SWAM project assignment 2020-2021

I18N-Store: a RESTful application featuring full-text search on localized items

Student: Francesco Areoluci

Credits: 9 CFUs

Abstract

Hibernate Search is an enabling technology for full-text search over ORM entities for a Java Application. This library allows the developer to add search functionalities with an easy-to-use interface. Full-text search is implemented using Lucene, for local indexing, and Eleasticsearch for remote indexing. This project implement Hibernate Search functionalities in an application which represent an online shop. The shop features the localization of its products: a customer will receive products localized in his/her locale.

1 Introduction

Hibernate Search is an enabling technology for indexing and full-text search on Hibernate applications. In particular, it allows to extract data from hibernate ORM entities and push it to local or remote indexes. Hibernate Search is particularly useful for applications where SQL-based searches are not suited, such as full-text and geolocation searches. The main difference with traditional search is that the stored text is not considered as a single block of text, but as a collection of tokens (words).
In order to do that, Hibernate Search consists of an indexing component, which associates indexes to entities, and an index search component, which allows to search through indexes. These services are offered by Apache Lucene, an high-performance, full-featured text search library written in Java. Indexes are created and updated each time an entity is inserted, updated or removed from the database. Once the index is created, entity search can be accomplished without dealing with the underlying Lucene infrastructure.
Moreover, remote indexing and searching can be accomplished using Elasticsearch, a distributed search engine. These search engines are based on the concept of inverted indexes: a dictionary where the key is a token found in a document and the value is the list of identifiers of every document containing the token. A search involves the following steps:

- Extract tokens from the input query;
- Lookup tokens in the index to find matching documents;
- Aggregrate results of the lookup to produce a list of matching documents

The project's application has been developed by integrating the Hibernate Search features. In particular, the application represents the backend of an online shop which let the users to purchase products and search for them through query on their name and/or

description. Products fields, such as name and description, are localized in multiple locales: this project features the italian and english localization. Products are displayed to customer in his/her configured locale: an italian customer will get products which fields are localized in italian.
Through this application, Hibernate Search functionalities can be used on product's localized attributes: each product has fields available in multiple languages and those localized versions belong to the same product. These fields in multiple languages will be used by Hibernate Search to retrieve matching entities.

1.1 About Internationalization and Localization

Localization, often referred as *l10n* where 10 is the number of letters between *l* and *n*, is the process of adapting a product, an application or a textual document to a specific country or region. Localization is not limited to textual translation but it includes a variety of adaptations, such as:

- adapting graphics;
- adapting to local currencies;
- adapting date/time format;
- adapting addresses format

The process of localization in a product or application can be easily enabled through a process of internationalization.
Internationalization, often referred as *i18n* where 18 is the number of letters between *i* and *n*, is the process of design and development of a product or application that **enables easy localization**. As an example, an application architectural design that enables an easier localization of the content of the application, is a process of internationalization.

2 Hibernate Search Architecture

This chapter offers a brief view of the Hibernate Search architectures and modules. In particular, Hibernate Search is composed by the following modules and functionalities:

- **Backend**: The backend module abstracts the full-text search engine, by implementing indexing and searching interfaces. Hibernate Search provides two abstraction: Lucene backend and Elastisearch backend. The backend can be configured using Hibernate configuration.
- **Mapper**: This module allows to map the user model to an index model. In particular, Hibernate Search mapper offers the indexing of Hibernate ORM entities. This can be done through annotations or programmatic API.
- **Mass indexer**: Thanks to the mass indexing, Hibernate Search can rebuild indexes starting from pre-exisiting data stored on database.
- **Automatic indexer**: This allows to keeps indexes in sync with a database. Each time an entity is added, updated or removed, the mapper detects the changed and the index is updated.
- **Searching**: This is how Hibernate Search provides ways to query the index. The search involves the creation of a query, the token extraction and filtering, the search in the index and the entities retrieval.

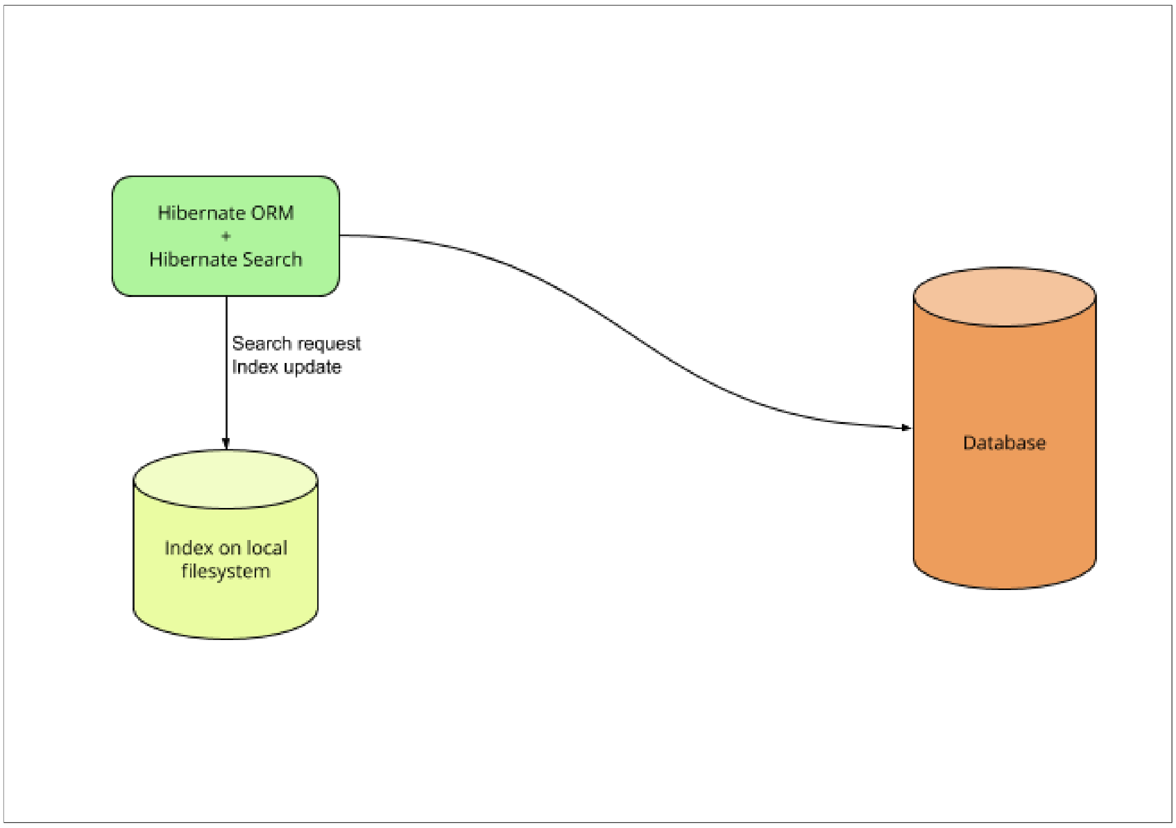


Figure 1: Hibernate Search Lucene Backend

3 Application requirements

The design of the application has started through the definition of its requirements. To do that, requirements have been divided in four categories:

- **FR**: Functional Requirements
- **NFR**: Non-Functional Requirements
- **DR**: Domain requirements
- **C**: Project Constraints

3.1 Functional Requirements

- **FR1**: The system must allow to manage the available products
- **FR2**: Each product must contains the following fields:
  - **FR2.1**: Product Name
  - **FR2.2**: Product Description
  - **FR2.3**: Price
  - **FR2.4**: Manufacturer
  - **FR2.5**: Product category
- **FR3**: The system must have the localization feature: products must be localized in multiple langauges (it, en). Localization feature must be used on the following fields:
  - **FR3.1**: Product Name
  - **FR3.2**: Product Description
  - **FR3.3**: Price
  - **FR3.4**: Product Category
- **FR4**: The system must allow the visualization and purchase of the products.
  - **FR4.1**: The product visualization must be localized: a product must be returned to the user with fields specified in FR3 properly localized.
- **FR5**: The system must features the search on products.
- **FR6**: The system must features the management of two types of user account
  - **FR6.1**: Administrator account - Responsabilities: Product management and research, as specified in FR1 and FR5
  - **FR6.2**: Customer account - Responsabilities: Product research and purchase, as specified in FR4 and FR5

3.2 Non-Functional requirements

- **NFR1** (Security Requirement): Application access must be managed through user authentication for all the account types described in FR6.
- **NFR2** (Implementation Requirement): The purchase functionality, as specified in FR4, must be prototyped as following:
  - **NFR2.1**: The user must have a shopping cart. The user can add and remove products to/from the cart;
  - **NFR2.2**: Once the user has added to the cart the desired prodcuts, he/she can proceed to the checkout. This operation will move the cart products to a shopping list. The cart will be cleared.

3.3 Domain Requirements

- **DR1**: The product management, as specified in FR1, must allow to add, remove and edit the products (CRUD operations)
- **DR2**: The products search, as specified in FR5, must be based on query keywords and must be used to search on products' name and description.
- **DR3**: A customer account must have an associated locale in order to implement the product visualization feature.

3.4 Project Constraints

- **C1**: The application must be developed using Java EE, with JPA and CDI technologies.
- **C2**: The Persistence layer must be managed using Hibernate.
- **C3**: The persistence must be managed by DBMS MariaDB.
- **C4**: The search layer must be manager through Hibernate Search library.
- **C5**: Application services must be exposed via REST API using JAX-RS technology.
- **C6**: Package management must be managed through Maven.

4 Application Design

4.1 Use Cases design

Using the previously described requirements, the use cases for the two types of account have been designed. The following image represents the use case diagram that has been created to show which operation each type of user can perform using the application.

