UDACITY

PROJECT SPECIFICATION

Critter Chronologer

Section 1: Connect Application and Unit Tests to Datasources

CRITERIA	MEETS SPECIFICATIONS
Connect applications to external MySQL database	**src/main/resources/application.properties** file contains entries specifying the datasource url and user credentials.
Connect unit tests to internal H2 database	**src/test/resources/application.properties** file contains entries specifying the internal h2 url and credentials.
Initialize DataSources from within Spring	Tables are created in a **schema.sql** or the **application.properties** file specifies an initialization mode and ddl-auto.

Section 2: Design Data Layer

CRITERIA	MEETS SPECIFICATIONS
Create Entities that represent the storage needs of the application	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.
Create components to access the data source	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.

CRITERIA	MEETS SPECIFICATIONS
Choose appropriate data types	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).
Set Transaction Boundaries	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

CRITERIA	MEETS SPECIFICATIONS
	@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.

Section 3: Multilayer Architecture

CRITERIA	MEETS SPECIFICATIONS
Transform Entity Data into DTOs	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.

CRITERIA	MEETS SPECIFICATIONS
	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.
Separate Domain Logic from Persistence Layer	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

CRITERIA	MEETS SPECIFICATIONS
	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.
DAO or Repository objects focus on their own Entities	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

CRITERIA	MEETS SPECIFICATIONS
Write code that passes all unit tests	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`

CRITERIA	MEETS SPECIFICATIONS
	• `testSchedulePetsForServiceWithEmployee` • `testFindScheduleByEntities`
Write code that allows all requests in the included Postman collection to execute	All the requests in the Postman collection can be run and return correct information based on the data requested.

Suggestions to Make Your Project Stand Out!

1. Add additional validations. Can you prevent requests from creating invalid schedules? Avoid detached Pets with no owners?
2. It may be possible to replace some or all DTOs by using JSONView and JSONIgnore annotations. Can you replace the DTO objects without compromising encapsulation between the data layer and the controller layer? Watch out for lazy loading of collections!
3. Add support for custom behaviors for Pets, or maybe a way to specify which activities are relevant to which type of pet. Can you implement this solution in a way that allows you to add new pets or change pet behavior without modifying code each time?
4. Create new endpoints that complete the rest of the missing CRUD operations and support them on the back end. Or come up with additional data about your Entities that may be relevant and write queries to reference it.
5. Add support for Schedules that include a startTime and endTime. Add a query that can find an open employee for a specific timeslot during the day.