Laboratory control check DP1 English (C2)

In this exercise, we will add the watch management functionality for the veterinarians of the 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 them (either through your favourite development environment or through the command "*mvnw test*" in the root folder of the project). Each successfully passed test will be worth one point.

To start the control-check you must accept the task of this lab control-check via the following link:

https://classroom.github.com/a/Jyvol_Vo

By accepting this task, an individual unique repository will be created for you; you must use this repository to perform the practical check. You must deliver the EV activity associated to the control check by providing as text the URL address of your personal repository. Remember that you must also deliver your solution of the check.

The delivery of your solution to the control will be done by 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 evaluated as not submitted.

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, there is a configuration help document in the control repository itself). You will then need to import the project into your favourite development environment and start the exercises listed below. When importing the project, you may encounter compilation errors. Don't worry, if there are any, these errors will disappear as you implement the various exercises in the control.

Important Note 1: Do not modify the class names or the signature (name, response type and parameters) of the methods provided as source material for the control. The tests used for evaluation depend on the classes and methods having the given structure and names. If you modify them you will probably not be able to make them pass the tests, and you will get a bad mark.

Important 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 by a git command prior to the execution of the tests for the issuance of the final grade, so your modifications to the tests will not be taken into account at any time.

Important note 3: As long as there are unsolved exercises there will be tests that do not work and therefore the command "*mvnw install*" will end with an error. This is normal due to the way the control is set up and there is no need to worry about it. If you want to test the application, you can run it as usual even if "mvn install" ends with an error.

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

Modify the class "Watch" to be an entity. This class is hosted in the package "org.springframework.samples.petclinic. watch", and must have the following attributes and constraints:

- The integer attribute called "id" shall act as a primary key in the relational database table associated with the entity.
- The date type attribute (LocalDate) called "date", which represents the day of the watch, shall follow the format "dd/MM/yyy" (you can use the Pet class and its date of birth as an example to see how to specify this format, but note that the format pattern is different). This attribute shall be mandatory.
- The time type attribute (LocalTime) called "beginTime", which represents the time at which the watch starts, shall follow the format "HH:mm:ss" (you can use the Pet class and its date of birth as an example to see how to specify this format, but note that the format pattern is different). This attribute must be mandatory. In the database it shall be stored with the column name "start".
- The time type attribute (LocalTime) called "finishTime", which represents the time at which the watch ends, shall follow the format "HH:mm:ss". This attribute must be mandatory. In the database it shall be stored with the column name "end".
- Do not remove @Transient annotations from the attributes from Vet representing the veterinarian that do the watch, and from WatchType representing the type of watch (localised, telephonic, on-call, etc.) for the time being. Do not remove @Transient annotations from methods either.

Modify the "WatchRepository" interface hosted in the same package to extend CrudRepository.

Test 2 – Creating the WatchType Entity

Modify the class "WatchType" hosted in the package "org.springframework.samples.petclinic. watch" to be an entity. This entity shall have the following attributes and constraints:

- An integer attribute called "id" acting as a primary key in the relational database table associated to the entity.
- A String attribute called "name" which cannot be empty and whose minimum length is 5 characters and maximum length is 30 characters. There cannot be two types of watch with the same name (the value must be unique).
- A mandatory boolean (Boolean) attribute called "overtime" representing if the watch occurs outside regular working hours.

Create a N to 1 unidirectional relationship from “Watch” to “WatchType”, using “type” as the attribute name. This attribute cannot be null.

Create a N to 1 unidirectional relationship from “Watch” to “Vet”, using “vet” as the attribute name. This relationship will represent the veterinarian doing the watch. Every watch has to be associated with a veterinary (it cannot be null).

Uncomment the method "List<WatchType> findAllWatchTypes()" in the WatchRepository and annotate it so that it executes a query to get all existing watch types.

Test 3 – Modification of the database initialisation script to include two watches and two types of watch.

