CECS 323 Project 3:

Feature Complete

Due date:

- Section 2:
- Section 4:
- Section 6:

You may work on and submit this project with one other person.

Overview

In this project, you will implement a JPA code-first application for the "canonical solution" to Project 2. You will not use DataGrip to design tables; instead, you will write Java classes and use JPA annotations to set columns, keys, and associations, then let JPA generate the database schema automatically. After implementing the classes, you will write a short application to fill in the database with specific object instances, and then write a demo application where a user gets to see details on automobiles in the database.

The Enterprise

The enterprise for this project is an excerpt from the **UML** design of the Project 2 Canonical Solution. Specifically, we will ignore/delete the Color, BodyStyle, and Manufacturer classes; you do not need to incorporate these fields in other places, e.g., you do **not** have to add a "color" field to the Automobile class -- simply assume that Color is no longer part of the enterprise. The remaining **six classes** -- Model, Trim, Package, Feature, AvailablePackage, and Automobile -- form the requirements for this project.

Modeling the Enterprise in JPA

You are tasked with modeling the above enterprise as JPA entity classes, *without* creating those entities as tables by yourself. (I repeat: **do not create tables in DataGrip**.) Each class in the diagram must be implemented as a Java class, using JPA annotations to set columns, keys, and associations. Specifically, your classes must:

- Include all fields from the UML design, using appropriate Java data types.
    - For string types, choose a reasonable maximum length for the field, and use @Length as an annotation.
    - You do not need to use @Column(name = ...) to rename fields from Java's naming rules to the rules we preferred earlier in the course. In general, you do not need to match the column and table names I used in the relation scheme for Project 2 Canonical Solution. You can reference them, but you do not need to match exactly.
    - You **do** need to use @Column(nullable = false) to indicate columns that cannot be null. (This is not necessary for primary keys, which can never be null.)
- Introduce surrogate keys for each class that is the parent of a one-to-many or involved in a many-to-many. In this design, that's *every single class.*
    - Your surrogate keys should be 4-byte integer values.
    - You must use @GeneratedValue(strategy = GenerationType.IDENTITY) annotations for your surrogate key fields, which will create Postgres tables with "serial" columns.
- Set the class primary key(s) using @Id.
- **Not include foreign key columns as fields**, e.g., the Automobile class should not have a "trimId" integer field. Instead, all associations **must be modeled as JPA associations** as shown in lecture.

    - This means using @OneToOne, @OneToMany, @ManyToOne, @ManyToMany, @JoinColumn, and @JoinTable correctly.
    - These associations should be **unidirectional** in the order specified:
        - Package -> Feature (example: a Package can tell you its set of Features, but a single Feature cannot tell you which Packages it comes from)
        - Trim -> Feature

        - Model -> Feature
        - Automobile -> Trim
    - These associations should be **bidirectional**:
        - Trim - Model
        - Trim - Package
        - Automobile - AvailablePackage
- Set up any **unique keys** in the class:
    - Automobile: unique vin.
    - Trim: unique (modelId, name).
    - Feature: unique (name)
    - Model: unique (name, year)
    - You will need to Google how to do a multi-column unique constraint in JPA. It uses an annotation on the class itself.
- **Using your IDE** (please god don't do this by hand), generate code for:
    - a parameter-less constructor
    - a constructor taking all **mandatory** fields
        - A mandatory field is one that the programmer **must** provide for the object to have a meaningful state.
        - Generated fields are not mandatory; they will be generated by JPA. Even if you attempt to set your own "trimId", it will be overridden by JPA if it has the @GeneratedValue annotation.
        - Associations (references to other objects) are only mandatory if they can & must be known at object construction, and not knowing them would leave the object in an invalid state. A Trim must know its Model at construction; but it might not know its entire set of AvailablePackages (especially because that might change over time), so the Trim constructor should not mandate a collection of AvailablePackages.
        - Any collection that is not mandatory should be initialized as empty in the constructor; **not** left to be null. (Another programmer should **not** have to check your "getX()" method to see if that's null prior to trying to add something to it.)
    - getters for all fields

    - setters for all non-primary key fields.
        - Once an object is persisted to the database, its primary key cannot change (you cannot UPDATE a primary key column). So there should not be public setters for PK fields.

Dealing with mistakes

Since we are using JPA to generate our tables, running our program before our work is complete will result in an incomplete database being created. This is a good and bad thing. It lets us inspect the tables created by JPA using DataGrip to make sure they look how we intend so far, but the JPA demo I gave you **won't create tables if the tables already exist**. This is normally what you want; you don't want all your data to be erased each time you run an application. But while you're doing development work, this can get annoying; if you make a mistake in a table, run the app, fix the mistake, and run the app again, the app **won't** update your table because the table already exists.

I recommend editing the **src/META-INF/persistence.xml** file in your application to change this. You want the entry for "jakarta.persistence.schema-generation.database.action"; uncomment the line to go to "drop-and-create" mode.

Once you're satisfied that your classes are correct, you can change this setting back to **create** so that you won't lose any objects that you add to the database when you are testing the application code. (More on this below.)