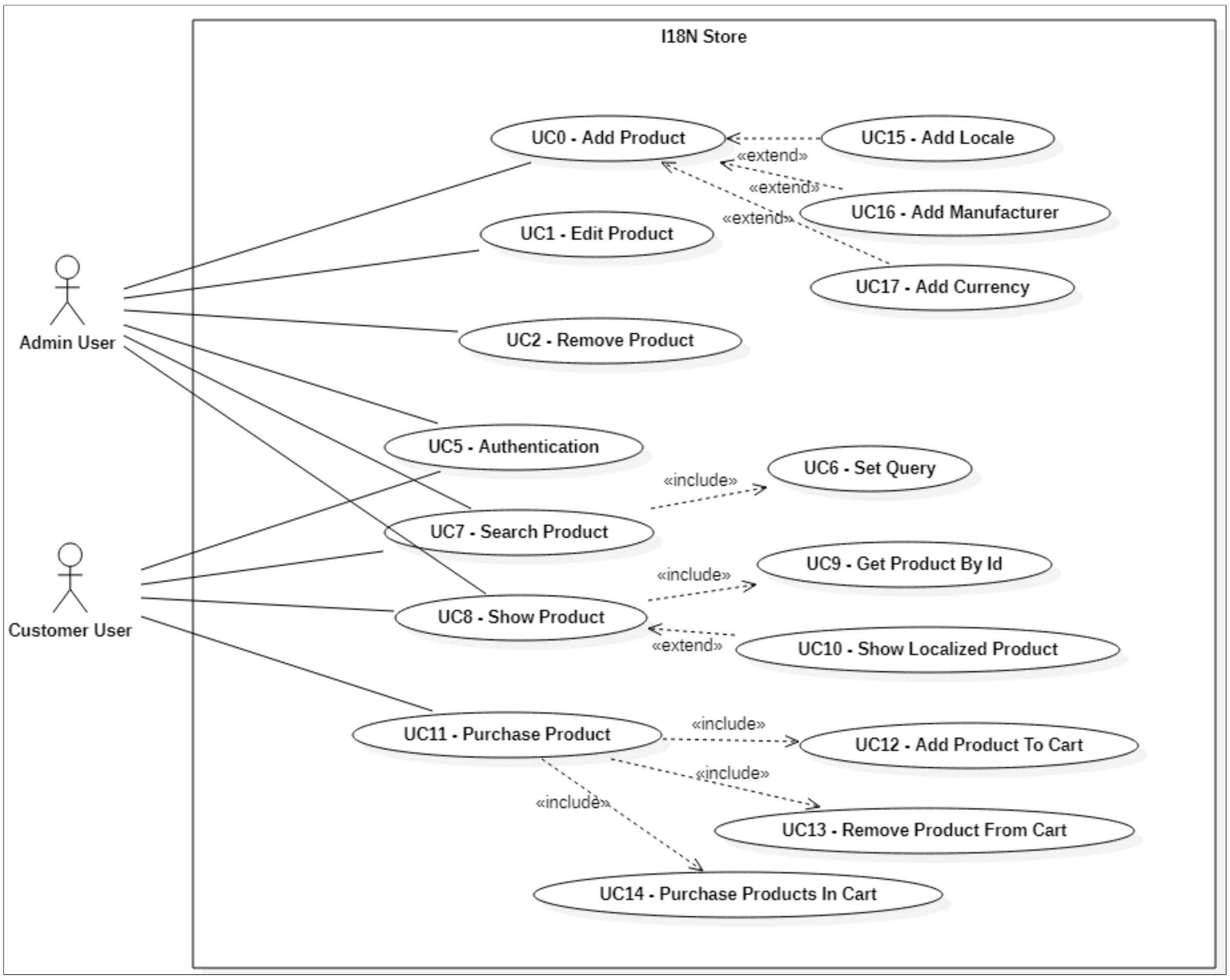


Figure 2: Use Case Diagram

As we can see from the diagram:

- an administrator user can interact with the system through insert, update and delete of products. To accomplish that, he/she can add manufacturer, locales and values;
- a customer user can interact with the system through (localized) visualization and purchase of products;
- common use cases between the two users are authentication, search and visualization of products.

4.2 Class design

Once the use cases have been designed, the application has been modeled through the UML class diagram. The application has been divided in packages, one for each logic domain. The diagram exposes the following packages:

- **Domain Model**: contains the entities that represents the application domain;
- **Translation Model**: contains the entities responsible of the field translation handling;
- **Rest**: contains the classes that implement the available rest endpoints;
- **Data Access Objects**: contains the entities responsible for DBMS interaction and instantiation of domain model entities;
- **Data Transfer Objects**: contains the entities used to transfer data from endpoints to client and vicevers, in JSON format;
- **Security**: contains the entities responsible of user authentication and authorization management.

The following image shows the class diagram of the entire application.

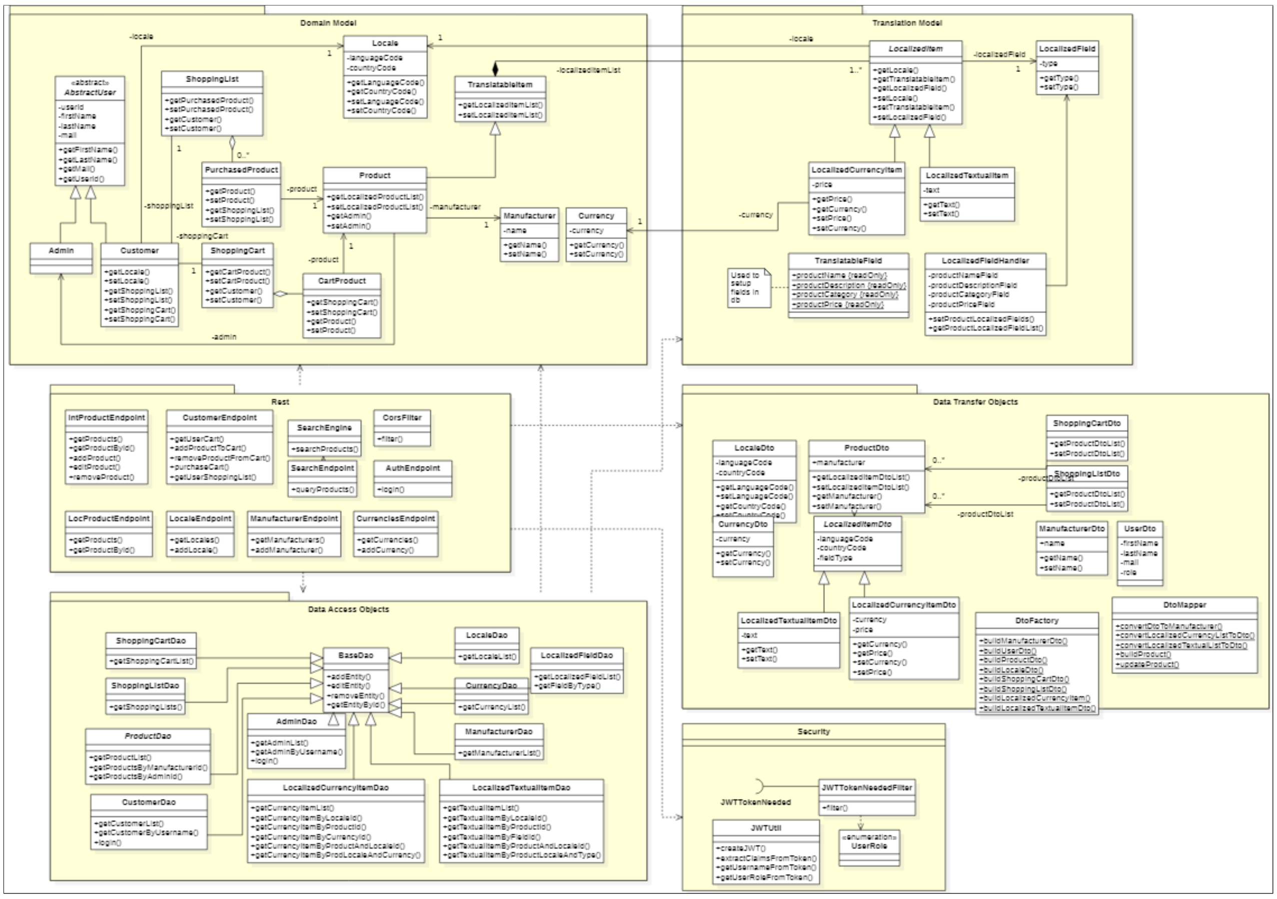


Figure 3: Complete class diagram

Each package will be examinated in the following sections.

4.2.1 Domain Model

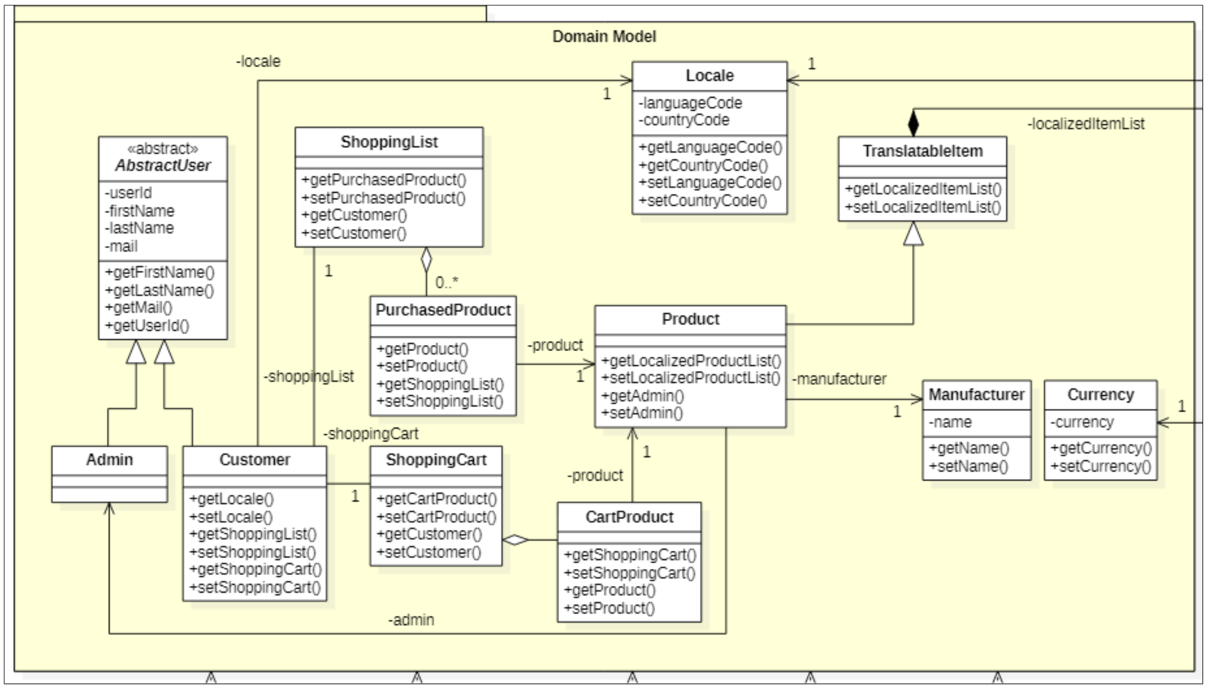


Figure 4: Domain Model Class Diagram

The domain model represents the base entities of the shop. It contains mainly POJO entities. In particular, it features the following entities:

- the two types of user: **Admin** and **Customer**;
- the **Product** entity, which instance is inserted by an Admin, has a **Manufacturer** and that subclasses a **TranslatableItem**. This entity has a relationship with the LocalizedItem entity of the Translation Model: each entity that need a translation of some fields can subclass the TranslatableItem entity;
- the **ShoppingCart** and the **ShoppingList**. These entities are associated respectively to the product in cart (**CartProduct**) and to the **PurchasedProducts**;
- the **Locale** entity, which is associated to Customer and LocalizedItems;
- the **Currency** entity, which represent te application available currencies.

