844.35	0.00%	16.3/sec	10.84	2.19	681.8
Add product to Shopping cart	200	648	1	1630	530.79	34.50%	17.7/sec	17.08	5.71	986.9
Create Order	200	794	20	1742	368.27	3.00%	16.8/sec	7.48	7.68	455.2
TOTAL	1200	3236	1	18982	4396.96	6.25%	48.9/sec	42.59	12.02	892.4

Figura 5.4: Métricas para 200 utilizadores

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
login	400	2702	685	4673	1093.19	0.00%	68.2/sec	50.29	16.75	755.2
Get all categories	400	20876	9370	35422	7140.02	0.00%	9.9/sec	18.71	1.52	1937.9
Get products of ...	400	8749	263	20253	6447.77	0.00%	12.9/sec	6.78	2.12	539.1
Get product Det...	400	4620	271	19536	4337.50	0.00%	13.4/sec	8.96	1.80	683.1
Add product to S...	400	401	0	3246	545.33	67.00%	14.9/sec	22.30	2.41	1536.5
Create Order	400	1711	31	15412	1989.50	3.75%	15.3/sec	6.98	6.92	468.2
TOTAL	2400	6510	0	35422	8232.41	11.79%	53.4/sec	51.43	11.69	986.7

Figura 5.5: Métricas para 400 utilizadores

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received KB/...	Sent KB/sec
login	500	3410	3405	5335	5608	6039	840	6223	0.00%	69.5/sec	51.24	17.07
Get all categ...	500	25154	24206	38058	40031	43818	8935	44600	0.00%	9.8/sec	18.58	1.51
Get products ...	500	11442	8853	25096	25439	25976	755	26923	0.00%	11.3/sec	5.96	1.86
Get product ...	500	6564	5117	16818	24091	25727	309	26580	0.00%	11.6/sec	7.76	1.56
Add product...	500	375	133	1288	1623	2038	3	5537	74.80%	12.0/sec	19.52	1.49
Create Order	500	2782	1991	5461	7001	22367	6	25715	6.40%	11.9/sec	5.98	5.27
TOTAL	3000	8288	3835	24990	32344	39488	3	44600	13.53%	52.9/sec	52.49	11.14

Figura 5.6: Métricas para 500 utilizadores

A partir dos 5 resultados é possível retirar algumas conclusões:

- O maior salto obtido no *Throughput* foi de 50 utilizadores para 100 utilizadores, sendo que para números de clientes superiores a subida não foi tão significativa
- Para 400 utilizadores em simultâneo, a percentagem de erro, nomeadamente, no pedido *Add product to Shopping Cart*, apresenta valores consideravelmente altos. Os erros obtidos eram do tipo *NoHttpResponseException*, sendo que uma análise ao *log* da aplicação *Spring* não permitiu diagnosticar a causa da exceção.
- Um número elevado de clientes, como por exemplo 400, para esta arquitetura comprometeria o bom funcionamento da aplicação.
- É notório que não existe uma quebra no *Throughput* com o aumentar do clientes, por outro lado é visível que a percentagem de erro aumenta com o número de clientes.

Os utilizadores do teste de carga são gerados pelo *script* que popula a base de dados, para isso foi gerado outro *script* em *Python* que contrói um csv, com o email e password desses utilizadores. Esse csv é utilizado depois para fazer o *login* dos clientes.

6. Conclusão

Inicialmente, o grupo pretendia tirar proveito da ferramenta *Visual Paradigm* para gerar, a partir dos modelos PSM, os ficheiros XML para o *Hibernate*. Contudo, como se optou por utilizar a tecnologia *Spring Boot* (com *Hibernate* incluído) e tendo em conta que esta tecnologia favorece o uso de anotações, por uma questão de coerência tomou-se a decisão de recorrer a anotações ao invés de configurações com ficheiros XML.

No levantamento de requisitos foram definidos alguns requisitos que não conseguiram ser implementados, nomeadamente, os requisitos legais como utilização de *cookies* e também o cumprimento do **Regulamento Geral sobre a Proteção de Dados**(RGPD). Além destes requisitos, existe por exemplo as *Reviews* que foram implementadas na aplicação *backend* porém, dado que não era uma requisito de alta prioridade, não estão implementados na aplicação *frontend* e, portanto, não estão visíveis para os utilizadores.