Modify the database initialisation script, so that the following watches (Watch) and watch types (WatchType) are created:

Watch 1:

- id: 1
- date: 05-01-2022
- start: 07:00:00[1]
- end: 15:00:00

Watch 2:

- id: 2
- date: 04-01-2022
- start: 16:30:00
- end: 23:30:00

WatchType 1:

- id: 1
- name: Localised watch
- overtime: true

WatchType 2:

- id: 2
- name: Telephonic watch
- overtime: false

Modify the database initialisation script so that:

- The *Watch* whose id is 1 is associated with the *Vet* with id=1 and *WatchType* with id=2.
- The *Watch* whose id is *2* is associated with the *Vet* with id=2 and *WatchType* with id=*1*

Note that the order in which INSERTs appear in the database initialisation script is relevant when defining associations.

[1] `Note that the following SQL syntax can be used to insert time values into H2:` PARSEDATETIME('16:22', 'HH:mm')

Test 4 – Creation of a watch management service

Modify the "WatchService" class to be a Spring business logic service and provide an implementation to the methods that allow:

1. Get all watches (*Watch*) stored in the database as a list using the repository (*getAll* method).
2. Get the entire set of registered watch types (*WatchType*) (method *getAllWatchTypes*).
3. Save a *Watch* in the database (*save* method).

All these methods **must be transactional**.

Do not change the implementation of the other methods of the service for the time being.

Test 5 – Annotate the watch repository to obtain watches from a veterinarian that overlap with a concrete day and time

Create a custom query that can be invoked via the watches repository "WatchRepository" and receives as parameters a veterinary and a concrete date and time. The method shall return all watches from that veterinarian occurring on the provided date whose start and end times contain the time passed as a parameter. You shall uncomment the method called "findOverlappedWatches" and annotate it. You shall also uncomment the "getOverlappedWatches" method at the watch management "WatchService".

Test 6 – Checking the coherence of the watches.

Modify the "*save*" method of the watch management service so that, if the watch to be performed is concurrent in time (in date and time) with another watch of the same veterinarian, an exception of type "*UnfeasibleWatchException*" is thrown (this class is already created in the watch package). Furthermore, if the exception is thrown, the transaction associated with the watch save operation must be rolled back. **Do not use the repository method created in the previous test**, but rather use the method "`isConcurrentInTime`" of the Watch class to check if two watches overlap each other considering their times.

Test 7 – Creating a Formatter for Watch Types

Create a method with a custom query to retrieve a watch type by name in the watch repository.

Use the above method to implement the "getWatchType(String name)" method of the watchmanagement service.

Implement the *formatter* methods for the WatchType class using the class called "WatchTypeFormatter" which is already hosted in the same "watch" package we are working with. The *formatter* should show the watch types using the string of their name and should get a watch type given its name by searching for it in the DB. For this, it should use the watch management service method and not the repository. Remember that, if the formatter cannot get an appropriate value from the provided text, it must throw an exception of type "ParseException".

Test 8 – Creating the controller to display a form for creating watches

Create a method in the "*WatchController*" controller (the class must be previously annotated for it to be a controller) that allows the form implemented in the JSP file called "*createOrUpdateWatchForm.jsp*" in the "*webapp/WEB-INF/jsp/watch*" folder to be displayed. This form allows to create and edit watches (*Watch*) by specifying the date and the start and end times. The form must be displayed in the URL:

http://localhost:8080/watch/create

The controller must pass to the form an object of type *Watch* with the name *watch* via the view model. By default, the form should appear empty as we are creating a new watch. The form data must be sent to the same address using the post method.

Test 9 – Creation of the controller for the recording of new watches

Create a controller method in the previous class that responds to post type requests in the url http://localhost:8080/watch/create and is in charge of validating the data of the watch, and show the errors found if they exist through the form, and if there are no errors, store the watch using the guard management service.

After saving the watch, in case of success, the application home page (welcome) will be displayed **using redirection**. Remember that the input format of the watch date must be "dd/MM/yyyy", and the input format of the times must be "hh:mm:ss" otherwise the tests will not pass (you can use as an example the Pet class and its date of birth to specify the formats).

In case that the watch managemente service throws any *"UnfeasibleWatchException"* exception, the custom error page "InvalidWatchS.jsp" must be shown. To do so you must use the ExceptionHandlingControllerAdvice class, and annotate the `handleInvalidWatchException` method.

Test 10 – Creation of a method to obtain al watches with pagination

Implement a method that returns the list of registered watches using pagination. The method shall be implemented within the watch management service (there is an empty method called "getPaginatedWatches" for this in the class) and you must modify the repository creating a new method that returns the data paginated, and then invoke that method within the service.