These entities are instantiated and persisted by the controllers and the Data Access Objects. The localization of the fields of the Product entities is managed through the Translation Model.

4.2.2 Translation Model

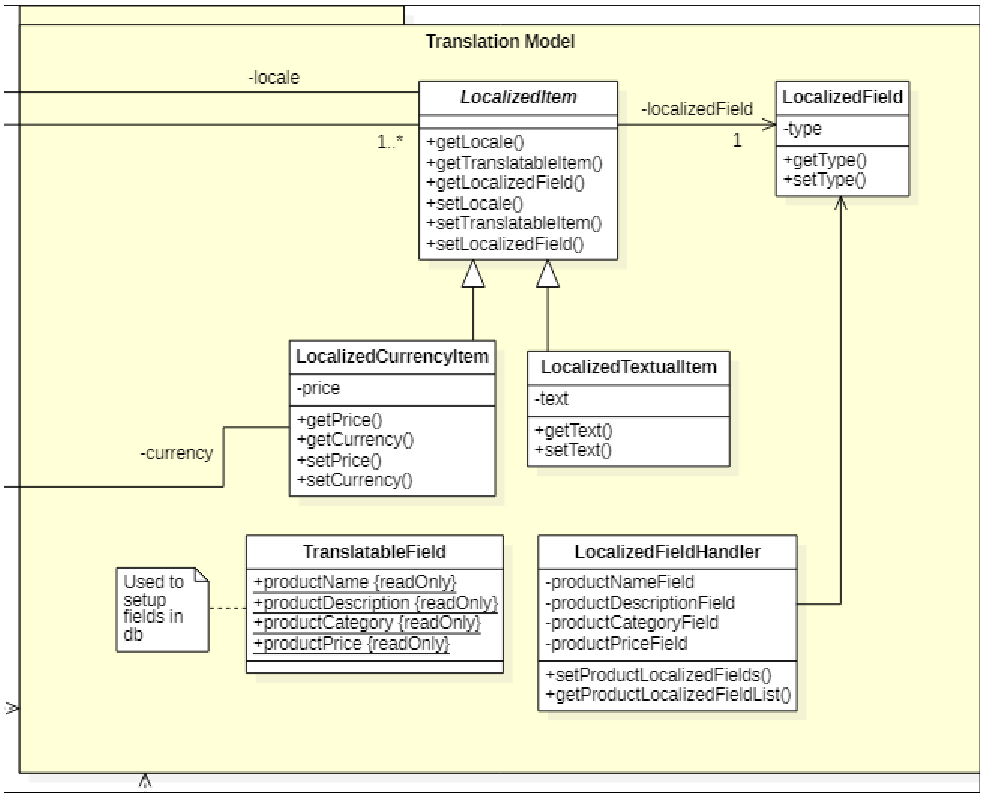


Figure 5: Translation Model Class Diagram

This package is used to manage the entity translations. Each entity that wants the localization feature for certain fields has to mantain an association with one or more **LocalizedItem**: this superclass is associated with a certain **Locale** and a certain **LocalizedField**.
The LocalizedField indicates for which fields the localization takes place (e.g product name, product description, etc.). Thus, a LocalizedItem is referred to a TranslatableItem, a Locale and a certain LocalizedField.
Two types of subclasses for the LocalizedItem have been created: **LocalizedTextualField** and **LocalizedCurrencyField**, which, respectively, can be used to represent a textual field or a price field (along with a currency).
The **TranslatableField** entity exposes static strings that can be used to persists the allowed localization fields.
The **LocalizationFieldHandler** entity can be used to manage the allowed localization fields: it will populate its fields with the persisted localized fields and expose them for entities that wants to translate a certain item.

4.2.3 Rest

The rest classes can manipulate the domain model entities through services that exposes functionalites to the application's clients. In particular, the controllers exposed by this layer offers:

- authentication interface: **AuthEndpoint**;

- product interfaces: **IntProductEndpoint** and **LocProductEndpoint**
- administration interfaces: **LocaleEndpoint**, **ManufacturerEndpoint** and **CurrenciesEndpoint**;
- customer interface: **CustomerEndpoint**;
- search interface: **SearchEndpoint** which uses the entity responsible for the product search (**Search Engine**)

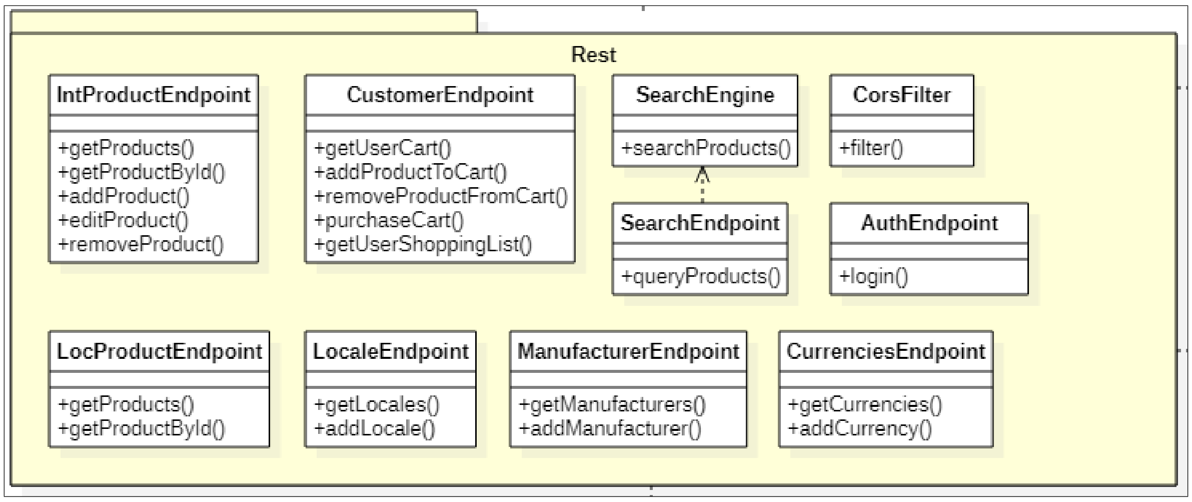


Figure 6: Rest Class Diagrams

The classes included in this package interact with the domain model entities using the Data Access Objects (DAO) exposed by the dedicated layer. These classes are exposed to the clients as REST interfaces, by usign the JAX-RS library. The information exchange between this layer and the clients is managed through JSON objects, which are created from base entities exposed by the Data Transfer Object layer.

Authentication and authorization are managed using **Json Web Token** (JWT). This type of token is self-contained and allow a stateless user authentication thanks to the signature emitted by a certificate authority. In particular, the authentication management in this layer is accomplished thanks to the annotation exposed in the Security layer.
JWT will be described more in detail in section **4.2.6**.

4.2.4 Data Access Objects

Data Access Object entites implement DBMS access and persistence functionalities, and offer them to the Controllers layer. The common functionalities between all DAO entities are contained in **BaseDao** and are the followings:

- Search entities by identifier
- Persist a new entity
- Update a persisted entity
- Delete a persisted entity

The other entities implent specific operations based on the referenced Domain Model entity.

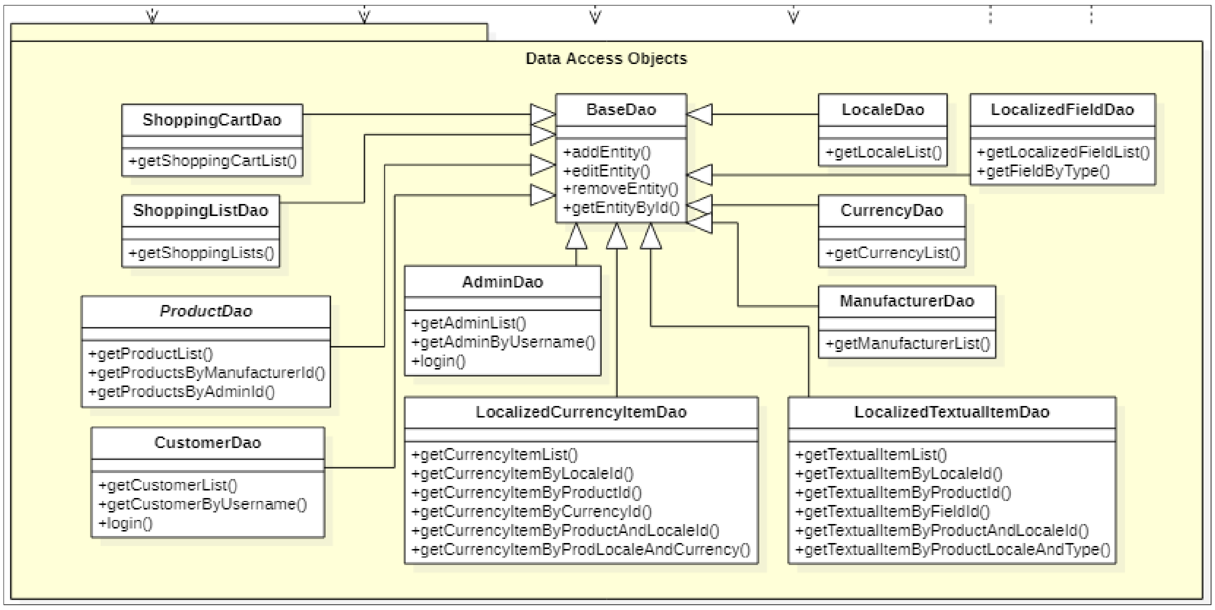


Figure 7: DAO Class Diagrams

4.2.5 Data Transfer Objects

Data Transfer Objects are POJO entities that allow to build object that can be sent or received on REST calls as JSON. They are used by some endpoints of the Controllers layer to manage input and response data. This layer has been used in order to abstract the modeling of messages between the Controllers layers and the clients from the Domain Model entities.