No desenvolvimento da aplicação *frontend* focámo-nos em proporcionar uma boa experiência de interação para o utilizador. Utilizámos, para isso, vários *design patterns* de modo a tornar a interação fácil e agradável.

Desde o inicio do trabalho foi adotada uma metodologia de trabalho que maximizasse a capacidade de trabalho em paralelo. O levantamento de requisitos e descrição dos *use cases* permitiu, à partida, definir as funcionalidades necessárias para a aplicação. Com isto e com os modelos desenvolvidos, o *frontend* sabia quais os métodos que ia precisar e informação a receber para gerar as páginas, por outro lado no *backend* era claro quais os métodos a implementar.

Como demonstrado na secção dos Modelos PSM, nomeadamente nos Modelos PSM Spring, foi feita uma divisão por componentes. Esta divisão permitiu que os componentes fossem implementados independentes um dos outros e em paralelo. Para auxiliar nesta tarefa foi utilizado o *Git* em que a cada funcionalidade adicionada era feito um *Pull Request* que, posteriormente, era analisado por outro membro.

O grupo considera que a metodologia de trabalho utilizada foi benéfica para o desenvolvimento da aplicação.

7. Anexos

14

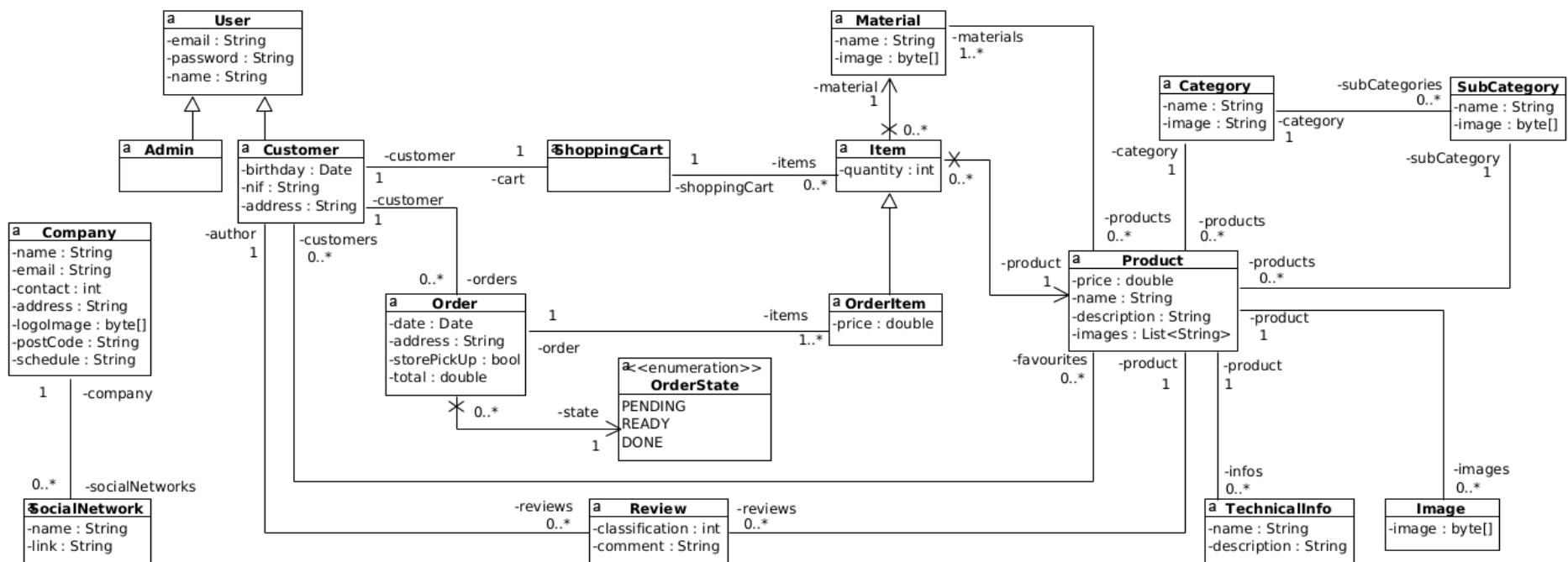


Figura 7.1: Modelo PIM

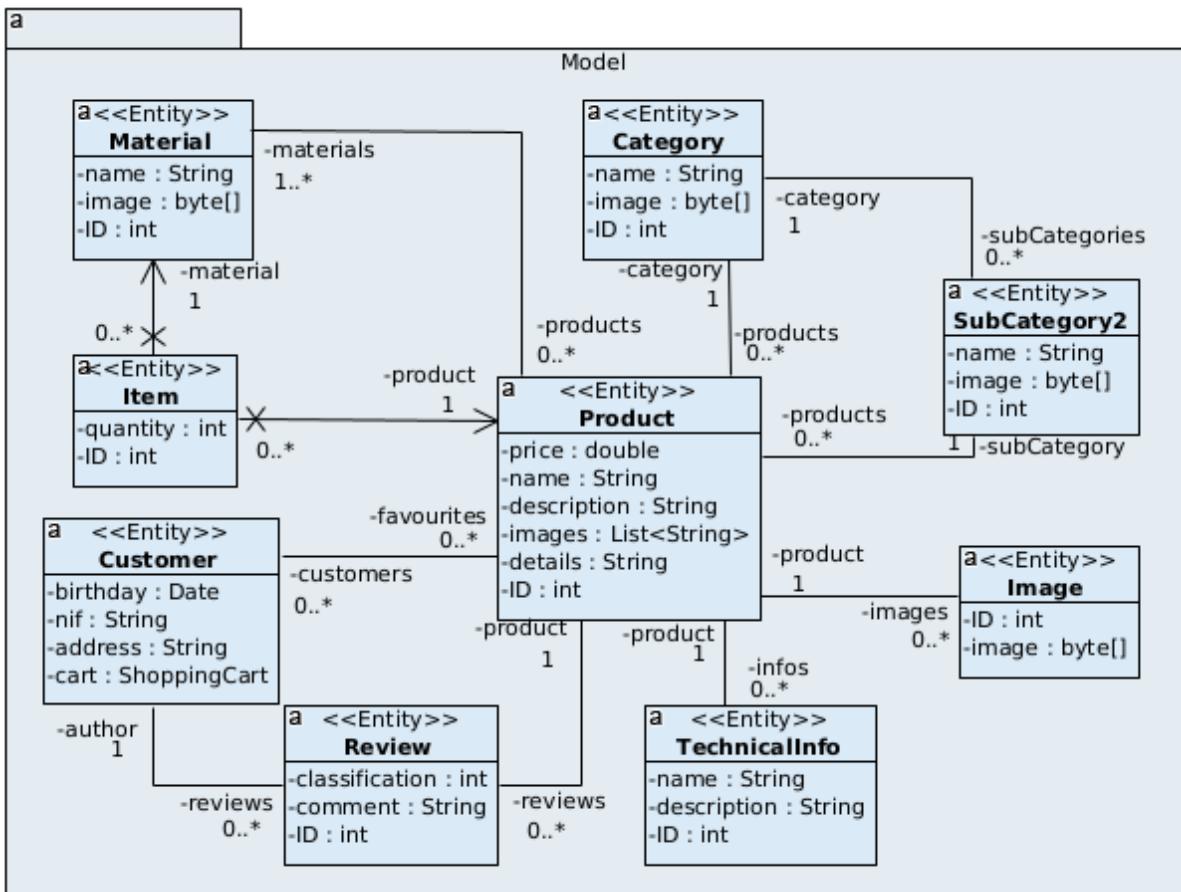


Figura 7.2: Product Model

a	Controllers
	<pre> <<REST Controller>> ProductController +addProduct(addProductDTO : AddProductDTO) : ProductSimpleDTO +removeProduct(productId : int) : void +editProductSimple(productId : int, editProductDTO : EditProductDTO) : ProductSimpleDTO +addProductImages(productId : int, images : List<byte[]>) : void +removeProductImages(productId : int, images : List<byte[]>) : void +addCategory(addCategoryDTO : AddCategoryDTO) : CategoryDTO +editCategory(categoryId : int, editCategoryDTO : EditCategoryDTO) : CategoryDTO +removeCategory(categoryId : int) : boolean +addSubCategory(addSubCategoryDTO : AddSubCategoryDTO) : SubCategoryDTO +editSubCategories(editSubCategoriesDTO : EditSubCategoriesDTO) : void +removeSubCategory(subCategoryId : int) : void +addMaterial(addMaterialDTO : AddMaterialDTO) : MaterialDTO +editMaterial(materialDTO : EditMaterialDTO) : MaterialDTO +removeMaterial(materialId : int) : void +addProductToCategory(productId : int, categoryId : int) : void +addProductToSubCategory(productId : int, subCategoryId : int) : void +getNumberOfProductsByCategory(categoryId) : int +getProductsByCategory(categoryId : int, offset : int, numItems : int) : EnvelopeDTO<ProductSimpleDTO> +getNumberOfProductsBySubCategory(subCategoryId : int) : int +getProductsBySubCategory(subCategoryId : int, offset : int, numItems : int) : EnvelopeDTO<ProductSimpleDTO> +getProductById(productId : int) : ProductDetailedDTO +productImage(productId : int, imageId : int) : ResponseEntity +categoryImage(categoryId : int) : ResponseEntity +subcategoryImage(subCategoryId : int) : ResponseEntity +materialImage(materialId : int) : ResponseEntity +getNumberOfCategories() : int +getAllCategories(offset : int, numItems : int) : EnvelopeDTO<CategoryDTO> +getAllMaterials(offset : int, numItems : int) : EnvelopeDTO<MaterialDTO> +getAllProducts(offset : int, numItems : int) : EnvelopeDTO<ProductSimpleDTO> +getNumberOfProducts() : int +addReview(addReviewDTO : AddReviewDTO) : void </pre>

