Booktique

Booktique is a system that implements an online book store.

Repository

The source code is available on GitLab at
https://gitlab.com/GiorgiaAuroraAdorni/3rdAssignment/.

Contributors

This project has been developed by Giorgia Adorni (806787).

Installation

```
$ git clone https://gitlab.com/GiorgiaAuroraAdorni/3rdAssignment.git
$ cd booktique
$ docker-compose up
```

An alternative to `docker-compose up` is the command `docker-compose up --build` that allows the rebuild of the app.
The database is available at http://localhost:5432/, (you can find the necessary credentials to access to pgAdmin and interact with the model in the `docker-compose.yml`).

Docker Compose also allows to locally run the unit tests:

```
$ docker-compose run app mvn test
```

It is also possible to generate the API References using Javadoc:

```
$ mvn javadoc:javadoc
$ mvn javadoc:test-javadoc
```

Containerization

The application is composed of two main components:

- **app:** the Java application, developed using the Spring Boot Maven plugin, that provides many convenient features such as the built-in dependency resolver, and the ORM Hibernate.
- **database:** a PostgreSQL instance responsible for persistently storing the user's data.

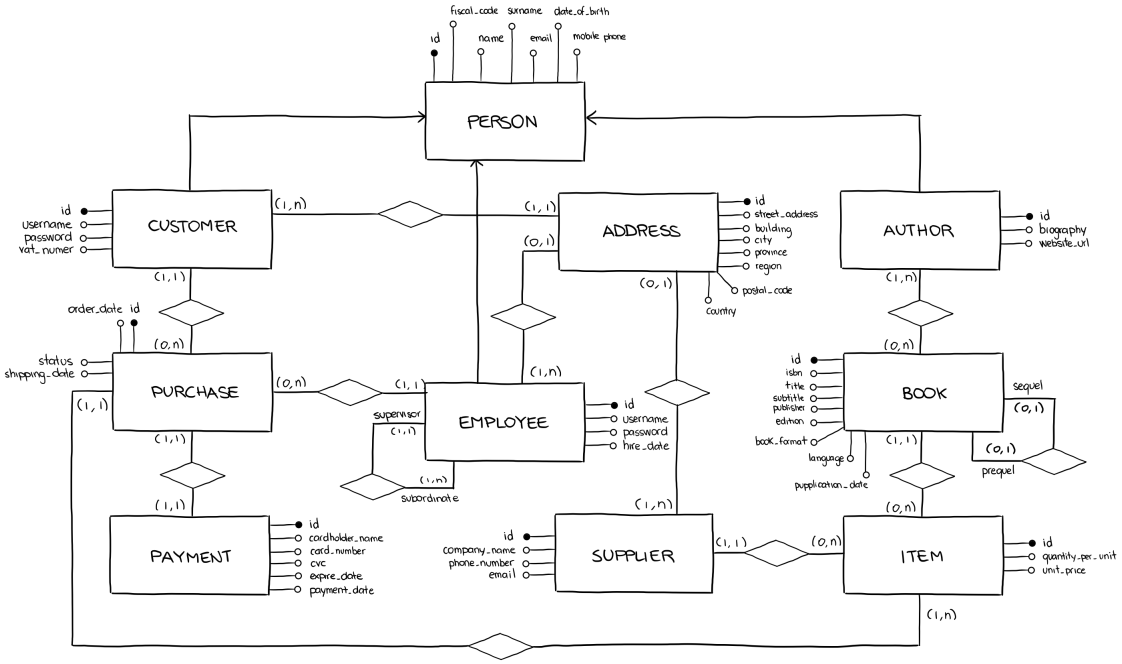
No web interface has been created.

Docker is used to containerize each of the two components, allowing to simplify the setup process of the development environment creating images that contain all dependencies needed.

The Docker-compose produces three images: DB, pgAdmin and the application.

Model

Booktique is a system that implements an online book store. The basic application involves querying books according to the fields ISBN, title, author etc. We support services for buying books. We build a personal profile page which is used for handling customer orders. We implement a ”recommendation system” for recommending related books (prequels and sequels) based on those searched and purchased.