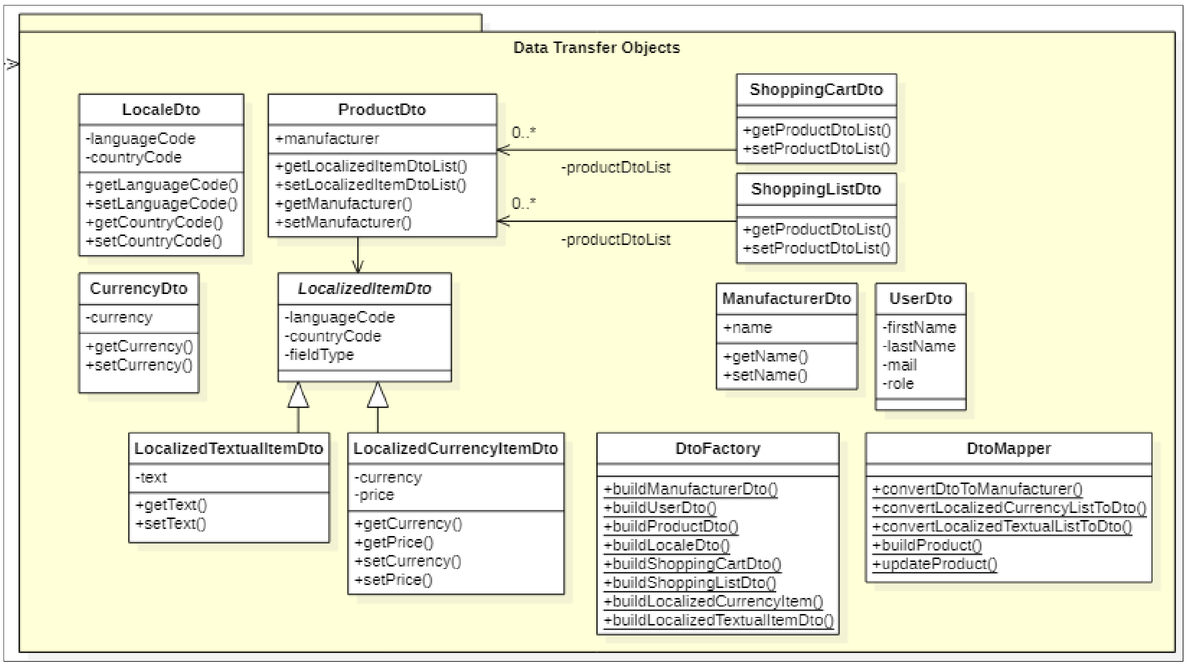


Figure 8: DTO Class Diagrams

4.2.6 Security

This layer implements authentication and authorization features by using Json Web Tokens (JWT). As said, JWT are self-contained tokens that enables stateless authentication thanks to the certificate signature: a tampering of the token will invalidate the signature.
A Json Web Token is composed of three parts:

- **Header**: Contains the signing algorithm;
- **Payload**: A set of keys/values which store application data. Keys are called *claims*;
- **Signature**: Created by passing header and payload to the signing algorithm.

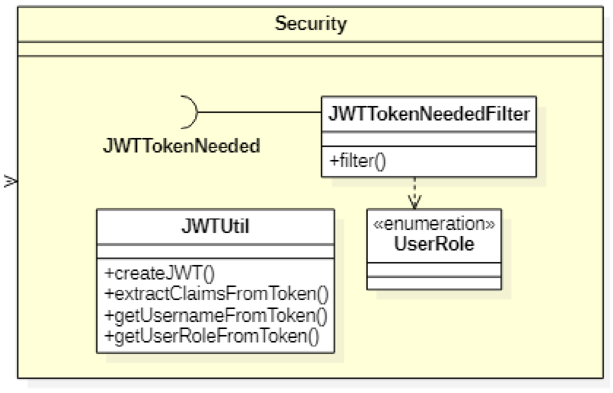


Figure 9: Security Class Diagrams

Each part of a JWT is encoded as Base64 and the three part are concatenated using the "." character.
Tokens are signed using HMAC + SHA256. For simplicity reasons, tokens emission and validation are handled by the application instead of using a separated authentication server.
These tokens have the following payload structure:

```
{
  "subject": "mario.rossi@example.com",
  "issuedAt": 1624223874036,
  "userId": 1,
  "userRole": "ADMIN",
  "lang": "en",
  "exp": 1624224474,
  "issuer": "i18n-store"
}
```

The user role has beeen included in the token claims in order to handle the authorization on exposed enpoints. This way, administration endpoint are available only for users with ADMIN role, while customer endpoints are available only for user with CUSTOMER role. A custom request filter binded to a custom annotation has been adopted in order authorize the user using the role included in the JWT.

Note: Tokens are not encrypted. They must be exchanged over an SSL/TLS layer to guarantee token encryption, for example by using HTTPS.

4.3 REST Endpoints

The following URI are the application's exposed endpoints.

- Authentication endpoint (**/api/auth/**):
  - **GET /api/auth/login**: User authentication, sent JWT to user if given username and password are correct
- Administration endpoints (accessible only if the logged user is an ADMIN):
  - **GET /api/users**: Show all the users
  - **GET /api/int-products**: Show all the internationalized products (along with all localizations)
  - **GET /api/int-products/{prodId}**: Show specified internationalized product (along with all localizations)
  - **POST /api/int-products**: Add an internationalized product
  - **PUT /api/int-products/id**: Edit an internationalized product
  - **DELETE /api/int-products/{prodId}**: Remove an internationalized product
  - **GET /api/locales**: Show all configured locales
  - **POST /api/locales**: Add a locale
  - **GET /api/manufacturers**: Show all manufacturer
  - **POST /api/manufacturers**: Add a manufacturer
  - **GET /api/currencies**: Show all currencies
  - **POST /api/currencies**: Add a currency
- Customer endpoints (accessible only if the logged user is a CUSTOMER):
  - **GET /api/products**: Show all products, localized in user locale
  - **GET /api/products/{prodId}**: Show specified product, localized in user locale
  - **GET /api/customer/{userId}/shopping-cart**: Show user shopping cart
  - **POST /api/customer/{userId}/shopping-cart/{prodId}**: Add specified product to user shopping cart
  - **DELETE /api/customer/{userId}/shopping-cart/{prodId}**: Removed specified product from the user shopping cart
  - **POST /api/customer/{userId}/shopping-cart/checkout**: Purchase all the product in shopping cart and move them to the shopping list
  - **GET /api/customer/userId/shopping-list**: Show user shopping list
- Search endpoints (**/api/search/**)
  - **GET /api/search/products/{query}**: Search products according to their name and description with a query (keyword separated by "+" character)
  - **GET /api/search/products/similar-to/{id}**: Search products similar to the one specified by the given identifier

4.4 Sequence Diagrams

Through sequence diagram, specific actions of an use case can be modeled in order to describe what messages are exchanged between the entities to accomplish the operation. The following sequence diagrams have been created:

- Add Product
- Show Product
- Add Product To Cart
- Purchase Products

These diagrams show the execution procedures for some operations of the specified use cases in section **4.1** and that can be implemented with the modeling described in section **4.2**.