Figura 7.3: Product Controller

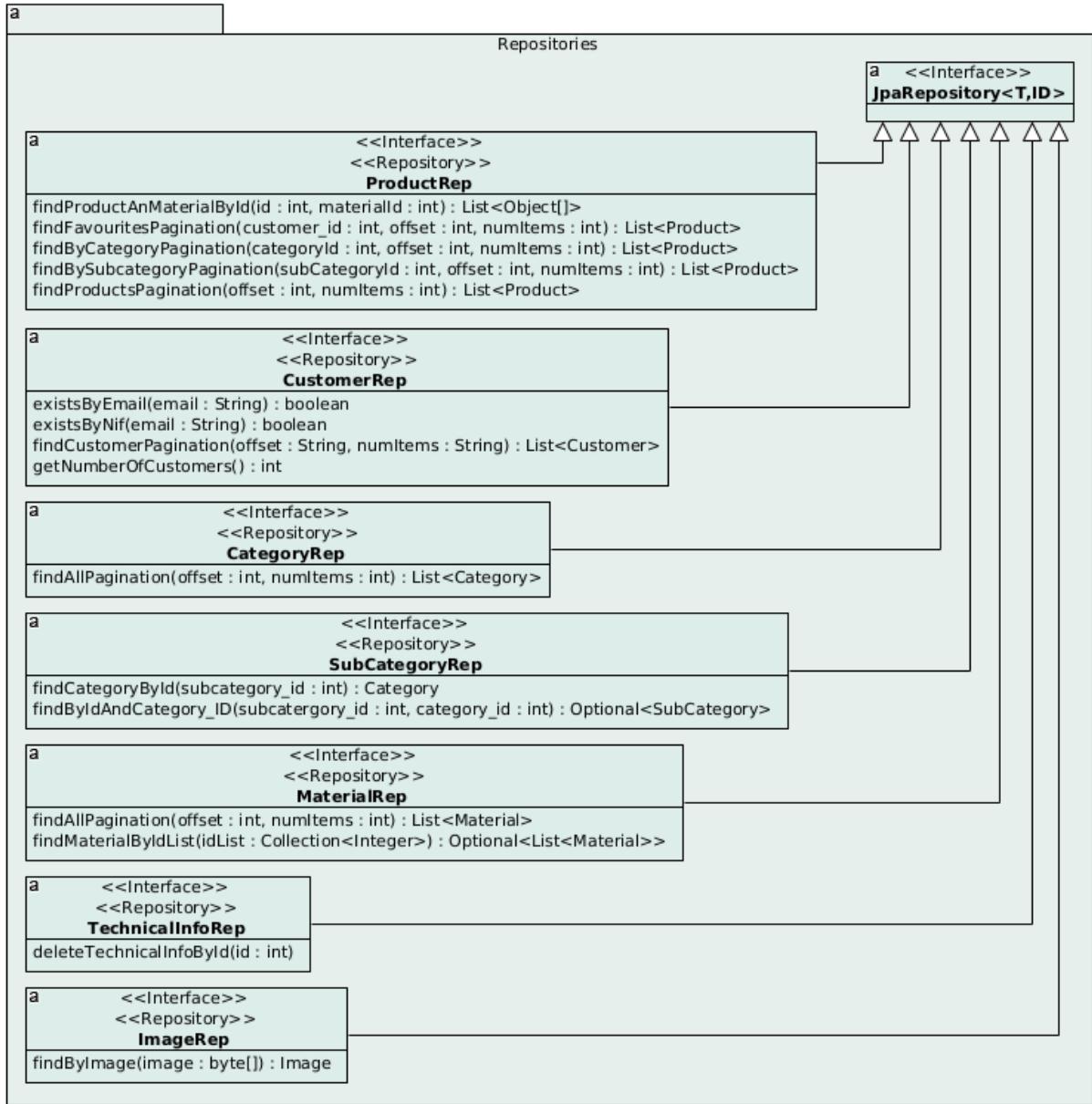


Figura 7.4: Product