Adding application logic

Until this point, your classes are only for modeling the data; they have no real *behaviors* / methods other than boilerplate getters and setters, and the bidirectional helper methods. You will add **two** "application" logic methods to your class Automobile:

1. public Set<Feature> getFeatures()
   Returns a Set of all Feature objects that this Automobile has because of its Model, Trim, or chosen Packages. The HashSet class and its addAll method will be helpful here.

2. public double stickerPrice()
   Returns the "sticker price" of the automobile: the sum of the Trim cost, plus the costs of all packages added to the automobile. Do not try to write a JPQL query to do this; you have all the data you need among the fields of the Automobile object and the methods of its related classes.

Instantiating the Model

Once your model classes are correct and JPA seems to be generating the correct database schema, you can move on to writing a Java method to instantiate the model by creating Java objects and persisting them to an EntityManager, in order to give the database some "default" values to work with.

Your method should create the following objects, using public methods of your entity classes to add associations between related objects:

- Features:
  - leather seats
  - plug-in hybrid engine
  - power sliding doors
  - hands-free sliding doors
  - Amazon FireTV
  - rear-seat entertainment screens
  - all-wheel drive
  - adaptive cruise control
- Packages:
  - Theater Package
    - Features: rear-seat entertainment screens
  - Amazon Theater Package
    - Features: rear-seat entertainment screens; Amazon FireTV
  - Safety Package
    - Features: adaptive cruise control
- Models:
  - Pacifica, 2022
    - Features: power sliding doors
    - Trims:

        - Touring ($30,000).
            - Compatible packages: Safety Package ($3000)
        - Limited ($34,000).
            - Features: leather seats, hands-free sliding doors.
            - Compatible packages: Amazon Theater Package ($2500)
        - Pinnacle($42,000).
            - Features: leather seats, hands-free sliding doors, rear-seat entertainment screens, Amazon FireTV, all-wheel drive.
    - Pacifica Hybrid, 2022
        - Features: power sliding doors, plug-in hybrid engine
        - Trims:
            - Touring ($43,000).
            - Limited ($48,000).
                - Features: leather seats, hands-free sliding doors.
                - Compatible packages: Amazon Theater Package ($2500)
            - Pinnacle($54,000).
                - Features: leather seats, hands-free sliding doors, rear-seat entertainment screens, Amazon FireTV.
    - Pacifica Hybrid, 2021
        - Features: power sliding doors, plug-in hybrid engine
        - Trims:
            - Touring ($41,000).
                - Compatible packages: Safety Package ($3000)
            - Limited ($46,000).
                - Features: leather seats, hands-free sliding doors.
                - Compatible packages: Theater Package ($2500), Safety Package ($2000)
            - Pinnacle($52,000).
                - Features: leather seats, hands-free sliding doors, rear-seat entertainment screens, adaptive cruise control.

- Automobiles:
    - VIN 12345abcde, 2022 Pacifica, Limited trim. Amazon Theater Package.
    - VIN 67890abcde, 2022 Pacifica Hybrid, Pinnacle trim. No packages chosen.
    - VIN 99999aaaaa, 2021 Pacifica Hybrid, Pinnacle trim. No packages chosen.
    - VIN aaaaa88888, 2021 Pacifica Hybrid, Touring trim. Safety Package.
    - VIN bbbbb77777, 2021 Pacifica Hybrid, Limited trim. Safety Package, Theater Package.

Programming the Registration Application

Once your model instantiation is done and tested, you can write the actual application for this project. Your main will print a menu with three options, then branch on the user's choice. Each branch should be its own method called by the main.

Options:

1. Instantiate model.
    1. Instantiates the model using the code you previously wrote.
2. Automobile lookup.
    1. User enters the VIN of an automobile.
    2. Use JPQL to find that automobile if it exists.
    3. If it does, print the automobile's information in the following format:
       [year] [model] [trim]
       [sticker price]
       Features:
       [all of the automobile's features, in alphabetical order, one per line]
    4. **WARNING**: this branch requires running a query with user input. You **must** correctly used a parameterized query to eliminate potential injection attacks. Doing this incorrectly will net a **20% deduction on your project.** (Yes, I am serious.)
3. Feature search.
    1. User enters the name of a Feature.

2. Use JPQL to find the Feature if it exists.
3. If it does, print the VIN of every Automobile with that Feature. This is tricky, because our Feature class does not have a bidirectional reference to the Trim, Model, or Package objects that include that feature. **This is deliberate, and you cannot add bidirectional references to solve this problem**. You need to find a different way.

   You must do this efficiently, which means you **cannot** select all Automobile objects and then ask each individually if it has the given feature. The correct way will be to use JPQL JOINs to navigate an automobile's relationships in JPQL. JOIN is a little different than in SQL; the JPQL query below will find any Automobile whose Trim has a feature named 'leather seats', and you should be able to extrapolate the rest from here.

   ```
   SELECT a
   FROM automobiles a
   JOIN a.trim t
   JOIN t.features tf
   WHERE tf.name = 'leather seats'
   ```

4. See the note above about injection attacks.

Deliverables

You must deliver:

- The **src/** folder and all its contents **only**. Do not turn in the lib/, out/, prod/ etc. folders.