4.4.1 Add Product

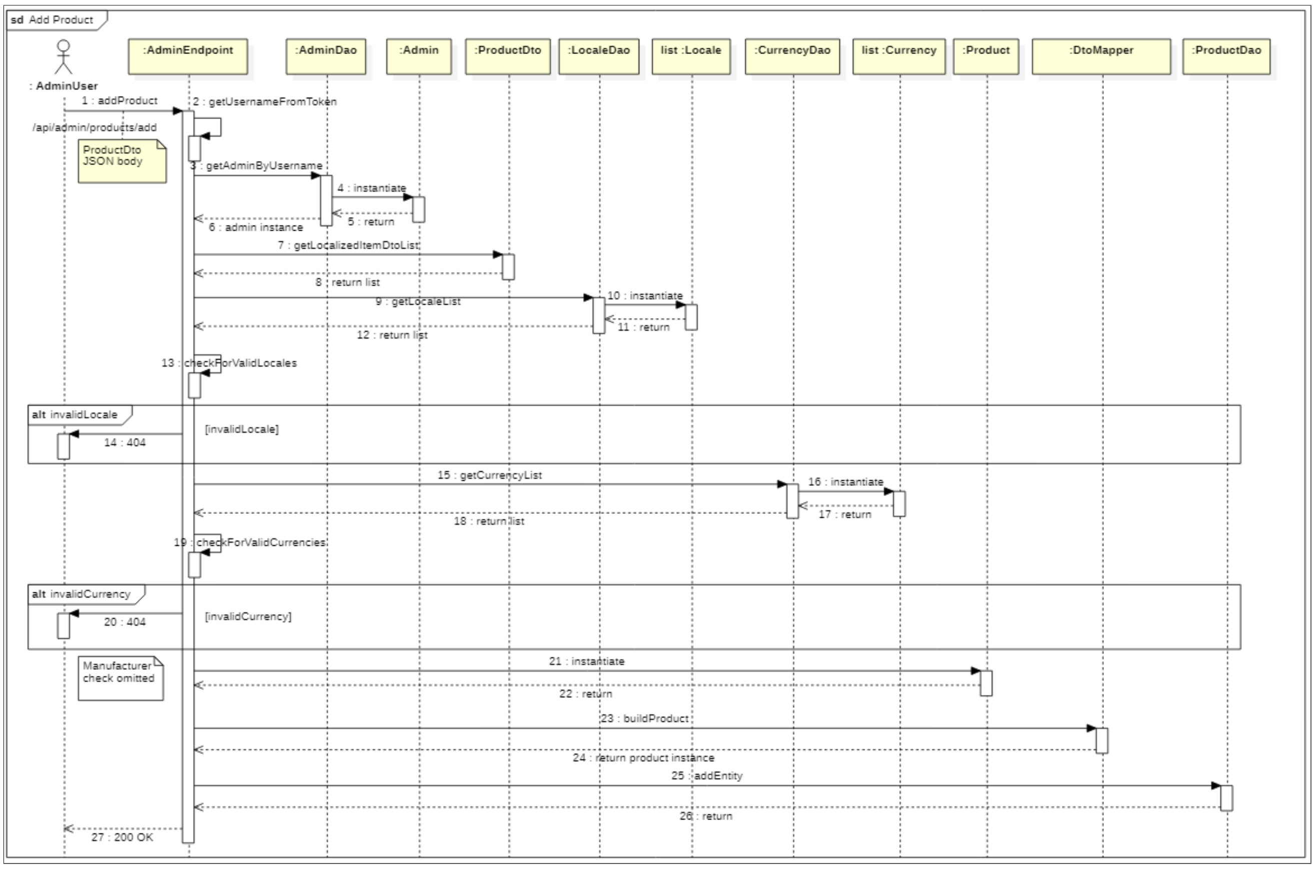


Figure 10: Add Product Sequence Diagram

4.4.2 Show Product

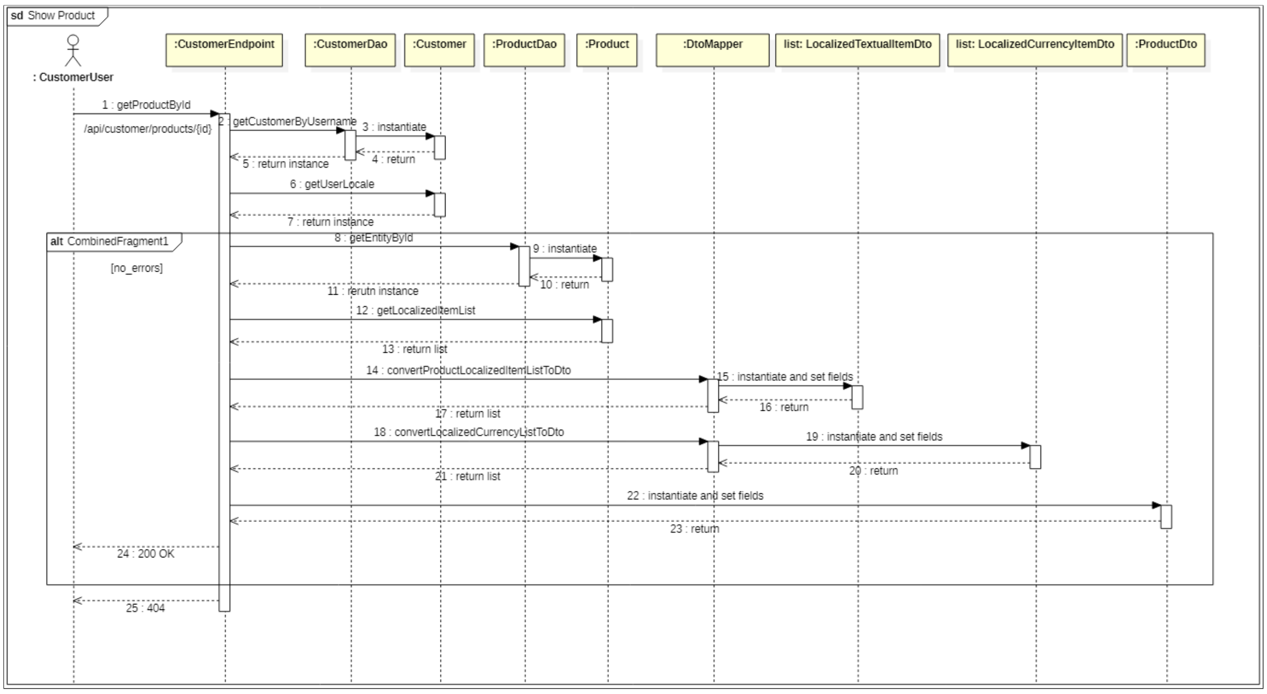


Figure 11: Show Product Sequence Diagram

4.4.3 Add Product To Cart

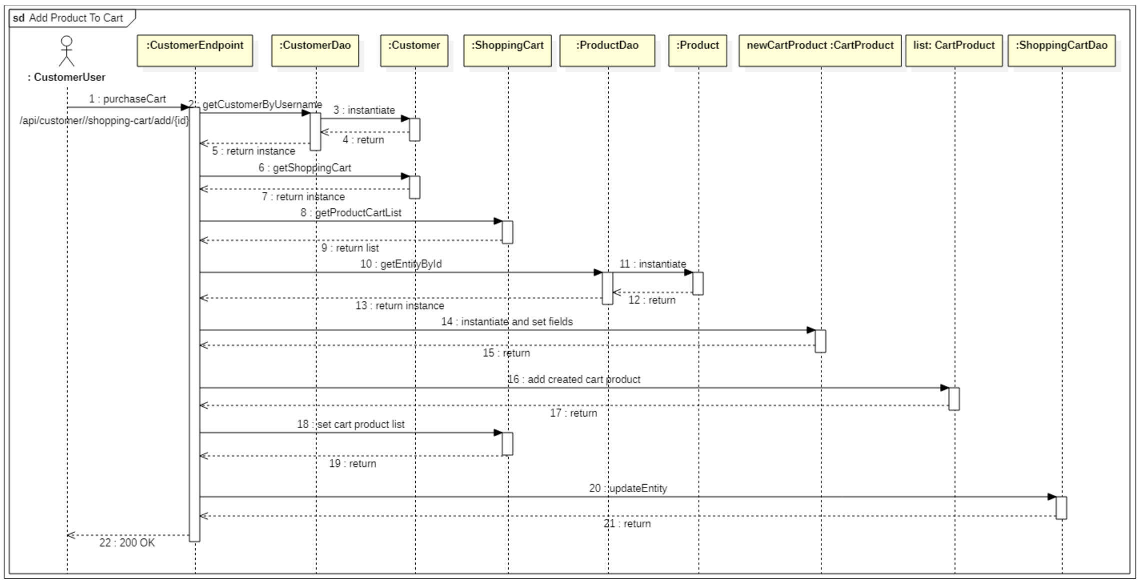


Figure 12: Add Product To Cart Sequence Diagram

4.4.4 Purchase Products

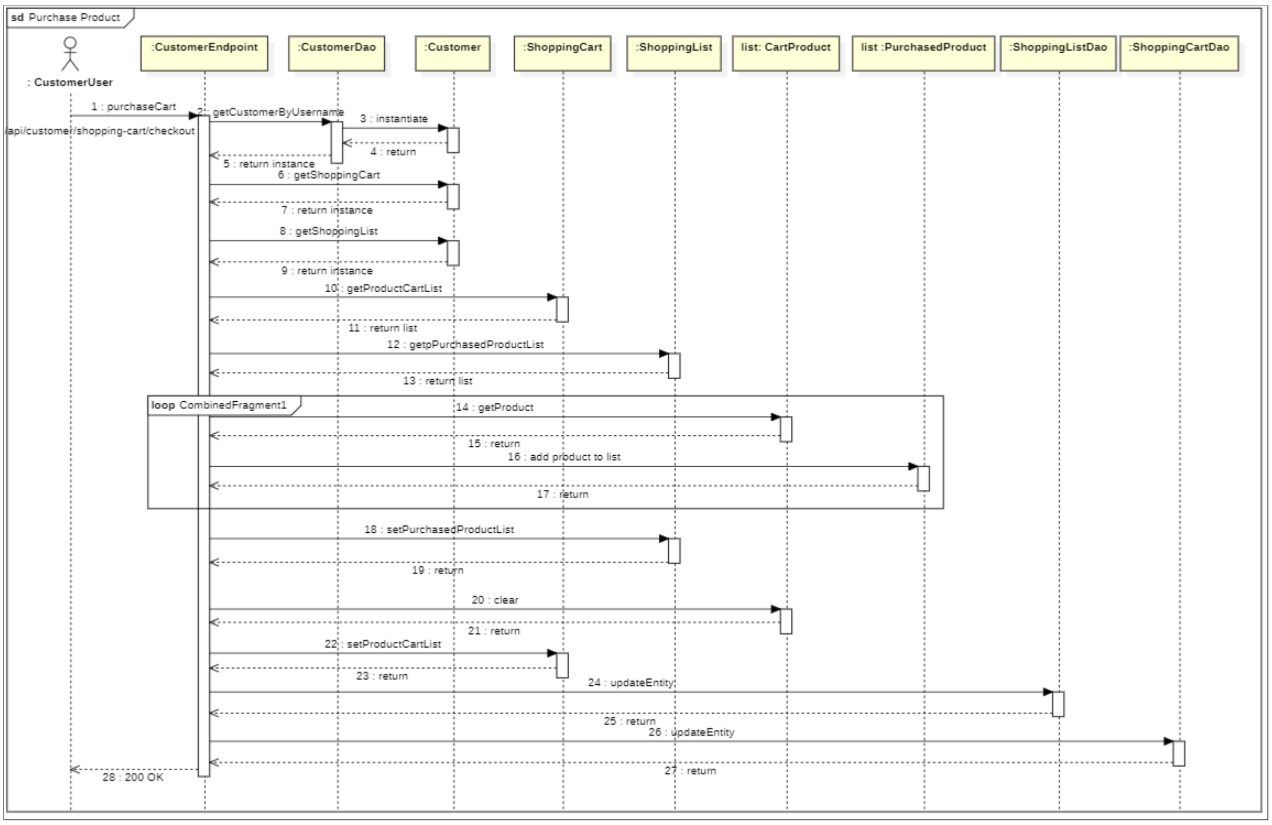


Figure 13: Purchase Products Sequence Diagram

4.5 ER Model

The following image shows the implementation of the relation database model on which the application is based on:

