UDACITY

< Return to Classroom

Critter Chronologer

REVIEW

CODE REVIEW 4

HISTORY

Meets Specifications

Congratulations. You have successfully implemented all the requirements for this project 🎉 🎉 🎉 🎉. I was able to successfully execute all the requests in your postman collection.

However, there are a couple of areas I think you can improve on.

1. Mapping enums to the DB: I can see that you created separate tables for `Skill` and `Day`. While this is good, JPA has an elaborate strategy for mapping enums to the DB. Instead of having separate tables for skills and day, you can reuse the enums `EmployeeSkill` and `DayOfWeek` and map to the DB with the `@Enumerated` annotation.

Here is a tutorial on how to map enums in Spring boot https://www.baeldung.com/jpa-persisting-enums-in-jpa

In addition, you can use the following resources for further learning.

Returning nulls or passing nulls: I can see lots of null checks in your application. This is awesome. However, it will be more awesome to not have to worry about nulls. To achieve this, never return null in methods or set null to variables. You can use default values, like an empty collection instead of a null.

See this article on why this is bad and how you can avoid it https://www.codebyamir.com/blog/stop-returning-null-in-java

See how to properly handle exceptions in spring here https://www.youtube.com/watch?v=PzK4ZXa2Tbc

Generally, I think you have done an awesome job. Keep up the good work.

Section 1: Connect Application and Unit Tests to Datasources

src/main/resources/application.properties file contains entries specifying the datasource url and user credentials.

Good job setting your datasource url and user credentials in the**src/main/resources/application.properties** file.

src/test/resources/application.properties file contains entries specifying the internal h2 url and credentials.

Great job configuring a different data source for testing. It is always good to isolate your testing environment from the main application. In addition, H2 is very fast and lightweight and thus good for testing.

Bravo 👏👏

Tables are created in a schema.sql or the application.properties file specifies an initialization mode and ddl-auto.

Great job initializing your database using JPA. You can either choose to define your schemas in **schema.sql** under the resources folder or let JPA do the heavy lifting for you. You can read more about database initialisation options here

Section 2: Design Data Layer

Each Entity should represent a single, coherent data type. This project will require Entities that represent multiple types of pets, both customer and employee users, and schedules that associate pets and employees. This will require at least three Entity classes and perhaps more, depending on which strategies for managing complex or polymorphic types are chosen.

Relationships between Entities should be clear from the Entity design. Entities that contain references to multiple objects of the same type should represent that relationship with collection classes, not by packing multiple values into a single field, such as a delimited String or bit-packed integer.

If polymorphic types are used, be sure to consider which table mapping strategy you wish to use and comment in the respective Entity classes why you've chosen a particular strategy.

Great job. You have included all the entities required for this application: `Customer`, `Pet`, `Employee`, and `Schedule`.

Again, your entities are clear from design and their relationships are correctly represented: instead of representing the Schedule-Pet relation by packing a collection of primitives representing pet ids, you chose to use a collection of pet types. This is very good.

Your entities and their relationships are correctly represented; Employees and Pets are correctly represented in the Schedule entity as a Collection of their types.

Although this is not a rubric requirement, it is a good practice to always program against abstractions rather than concrete types. You can create a superclass (call it Person or User or another descriptive name) and allow `Employee` and `Customer` subclass it. This way, as your application grows, you can abstract common attributes and behaviors in the superclass and let subclasses inherit.

You can then use an appropriate mapping strategy to represent these entities as relations in your table. Please check this article on Hibernate Mapping Strategies

The application should either use the DAO pattern or Repository pattern to isolate access to your data source. You should have one DAO or Repository for each Entity you define and that component will expose CRUD operations to the rest of the program.

A DAO or Repository can handle complex query operation, but should not combine multiple separate actions into a single call.

When considering data that can be represented by a variety of types, prefer choices that maximize clarity and limit the potential for invalid data.

- **Date vs. DateTime vs. Time - Do not store date or time information unrelated to the requirements of that field.**
- **Set vs. List - Select collections that match your constraints, such as uniqueness.**
- **Enum vs. String constant vs. int constant - Where possible, prefer Enum representation in Java to maximize compiler assistance. SQL representation is up to you.**

Additionally, consider both the current and long term needs of the data and choose data types that will be resilient to change without being inefficient.

- **Long vs Int - Generally longs for ids and ints for everything else is fine, but always consider the ranges for your values**
- **SQL Column width - Large widths don't cost anything if unused, so avoid imposing narrow width restrictions on fields that could occasionally be long, like name. Restrict width on fields you know will always have a finite length and increase width on fields that you want to allow more than the default (255 for hibernate).**

Your choice of types to represent the data in your entities is awesome. Imagine using integers to store ids for an online store that is meant to serve billions of users? In your application, you correctly anticipate such circumstances and use correct types in different scenarios. Great job 👏👏

Individual SQL request are already transactional, but any part of your program that initiates a database request should start a transaction before taking steps that can cause failure. For most projects, a sensible transaction boundary will likely occur at the Service layer. You can begin transactions by annotating the methods @Transactional, or you can specify an entire class as @Transactional to mark all methods transactional.

If you wish to use the EntityManager to manually start and end transactions you may, but be sure to minimize the scope of each method to limit the necessity.

Some developers overlook adding appropriate transaction boundaries in their application but you didn't. This is simply awesome!

With SQL requests wrapped around transactions, you ensure that your application's state is correct at any given time as requests will be rolled back when necessary to prevent issues.

Although you chose to add the @Transactional annotation, there are different ways of configuring transactions in Spring.