Repository

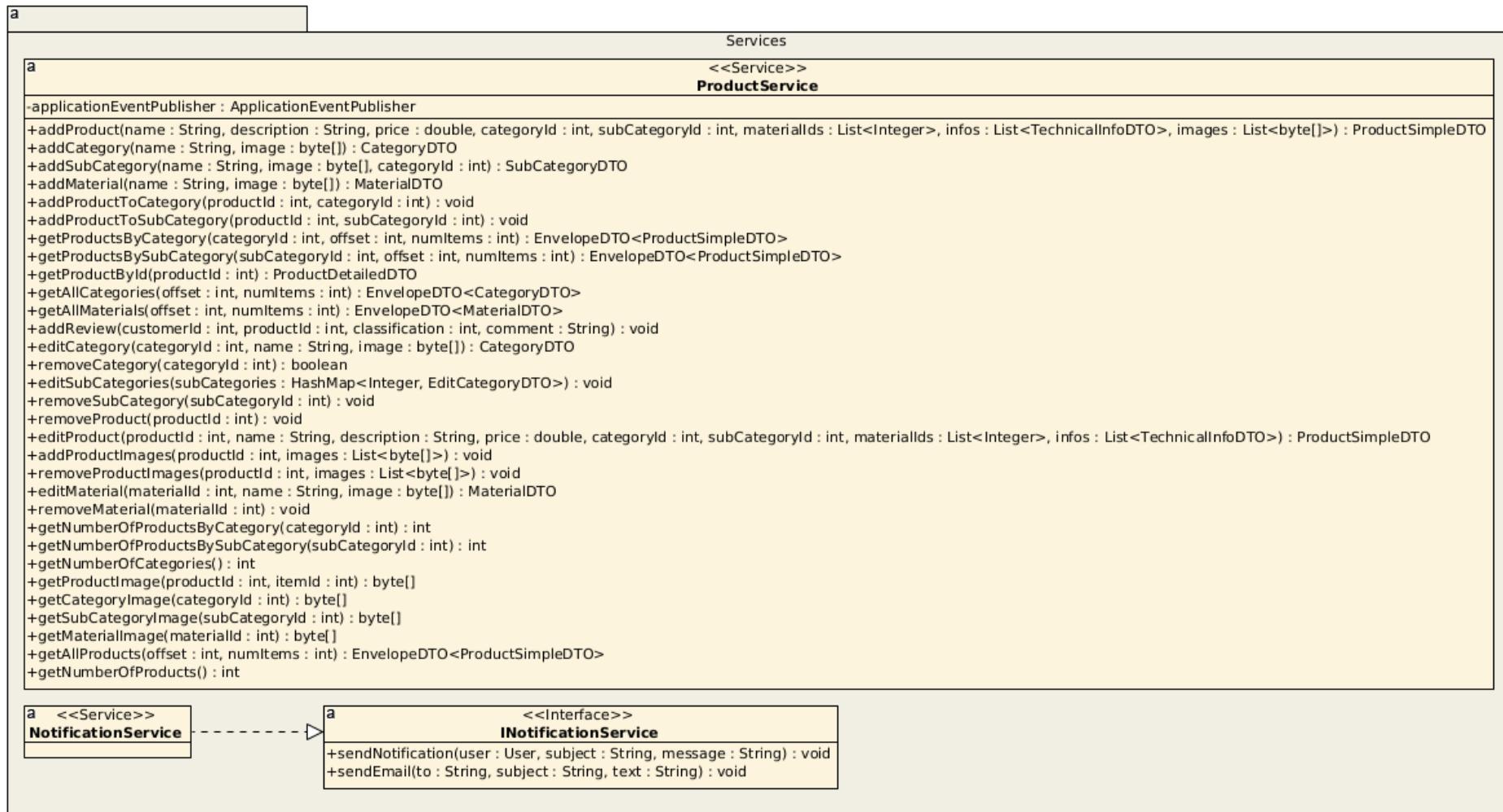


Figura 7.5: Product Service