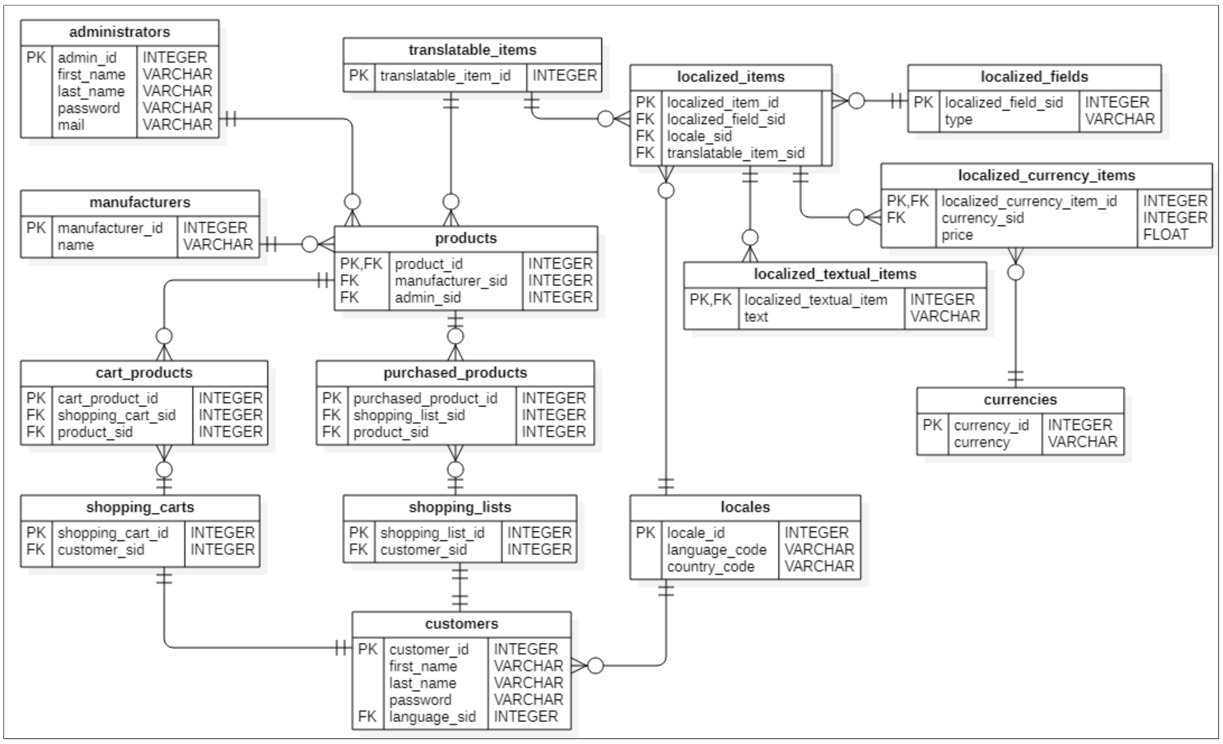


Figure 14: DB ER Model

The model is based on the following entities:

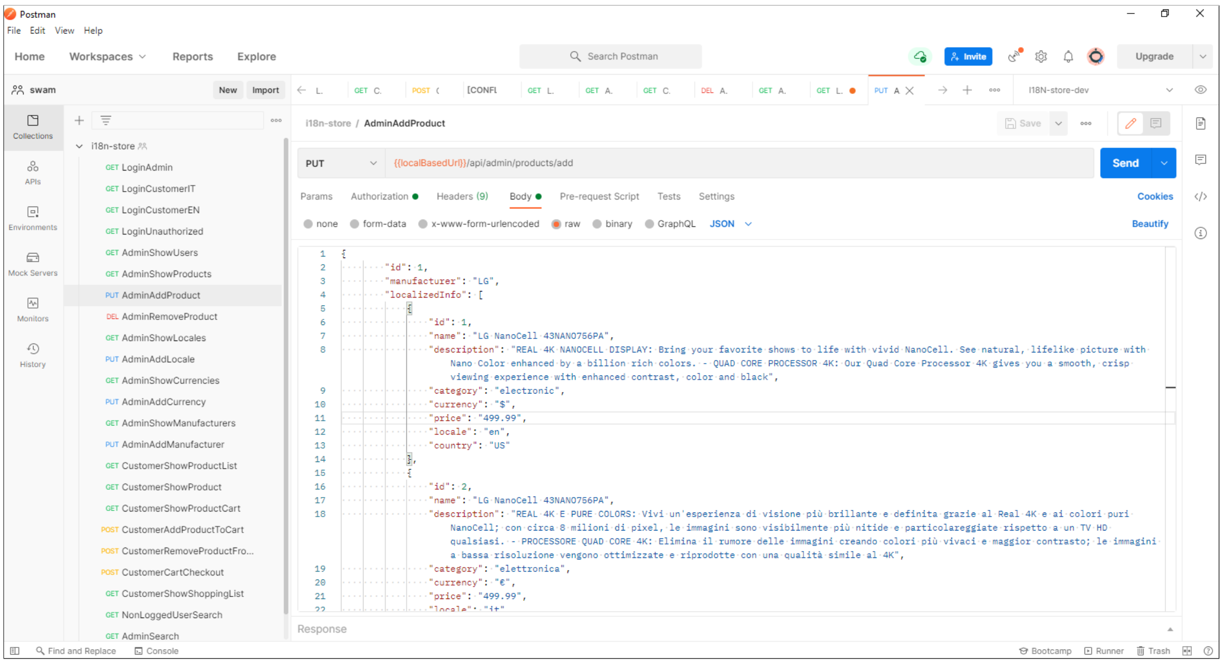
- **administrators**: admin users that manage the products as described in use cases **UC0**, **UC1** and **UC2**;
- **customers**: customer users, they manages a **shopping cart** and a **list of purchased products**. They are associated to a specific **locale**;
- **cart products**: associated to a specific **shopping cart** and to a specific **product**;
- **purchased products**: associated to a **shopping list** and to a **specific product**;
- **products**: available products, associated to a **manufacturer** and inserted by an **administrator**. This table associated to the **translatable items** table (Product's superclass);
- **localized items**: localizations are separated from the **translatable items** table and each localization is associated to a **locale** and a **localized field**. This way, if a locale is added to support the application, it will result in a new row in this table for each product, instead of new fields added in products table. Moreover, it is associated with **textual items** and **currency items** tables (subclasses). A currency item is associated to a certain **currency**, to express the product price.

5 Packages Testing

In order to verify and test the developed code, a test suite has been created. This suite allows to test the packages in an isolated way, using JUnit, and to execute integration test of all packages, using Postman.
Through JUnit, test cases have been created for the following packages:

- **Model**
- **Dao**
- **Dto**
- **Security**

Controllers tests and integration tests have been created through Postman. Thanks to Postman, a collection of REST calls has been created, in order to test all the exposed endpoints. Thus, correctness of returned values and error handling can be easily verified by running the collection.



PUT {{localBasedUrl}}/api/admin/products/add

```json
{
    "id": 1,
    "manufacturer": "LG",
    "localizedInfo": [
        {
            "id": 1,
            "name": "LG NanoCell 43NANO756PA",
            "description": "REAL 4K NANOCELL DISPLAY: Bring your favorite shows to life with vivid NanoCell. See natural, lifelike picture with
                Nano Color enhanced by a billion rich colors. - QUAD CORE PROCESSOR 4K: Our Quad Core Processor 4K gives you a smooth, crisp
                viewing experience with enhanced contrast, color and black",
            "category": "electronic",
            "currency": "$",
            "price": "499.99",
            "locale": "en",
            "country": "US"
        },
        {
            "id": 2,
            "name": "LG NanoCell 43NANO756PA",
            "description": "REAL 4K E PURE COLORS: Vivi un'esperienza di visione più brillante e definita grazie al Real 4K e ai colori puri
                NanoCell; con circa 8 milioni di pixel, le immagini sono visibilmente più nitide e particolareggiate rispetto a un TV HD
                qualsiasi. - PROCESSORE QUAD CORE 4K: Elimina il rumore delle immagini creando colori più vivaci e maggior contrasto; le immagini
                a bassa risoluzione vengono ottimizzate e riprodotte con una qualità simile al 4K",
            "category": "elettronica",
            "currency": "€",
            "price": "499.99",
            "locale": "it"
```

Figure 15: Postman

6 Full-Text Search

Product search and retrieval have been accomplished through integration of Hibernate Search functionalities. Using the version 5.10.7 of this library, the following functionalities have been implemented:

- Startup indexing
- Indexing annotations
- Standard Search
- Standard AND Search
- Fuzzy Search
- Phrase Search
- Similarity Search

These implementations will be briefly described in the following sections.

6.1 Startup Indexing

Hibernate Search index is used to perform fast searching of entities by storing keywords that belongs to certain documents. The index can be mantained in filesystem or can be temporary stored in memory. In both cases, at startup, the Hibernate Search indexer component must be initialized. This can be done as:

```
FullTextEntityManager fullTextEntityManager = Search
                              .getFullTextEntityManager(entityManager);
fullTextEntityManager.createIndexer().startAndWait();
```

This way, persisted entities that are not already indexed will be stored in the index and thanks to the automatic indexer the indexes will be kept in sync with the database objects.

