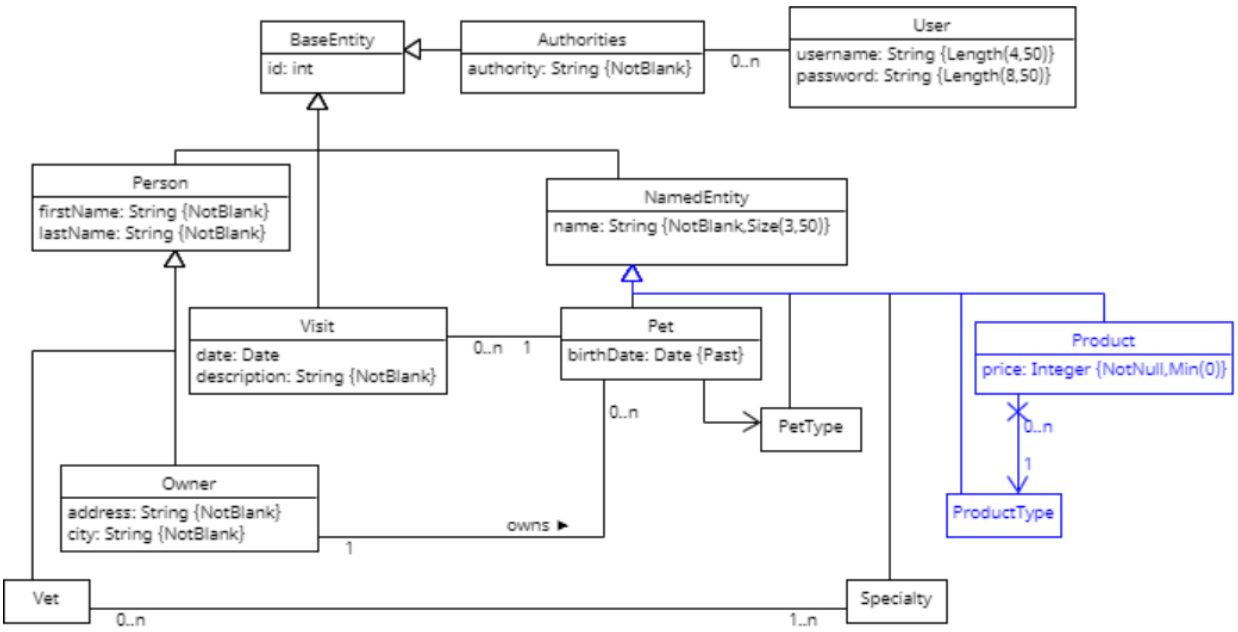
Simulation of the lab control-check of DP1

Introduction

In this exercise, we will add the functionality of managing products that will be sold in our pet clinic. To do so, we will carry out a series of exercises based on functionalities that we will implement in the system, and we will validate them through unit tests. If you want to see the result of the tests, you can run the "*mvnw test*" command in the root folder of the project. Each test successfully passed will be worth one point.



We will carry out a series of exercises based on functionalities that we will implement in the system, and we will validate them through unit tests. If you want to see the result of the tests, you can run them (either using your favorite development environment, or using the "*mvnw test*" command in the project's root folder). Each test successfully passed will be worth one point.

To start the control check, you must accept the task of this practical control via the following link:

https://classroom.github.com/a/t-_ze9ld

By accepting such a task, an individual unique repository will be created for you, you must use that repository to solve and submit the practical control. Moreover, you must submit the activity in EV associated with the check control by providing the url of your personal repository as text. Remember that you must also submit your control solution.

The submission of your solution to this control should be done using a single "*git push*" command to your individual repository. Remember to push before logging out of the computer and leaving the classroom, otherwise your attempt will be assessed with a 0. Your first task in this control will be to clone (remember that if you are going to use the classroom computers to perform the control you will need to use a GitHub authentication token as a key, you have a configuration help document in the control's own

repository). Next, you'll need to import the project into your favorite development environment and begin the exercises listed below. When importing the project, it may have compilation errors. Don't worry, if they exist, these errors will disappear as you implement the different exercises.

Note 1: Do not change the class names or the signature (name, response type, and parameters) of the methods provided as the base material for the control. The tests used for assessment depend on the classes and methods having the structure and names provided. If you modify them, you probably won't be able to make them pass, and you'll get a bad grade.

Note 2: Do not modify the unit tests provided as part of the project under any circumstances. Even if you modify the tests in your local copy of the Project, they will be restored using a git command prior to the execution of the tests for computing the final grade, so your modifications of the tests will not be taken into account.

Important note 3: As long as there are unsolved exercises, there will be tests that do not pass and, therefore, the *"mvnw install*" command will end with an error. This is normal because of the way the control is set up, and you don't have to worry about it. If you want to test the application, you can run it in the usual way even if "mvn install" ends with an error.

Important note 4: Downloading the test material using git, and submitting your solution with git through the GitHub repository created for this purpose are part of the competencies evaluated during the exam, so submissions that do not use this medium will not be accepted, and teachers will not be able to request help to perform these tasks.

Important Note 5: Projects that do not compile correctly or cause application startup failures in the initialization of the Spring context will not be accepted as valid solutions. Solutions whose source code does not compile or is unable to boot the Spring context will be evaluated with a score of 0.