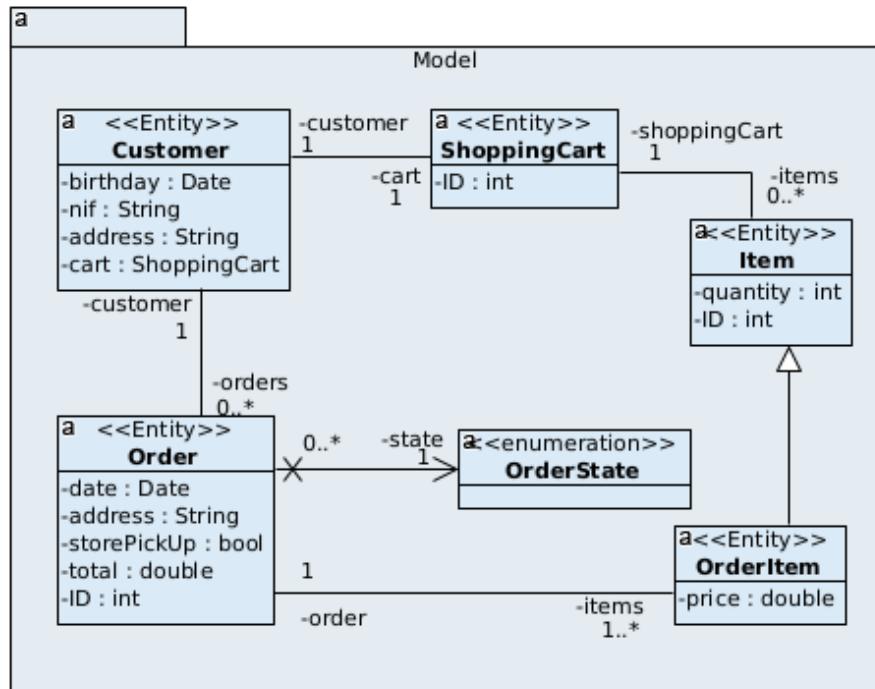


Figura 7.6: Order Model

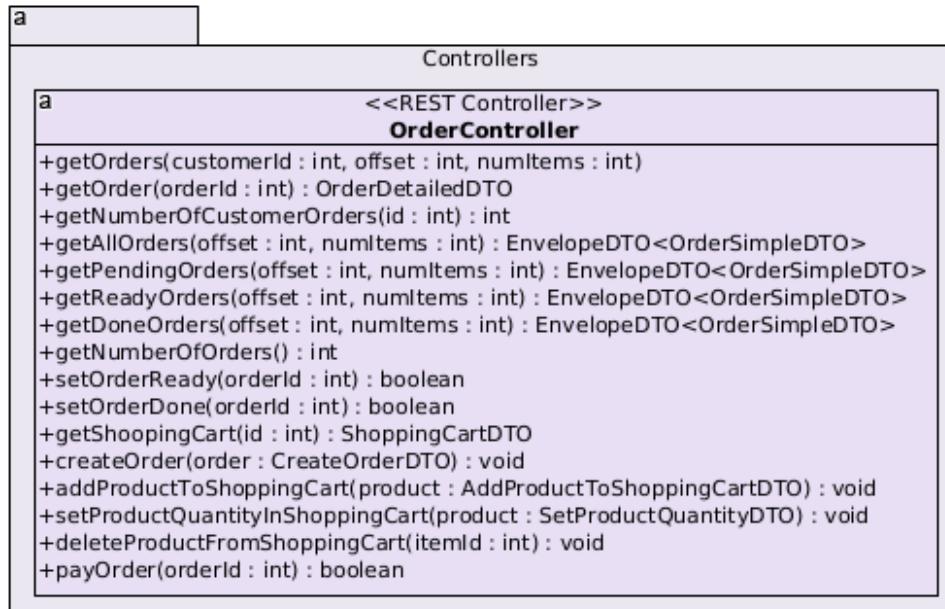