6.2 Indexing annotations

Entities that should be considered for index building and retrieval must be annotated with Hibernate Search annotations.
An entity that should be indexed must be annotated with the keyword **@Indexed** and its searchable fields must be annotated with **@Field**. Moreover, entities can be defined to be embedded into an indexed entity with the keyword **@IndexedEmbedded**.
For example, the resulting indexed **LocalizedProduct** is the following:

```
@Entity
@Table(name = "localized_items")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class LocalizedItem extends BaseEntity {
    ...
    @Field(termVector = TermVector.YES)
    @Transient
    public String getText() { return null; }
    ...
}
```

the **TranslatableItem** entity is the following:

```
@Entity
@Table(name = "translatable_items")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class TranslatableItem extends BaseEntity
{
    ...
    @IndexedEmbedded
    @OneToMany(mappedBy = "translatableItem", cascade = CascadeType.ALL)
    private List<LocalizedItem> localizedItemList;
    ...
}
```

and the resulting **Product** entity is the following:

```
@Indexed
@Entity
@Table(name = "products")
public class Product extends TranslatableItem {
    ...
}
```

6.3 Standard (AND) Search

Search can be performed by creating a query end execute it to retrieve matching entity instances. Query creation can be accomplished by specifying the indexed entity and the fields that should be used for the search. A query can be built as:

```
QueryBuilder qb = fullTextEntityManager.getSearchFactory()
                .buildQueryBuilder().forEntity(Product.class).get();

Query query = qb.simpleQueryString()
                .onFields("localizedItemList.text")
                .matching(matchQuery)
                .createQuery();
```

This will create a query that will search on *localizedItemList* text field for matching keyword specified in *matchQuery* string. This variable is a space-separated list of words. With this query, an entity instance will be retrieved if one of its fields contains at least a keyword contained in *matchQuery*.
If all keywords must be contained in a field to declare the match, the *withAndAsDefault-Operator()* method can be used.
Once the query has been created, the matching entities can be retrieved as:

```
javax.persistence.Query persistenceQuery =
       fullTextEntityManager.createFullTextQuery(query, Product.class);

List<Product> productList = persistenceQuery.getResultList();
```

6.4 Fuzzy Search

Sometimes, keyword exact matching is not the most appropriate way of searching. For example, if a user search for a product and introduces one or more typos in keywords, the correct object could not be retrieved. Even if no typos are introduced, this type of matching could not be satisfying: a user does not know the exact keywords that are included in object's fields.
Fuzzy queries can solve this problem by allowing for approximate matching using the Levenshtein distance algorithm. A fuzzy query can be built as:

```
Query query = qb.keyword()
                .fuzzy()
                .withEditDistanceUpTo(editDistance)
                .withPrefixLength(prefixLength)
                .onFields("localizedItemList.text")
                .matching(matchQuery)
                .createQuery();
```

where *editDistance* specify how much a term can deviate from the other and *prefixLength* define the length of prefix that should be ignore when evaluating the distance.

6.5 Phrase Search

This type of search allows to search for exact or approximate **sentences** in object's fields.

```
Query query = qb.phrase()
                    .withSlop(slop)
                .onField("l"ocalizedItemList.text")
                .sentence(matchQuery)
                .createQuery();
```

where *slop* is the number of other words that can be contained in the specified phrase.

6.6 Similarity Search

Similarity Search can be accomplished through *More Like This* (MLT) queries. This type of query will return all the entities which fields are similar to the one with the requested identifier. The query can be built as:

```
Query query = qb.moreLikeThis()
                .excludeEntityUsedForComparison()
                .favorSignificantTermsWithFactor(2f)
                .comparingField("localizedItemList.text")
                .toEntityWithId(objectId)
                .createQuery();
```

where:

- *excludeEntityUsedForComparison()* can be used to exclude from the returned entities the one used for comparison;
- *favorSignificantTermsWithFactor(float)* can be used to give an higher score to the very similar entities and downgrade the score of mildly similar entities

7 Application frontend

A demonstrative frontend application has been developed in order to be able to easily interact with the backend and to show its functionalities.
The frontend application allows to:

- Log in the user, with an administrator or customer account;
- Show available products, users, manufacturers, locales and currencies
- Purchase products through the use of a shopping cart
- Add, edit and remove localized products
- Search for products

Moreover, the frontend has been internationalized through the use of third party libraries: each label in the frontend is available in multiple languages. The language choice is based on the user configured locale.
The frontend has been developed using React library and its state is managed through Redux.

7.1 React

React is a Javascript library designed for building user interfaces.
A React application is built of smaller entities, called **Components**. Components are stateful and they can communicate by passing data to each others, which is called prop. Prop and state management is done by React using the Component lifecycle. Moreover, it offers various hooks that developers can use to handle component life scenarios.
The user interface can then be built by grouping together smaller Components. Through the use of components, code reusability and management can be greatly improved.

7.2 Redux

Redux is a Javascript state container. It is particularly useful when used with React in order to manage components' props, which can grow and their managing can become difficult as the application gets bigger.
Redux is composed of four basic components: **centralized state**, **actions**, **reducers** and **dispatchers**.
React components can register to the Redux store to either receive updates of the centralized state or to update it through the use of actions. Once an action is fired by a component, the reducers will update the affected field of the centralized state following the developed logic. After an update, the dispatcher will send the changes to the registered components.

7.3 I18next library

I18next and react-i18next libraries have been choiced to handle the internationalization of the frontend application.
The frontend will receive products already localized in the user language (or in all languages if the user is an administrator), but all the frontend labels need to be localized in

the user locale.
Thanks to these libraries, a React component can be defined in order to define (or load) localizations for a particular label and in a given locale. These labels can then be reused by other components in order to display text in the configured language, which can be changed programmatically.
The user locale is detected by the frontend by reading the *lang* claim in the exchanged Json Web Token.

7.4 Frontend pages

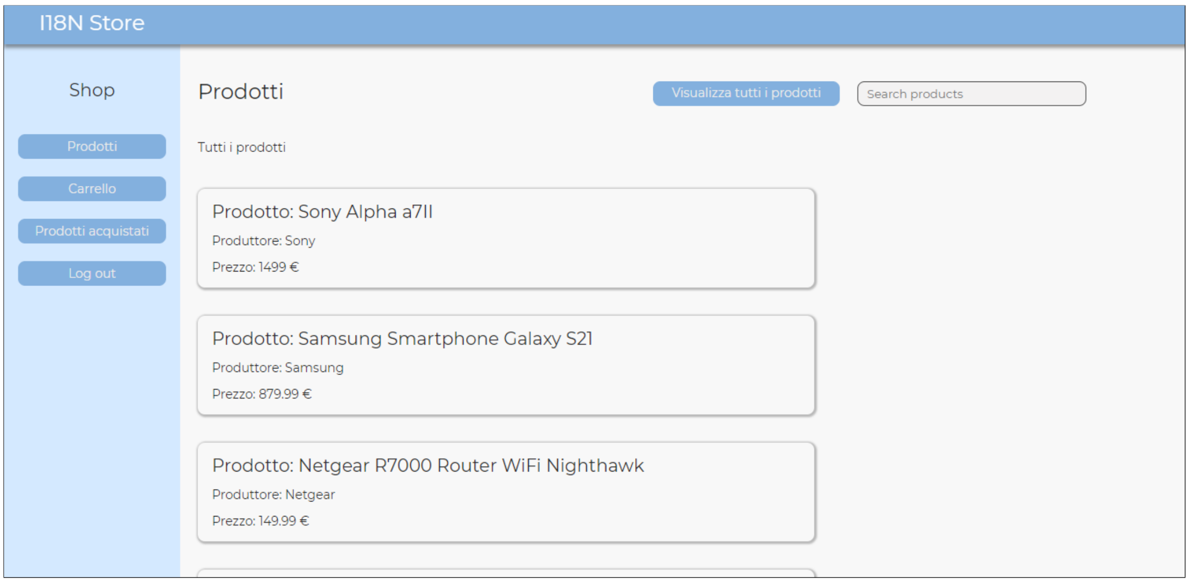
The following sections describe the developed components of the web interface

7.4.1 Customer section

Once a customer user is logged, the related pages will be shown to the user. This section includes the following pages:

- Products: this page displays all the available products. By clicking on a product, the product informations page will be shown. This page shows the product localized in the customer locale;
- Shopping Cart: this page displays all the product added to the cart by the user and allows to purches them;
- Shopping List: this page displays all the purchased products.

This pages can be freely navigated from the menu.



Prodotti

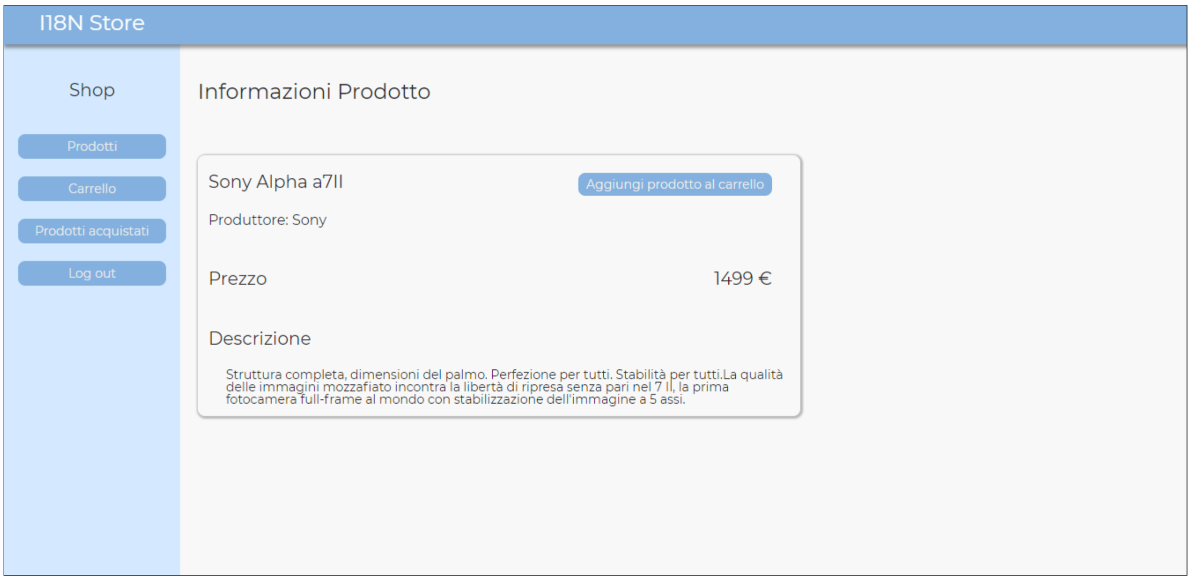
Tutti i prodotti

Prodotto: Sony Alpha a7II
Produttore: Sony
Prezzo: 1499 €

Prodotto: Samsung Smartphone Galaxy S21
Produttore: Samsung
Prezzo: 879.99 €

Prodotto: Netgear R7000 Router WiFi Nighthawk
Produttore: Netgear
Prezzo: 149.99 €

Figure 16: Italian localized customer home page



Informazioni Prodotto

Sony Alpha a7II

Produttore: Sony

Prezzo 1499 €

Descrizione

Struttura completa, dimensioni del palmo. Perfezione per tutti. Stabilità per tutti.La qualità delle immagini mozzafiato incontra la libertà di ripresa senza pari nel 7 II, la prima fotocamera full-frame al mondo con stabilizzazione dell'immagine a 5 assi.