Entities

The model is composed of 10 entities:

- **book** - items available in the online store. The book object provides a set of information such as the title, ISBN, authors, publisher, format etc. A unique id is used as the primary key, despite this, the ISBN provides a perfect primary key because every book has a different ISBN.
- **person** - a real person. The person object provides personal information such as name, surname, fiscal code, date of birth, email and telephone number. A unique id is used as a primary key, in addition, the fiscal code is marked as a natural identifier as it identifies a person in the real world.
- **author** - the author of the book. Because an author is also a person, the object inherits

the same attributes of the person superclass. More information about the author is provided, for example, an optional biography and the web site URL.
- **customer** – people who buy books in the online store. Every customer has a basic set of data inherits from the person superclass such as for the authors. Mandatory attributes, such as login credentials and postal address, is added to customers but can be also provided a VAT number.
- **employee** – is a person who carries out activities related to orders. An employee has the same properties of its parent class, in addition, he has associated with another employee as a supervisor. More information about the employee is given, such as the login credentials, the hire date and his postal address.
- **supplier** - organization that provides books to the online store. To the supplier object are associated a unique id as primary key, the company name and other information such as email, telephone number and postal address.
- **address** - postal information about customer and supplier. In this object is stored the street address, the postal code, the city, region and country name.
- **item** – articles selected by the customer for purchase in online store. To each item is assigned the reference to the book in the catalogue that the customer wants to buy, its unit price, the desired quantity and its supplier.
- **purchase** - items ordered. To every purchase is associated a hypothetically unlimited number of items, the customer, the employee who takes charge of the order, order and shipping dates, the total amount, order status and the transaction information (payment type and date).
- **payment** - the transaction information associated to all the purchases. Every payment details provide information about cardholder, the card used and the date of the payment.

Relationships

Between those entities there's 15 relations, in particular:

- two **self-relations**:
    - the first one is a *One-to-One bidirectional* relation between a book and his prequel, in one direction and in the other between the book and his sequel;
    - the second is a *Many-to-One* relation between employees and supervisors (each employee is associated with only one supervisor).
- one relation **Many-to-Many** between books and authors, handled with *lazy loading*.
- one **inheritance hierarchy** that involves 4 entities: the class person serves as a superclass for the employee, author and customer subclass.

All entities will also inherit from an Auditable abstract class that provides the `createdDate` and `modifiedDate` attributes using **JPA Auditing**. This allows to tracking changes to the entities made from the Java application.

Project architecture and responsibility

The application consists of two parts:

- `booktique/src/main` contains the source code files of the application:
    - **BooktiqueApplication** is the class that serves as the entrypoint of the app.

    - `model` contains the models used by the application. The domain models are the java classes that are mapped to the corresponding tables in the database. Every model is decorated with Hibernate annotations in order to perform operations on the relational database.
    - `repository` contains repository interfaces for all models, excluding abstract classes. *Spring Data JPA* create automatically a repository implementations from the repository interface. Extending **JpaRepository** every repository inherits several methods for working with entity persistence, including methods that implement CRUD operations such as save and delete, but also for entity search operations.
      Spring Data JPA also allows defining other custom query methods by simply declaring their method signature. For all models, customized `findBy{...}()` methods have been implemented. For example in the case of **BookRepository** the `findByAuthors_Name()` method.
    - `utility` contains useful classes and interface for operating on associations and entities:
        - **Associations** a class containing a static method that returns `true` if two associations are equal to each other.
        - **EntityEqualsByAttributes** an interface that compares two instances and returns `true` if the entities are equal to each other.
        - **EntityToDict** an interface the trasforms an entity in a dictionary.
- `booktique/src/test` contains the JUnit tests that exercise all the functionalities of the Java application. The package has the same structure as the source code directory:
    - **BooktiqueApplicationTest** is the test class the verify the correct start of the application.
    - `model` contains a factory interface **EntityFactory** used by the test suite to creates instances. In particular implement the `createValidEntity()` method that creates a default instance, and the `createValidEntities()` method the creates a list of valid instances.
      For each domain class, there's a factory that implements the **EntityFactory** methods. All the factories are instantiated using default values for all the fields. In order to respect the unique constraint in case of the creation of multiple instances is used an index as param for the methods.
    - `repository` contains tests on repositories and on the associated models. In particular, for every model are defined:
        - tests for the proper functioning of the CRUD operations implemented by the repository, like insertion, updating, deleting and reading of the data;
        - tests for the proper functioning of the search operations implemented by the repository and the customized one;
        - tests on the associations between the various models, in particular on cascade policies;
        - tests that throw exceptions in case you try to insert data that violates the domain rules.
    - `utility` contains the definition of two useful assertions:
        - **assertAssociationEquals** that *asserts* that the expected set of entities and the

actual ones are equal. Each of the entity sets is the attribute that identifies the association between entities;

- **assertAttributesEquals** that *asserts* that expected attributes of an entity and the actual one are equal.