Section 3: Multilayer Architecture

The output from your Data layer is in a format suitable for use by the rest of the application, but not necessarily the format you want to provide to consumers of your REST endpoints.

For the purpose of this program, a number of pre-made DTOs have been provided to demonstrate the requirements of the front end without implying the structure of data on the back end. DTO needs will vary by consumer, but for the purpose of this program you will need to convert your Entity data back into the provided DTO format.

This mapping should happen either in the DTO itself or in the Controller layer, so that the Service layer does not need to know anything about DTO structure. It should not happen in the Service layer. Feel free to use Spring utilities such as BeanUtils.copyProperties to facilitate copying properties of the same name between objects.

If you're feeling adventurous, check out the Stand Out Suggestions #2 and replace some of the DTOs with the original Entities annotated using JSONView and JSONIgnore. Many applications will create DTOs for large transformations but expose annotated Entities for scenarios where little or no transformation is necessary. Note that this approach may require updates to the unit tests.

Sometimes, the structure of data we use within our application is different from what consumers of our endpoints need. Therefore, using **D**ata **T**ransfer **O**bjects to format the data that is exposed via these endpoints is

necessary.

In addition, your mapping only occurs in the controller layer. This makes sense because the controller layer is responsible for exposing data to our clients. Therefore, tying this mapping in, for instance, the service or data layer, will be a very bad decision and can lead to tightly coupled code.

Great job meeting this requirement.

Partition your logic into these layers:

- Service Layer retrieves information from one or more data sources. It can handle coordination between multiple data sources to solve multi-step problems
- Data Layer modifies or returns Entities. It can join tables to aggregate data across multiple Entity types but should avoid performing multiple operations in a single request.

For example, if you wanted to modify an incoming Schedule request if it occurs on the same date as a Pet's birthday, you would do this in the Service layer by first looking up the Pet and then modifying and saving the Schedule.

However, if you wanted to find all pets who had an event scheduled on their birthday, this would make more sense as a single query, rather than having the Service layer request all Schedules and compare dates itself.

Validation should occur in the Service layer rather than the Data layer, so rather than writing a query that will fail for invalid data, it is better to handle or throw an exception from the Service layer.

Do not put any domain logic into the Controller layer.

```
    public List<Employee> findEmployeesForServiceAndDate(Set<EmployeeSkill> s
killsFromRequest,
                                                         LocalDate date) {
        if (skillsFromRequest == null || skillsFromRequest.isEmpty()) {
            throw new RuntimeException("Request service can not be empty!");
        }
        if (date == null) {
            throw new RuntimeException("Service date can not be empty!");
        }

        // can cause issues when data becomes very large
        List<Employee> employeesFromDatabase = getAllEmployees();

        // Filter employees based on skills and date
        Set<Employee> foundEmployeesByServiceAndDate = new HashSet<>();
        for (Employee employee : employeesFromDatabase) {
            Set<EmployeeSkill> employeeSkillsFromDatabase = employee.getSkill
s();
            Set<DayOfWeek> daysAvailableFromDatabase = employee.getDaysAvaila
```

```
ble();

            if (employeeSkillsFromDatabase == null || daysAvailableFromDataba
se == null) {
                continue;
            }

            if (skillsExists(skillsFromRequest, employeeSkillsFromDatabase)
                    && checkDaysAvailable(date.getDayOfWeek(), daysAvailableF
romDatabase)) {
                foundEmployeesByServiceAndDate.add(employee);
            }

        }
        return foundEmployeesByServiceAndDate.stream().sorted(
                Comparator.comparingLong(Employee::getId)
        ).collect(Collectors.toList());
    }
```

Please, consider writing a single query to get employees available for date and activity. Fetching all employees from the database will not scale when the application grows.

Repositories should return objects of their own Entity type. For example, you would expect a “findPetsByOwner” query in the Pet Repository, rather than creating a method in a User or Customer Repository that looks up a customer then returns the Pets associated with it.

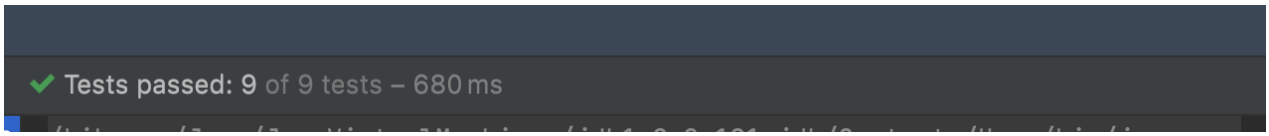
While the Data layer can join and sometimes even modify other tables, the primary focus should be on the Entity it manages. In this project, every method that returns an Entity or List of Entities from the data layer should exist in the DAO or Repository of the same name as that Entity.

Section 4: Workflow Functionality

All original unit tests must all pass. You can write your own, but you will only be evaluated on the passing of:

- `testCreateCustomer`
- `testCreateEmployee`
- `testAddPetsToCustomer`
- `testFindPetsByOwner`
- `testFindOwnerByPet`
- `testChangeEmployeeAvailability`
- `testFindEmployeesByServiceAndTime`

- `testSchedulePetsForServiceWithEmployee`
- `testFindScheduleByEntities`



✅✅✅✅

All the requests in the Postman collection can be run and return correct information based on the data requested.

Awesome 👏🏽👏🏽 I was able to successfully execute all the requests in your postman collection.

DOWNLOAD PROJECT

4 CODE REVIEW COMMENTS ›

RETURN TO PATH

Rate this review

START