Figura 7.7: Order Controller

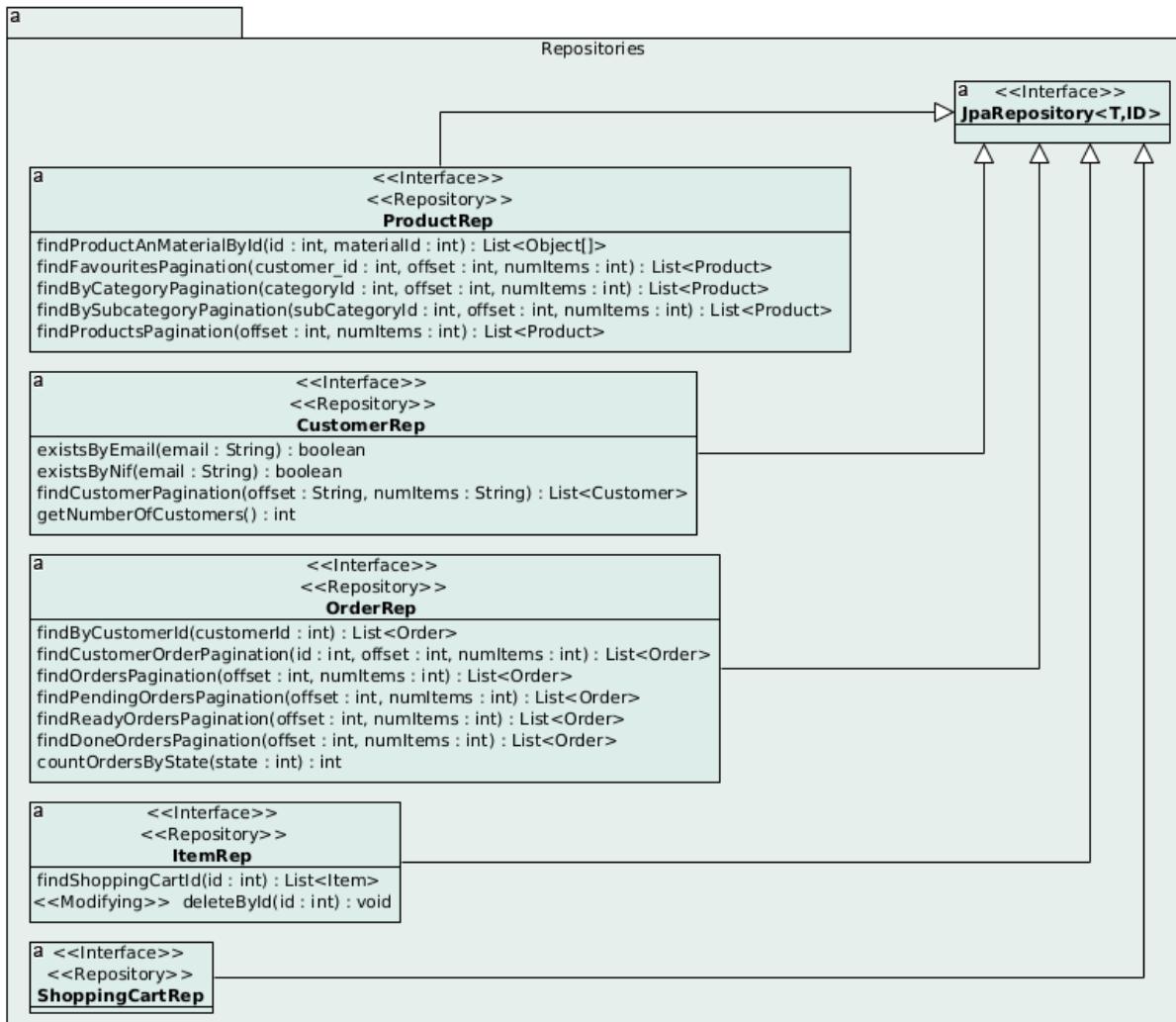