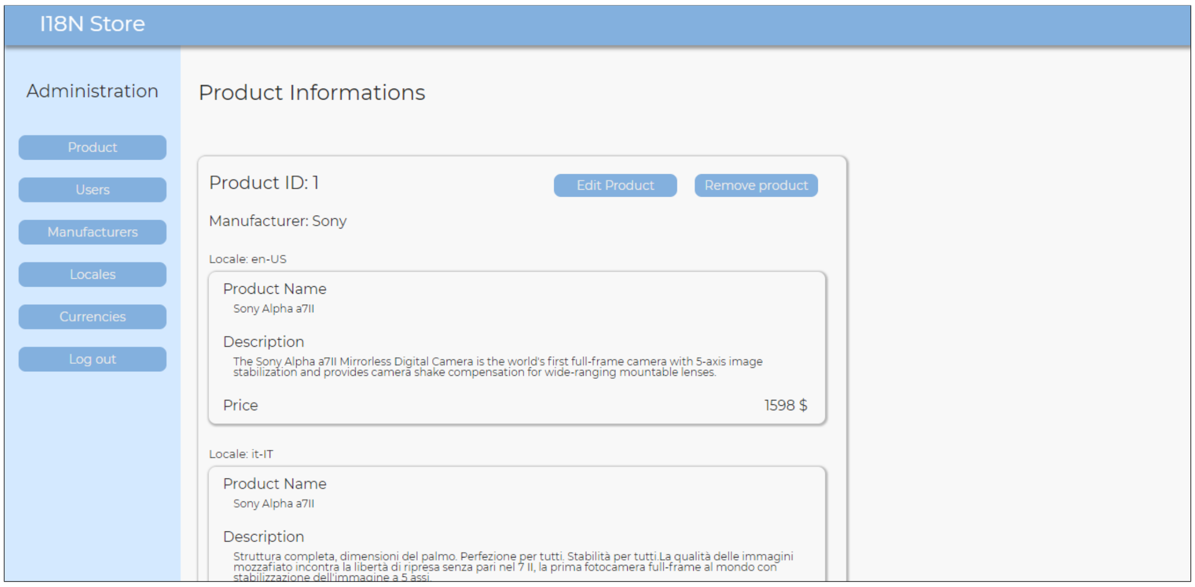
Figure 17: Italian localized customer product info page

7.4.2 Administration section

An administrator user can log in the administration section, which includes the following pages:

- Products: same as the customer product page. The product info page shows the product and all its configured localizations;
- Users, Locales, Manufacturers, Currencies: these pages shows the respective configured values;

Note: An administrator user does not have an associated locales, so all the administration pages will be always localized in a default language (english).



Product Informations

Product ID: 1

Manufacturer: Sony

Locale: en-US

Product Name

Sony Alpha a7II

Description

The Sony Alpha a7II Mirrorless Digital Camera is the world's first full-frame camera with 5-axis image stabilization and provides camera shake compensation for wide-ranging mountable lenses.

Price 1598 $

Locale: it-IT

Product Name

Sony Alpha a7II

Description

Struttura completa, dimensioni del palmo. Perfezione per tutti. Stabilità per tutti.La qualità delle immagini mozzafiato incontra la libertà di ripresa senza pari nel 7 II, la prima fotocamera full-frame al mondo con stabilizzazione dell'immagine a 5 assi.

Figure 18: Administration product info page

7.5 A brief comparison between React and Angular

React and Angular are two of the major frameworks that allow to build web interfaces with an optimal level of code mantainability and separation of concerns.
Angular framework allows to implement an application that respect the Model View Controller (MVC) pattern by leading the developer to follow a code structure that implement this pattern. This is done by defining:

- **the data model** on which the application will operate;
- **the views**, which will display (parts of) the instance of the data model;
- **the controllers**, which will read/write the instance of the data model.

Angular uses TypeScript as its main language. An Angular application is built of **components**, which are splitted in three main parts:

- **HTML file**: typical html file that define the visual structure of the component (view). HTML files can include other components and variables can be passed to child component through *binding*. Angular supply proprietary keywords that can be used in HTML to ease some operations, such as *ngFor* and *ngIf*. It can grab the class state fields through the use of **{**{}**}** operator;
- **TS file**: TypeScript file that is responsible to handle the logic of the component. It defines a class that mantains a state and has methods to use services and/or to handle the view callbacks;
- **CSS file**: define the style for the component

Moreover, Angular allows to define **services**, that allow to operate on the data (typically by fetching/posting from/to REST API). Tha data model is defined in Typescript files that export the related interfaces.

React, as explained before, lead to a slightly different approach: it is a library focused on building of user interfaces. React allows to define components that render a view which can use/update the component state and that follow the React component's lifecycle. A more sophisticated (and mantainable) use of React can be achieved by adding the Redux library to a React application. Through Redux, the MVC pattern can be implemented: Redux allows to define a centralized store which mantain a state that can be modified/dispatched by using *(*actions) and *(*dispatchers). Components can register to the fields of the centralized state to receive their updates, and can use actions to modify them.
The following sections briefly describe how the implementation of a simple component (such as a simplified version of our shopping list) differs between Angular and React Redux.

7.5.1 A React Redux shopping list

This is how a simplified shopping-list.jsx file would look like this:

```
/* Imports are omitted... */
/* This mapping allow to use the Redux actions by receiving them as props */
function mapDispatchToProps(dispatch) {
  return {
    getShoppingList: () => dispatch(getShoppingList())
  };
}
/* Register to the centralized state field update.
   Updates will be received as props */
const mapStateToProps = (state) => {
  return {
    shoppingList: state.getters.customer.shoppingList,
  };
};
/* The shopping list component */
class ShoppingList extends React.Component {
  constructor(props) {
    super(props);
  }

  /* Hook called when the component is mounted (inserted in DOM tree). It
     will call the action */
  componentDidMount() {
    this.props.getShoppingList();
  }

  /* The render method is called when the component is mounted (see above) and
     whenever the props or the component state change */
  render() {
    /* The actual view */
    return (
      <div className="shopping-list-container">
        <div className="shopping-list__title">
          Shopping list
        </div>
        /* Iterate on shopping list array and render a
           product card component for each item */
        {this.props.shoppingList.products.map((p, i) =>(
          /* This is how variables can be passed to child
             components. The child can access them as props */
          <ProductCard
            prodId={p.id}
            name={p.name}
            manufacturer={p.manufacturer}
            price={p.price}
          />
        ))}
      </div>
    );
  }
}
```

The simplified version of the ProductCard component is the following:

```
class ProductCard extends React.Component {
  ...
  render() {
    return (
      <div className="product-card"
        <div className="product-card__name">
          /* This is how outer variables can be accessed inside the view */
          Product Name : {this.props.name}
        </div>
        <div className="product-card__manufacturer">
          Manufacturer : {this.props.manufacturer}
        </div>
        <div className="product-card__price">
          Price : {this.props.price}
        </div>
      </div>
    );
  }
}
```

The simplified getShoppingList action is the following:

```
/* Imports are omitted... */
export function getShoppingList() {
  return function (dispatch) {
    let purchasedProducts = [];
    const url = "api/shopping-list";
    /* Axios allows to async fetch the api */
    return axios.get(url)
      .then(result => {
        let sl = result.data;
        /* Iterate through the received data */
        sl.purchasedProducts.map((p) => {
          /* Payload processing omitted */
          let product = {
              id: p.id,
              manufacturer: p.manufacturer,
              name: productName,
              price: price + " " + currency
          };

          purchasedProducts.push(product);
        });

        let payload = { products: purchasedProducts }
        /* Dispatch the object to the store */
        dispatch({ type: GET_SHOPPING_LIST, payload })
      })
      /* Error catching omitted */
  }
}
```

7.5.2 An Angular shopping list

This is how an Angular counterpart shopping list component looks like:

```
/* Imports are omitted... */
/* Component decorator is omitted... */
export class ShoppingListComponent implements OnInit {
  products: ProductInfo[] = [];

  constructor(private shoppingListService: ShoppingListService) { }

  /* Called when the component is initialized */
  ngOnInit(): void {
    this.getShoppingList();
  }

  getShoppingList(): void {
    /* Subscribe to get the async callback that return the list
       of products */
    this.shoppingListService.getShoppingList()
      .subscribe(products => this.products = products)
  }
}
```

This is the view of the ShoppingListComponent:

```
<div class="shopping-list-container">
  <div class="shopping-list__title">
    Shopping List
  </div>

  /* ngIf can be used to render html blocks if conditions are met */
  <div *ngIf="products">
    /* ngFor allows to iterate on component state fields */
    <div *ngFor="let product of products">
      /* Pass the product to the ProductCard component (binding) */
      <app-product-card [product]="product"></app-product-card>
    </div>
  </div>
</div>
```

A simplified product card component can look like this:

```
/* Imports are omitted... */
/* Component decorator is omitted... */
export class ProductCardComponent implements OnInit {
  /* Define the type of binded input field */
  @Input() product?: ProductInfo;

  constructor() { }

  ngOnInit(): void {
  }
}
```

Its view can look like the following:

```
<div class="product-card">
  <div class="product-card__name">
    Product Name: {{product.name}}
  </div>
  <div class="product-card__manufacturer">
    Manufacturer: {{product.manufacturer}}
  </div>
  <div class="product-card__price">
    Price: {{product.price}}
  </div>
</div>
```

Thanks to TypeScript, we can define the data model:

```
export interface ProductInfo {
    id: number,
    name: string,
    manufacturer: string,
    price: string
}
```

The shopping list service is the following:

```
/* Imports are omitted... */
/* Angular allows to use dependency injection
   to automatically inject the service dependency
   in interested components */
@Injectable({
  providedIn: 'root'
})
export class ShoppingListService {
  private shoppingListUrl = 'api/shopping-list';
  constructor(private http: HttpClient) { }

  /* Return an Observable list of ProductInfo array (async).
     A component subscribed to this observable will be notified
     when the value is returned */
  getShoppingList(): Observable<ProductInfo[]> {
    const url = 'api/products'
    return this.http.get<ProductInfo[]>(url)
      .pipe(
        catchError(this.handleError<ProductInfo[]>(/* Error handling omitted */)
      );
  }
}
```