Test 1 – Creation of the Product Entity and its associated repository

It is proposed to modify the "Product" class to be an entity. This entity will be hosted in the "org.springframework.samples.petclinic.product" package, and must have the following attributes and constraints:

- An attribute of type Integer called "id" that acts as the primary key in the relational database table associated with the entity.
- A "name" attribute of the obligatory string type, and whose minimum length is 3 characters and the maximum is 50.
- A mandatory "price" attribute of the numeric type of integer (Integer), which can only take positive values (including zero).
- A mandatory “type” attribute of type ProductType. Do not remove the @Transient annotation by now.

It is proposed to modify the "ProductRepository" interface hosted in the same package, so that it extends to CrudRepository.

Test 2 – Creation of the Product Type Entity

Modify the class named "ProductType" to be hosted in the package "org.springframework.samples.petclinic.product" to be an Entity. This entity must have the following attributes and constraints:

- An attribute of type Integer called "Id" that acts as the primary key in the relational database table associated with the entity.
- A "name" attribute of the obligatory string type, and whose minimum length is 3 characters and the maximum is 50. In addition, **the name of the product type must be unique**.

In addition, a “findAllProductTypes” in the product repository must be annotated to run a query that retrieves all existing product types as a list.

Test 3 – Modifying the Database Initialization Script to Include Two Products (and Associated Product Types)

Modify the database initialization script so that the following products and product types are created:

Product 1:

- Id: 1
- Name: Wonderful dog collar
- Price: 17

Product 2:

- Id: 2
- Name: Super Kitty Cookies
- Price: 50

Product Type 1:

- Id: 1
- Name: Accessories

Product Type 2:

- Id: 2
- Name: Food

In addition, you must create a one-way N-to-1 relationship from Product to ProductType, modify the database initialization script so that each product is associated with the corresponding product type.

Test 4 – Creating a Product Management Service

Modify the "ProductService" class, to be a business logic Spring service, which allows you to get all the products (as a list) and save the products using the repository. Do not change the implementation of the other methods of the service for now. Remember that the methods of the service should be annotated with @Transactional (and the appropriate annotation parameters in each case).

Test 5 – Annotate the repository to get product types by name, and implement associated method in the product management service

Create a custom query that can be invoked through the product repository that gets a product type by name. Expose it through the product management service using the "`getProductType(String name)`" method. Remember that the methods of the service should be annotated with @Transactional (and the appropriate annotation parameters in each case).

Test 6 – Creation of a custom query for products cheaper than a certain quantity

Create a custom query by annotating a method called "`findByPriceLessThan`" in the repository, so that it takes a cost parameter (a parameter of type Integer) and returns all products cheaper than the specified quantity. Extend the product management service by implementing the method called "`getProductsCheaperThan`" to invoke the repository. Remember that service methods should be annotated with @Transactional (and the appropriate annotation parameters in each case).

Test 7 – Creation of the Controller and the method for the creation of new products.

It is proposed to create a handler method in the "ProductController" class that responds to POST requests in the url "/api/v1/products" and is responsible for validating the data of the new product, showing the errors found if they exist as part of the response (with status code 400 in that case), and if there are no errors, store the product through the product management service. Therefore, you will need to implement the "`save`" method of the product management service. Remember that such a method must be transactional.

Test 8 – Obtaining product by ID and modifying the representation of products through the API

It is proposed to modify the product controller to include an operation that allows you to obtain the data of a product based on its ID. The method should respond to GET requests in the url: "/api/v1/products/X", where X is the product id.

If a product is requested whose ID does not exist in the database, a response with a 404 code must be returned.

It is also proposed to modify the representation of the data provided by the API so that the product type is displayed as a simple text string with its name. Example of JSON rendering a product:

```
{
        "id": 1,
        "name”: “Wonderful dog collar”,
        “price”: 17,
        “type”: “Food”
}
```

You can use whatever mechanism you see fit for encoding the data (create a JSON encoder/serializer and annotate the attribute in the "Product" class, or use a DTO in this specific controller operation). Remember that to get the corresponding product, you must use the product management service.

Test 9 – Modification of existing products and security settings

It is proposed to modify the product controller to include an operation that allows you to modify a product based on its ID. The method should respond to PUT requests in the url: "/api/v1/products/X", where X is the product id. To do this, you will also need to create a deserializer or DTO that allows you to build objects with the product type expressed directly as a string. It is important that if the type of product associated with the product is not hosted in the database (e.g. you specify “unicorn” as product type), the value that is set for it in the product generated by the deserializer (or the transformation of the DTO to the corresponding product to be saved in the DB) is null and therefore a 400 error is generated.

If you are asked to modify a product whose ID does not exist in the database, a response with a 404 code must be returned.

The controller should be responsible for validating the modified product data, displaying the errors found if they exist as part of the response (with status code 400 in that case), and if there are no errors, storing the product through the product management service.

Remember to make this operation accessible only to system administrators (authority="admin").

Test 10 - Implementation of a business rule that makes it impossible to make price increases on existing products greater than 100% of their previous price

It is requested to implement a business rule in the system that prevents modifications to a product that involves a price increase of more than 100% of its original price. To do this, you must make use of the "`UnfeaseibleProductModificationException`" exception, which must be thrown by the product management service in case this situation is detected. The handler will need to catch that exception and return a 400 response code in that case. The transaction associated with the product modification must be reversed in case the exception is thrown.