Figura 7.8: Order Repository

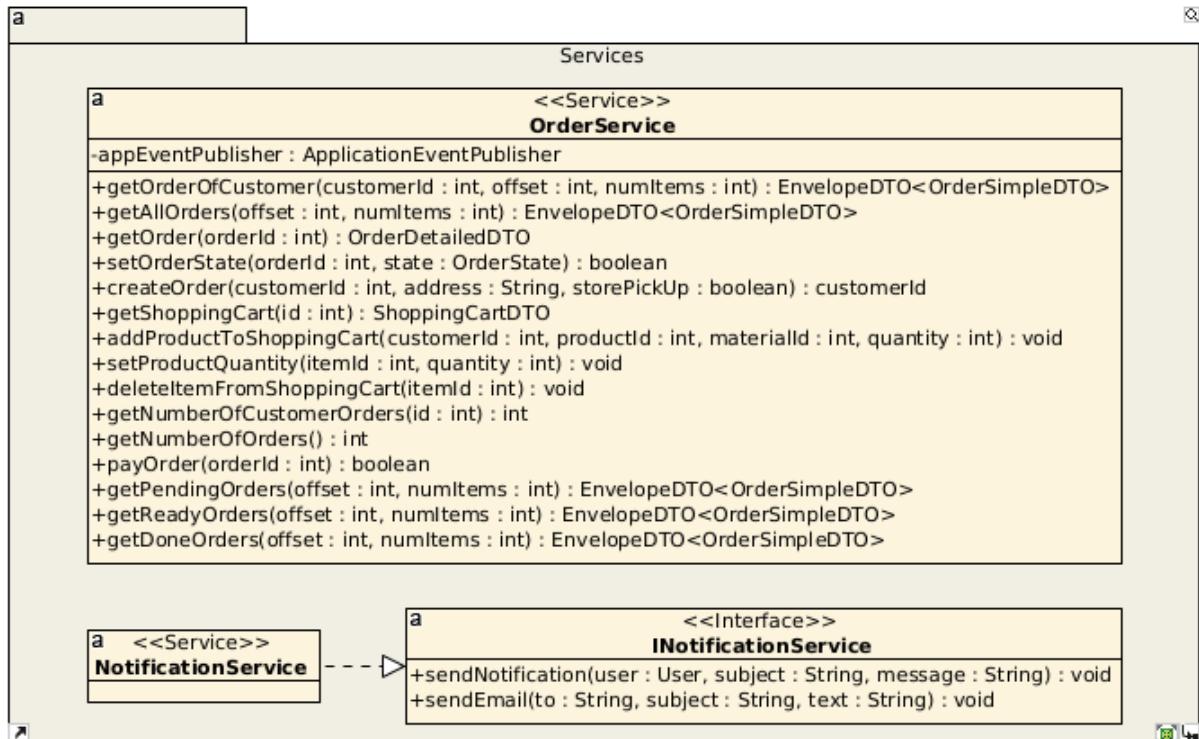


Figura 7.9: Order Repository