Ticket to Ride Phase 4

Overview

In this phase you will add data persistence to your Ticket-to-Ride server. This will allow your server to be shutdown and restarted without losing its state (i.e., user accounts and games). You will design a pluggable persistence subsystem that will allow your server to run with different persistence implementations, and allow new persistence implementations to be plugged in without recompiling your server. You will gain experience with the following design principles and patterns:

1. Program to interfaces
2. Command Pattern
3. Data Access Object Pattern
4. Abstract Factory Pattern
5. Plugin Pattern
6. Database design, transactions, object serialization

New Patterns and Design Principles

1. Plugin Pattern
2. Abstract Factory Pattern
3. DAO Pattern
4. Database design, transactions, object serialization

Requirements

Modify your Ticket-to-Ride server to save its state to persistent storage. It should be possible to shutdown your server and restart it without losing user account or game information. This includes both intentional shutdowns and unintentional shutdowns caused by a crash, power failure, etc. This will require you to keep the state of your in-memory server model and persistent storage in sync at all times. To achieve this, you should introduce a persistence subsystem. After creating the persistence subsystem, you should modify your server's web API

handlers to call the persistence subsystem to keep the in-memory model in sync with persistent storage.

Define the API for your persistence subsystem using abstract interfaces that can be implemented using a variety of different underlying technologies (relational DB, document DB, cloud storage, etc.) In designing your persistence API, you should use both the Data Access Object and Abstract Factory design patterns as follows:

1. Design a "persistence provider" interface to serve as the entry point into your persistence subsystem. This interface should provide operations for starting and ending transactions and also for clearing existing data from the database. It should also serve as an Abstract Factory for the user and game DAOs (described next).
2. Design a DAO interface for managing user accounts.
3. Design a DAO interface for managing games.
4. Design any additional DAOs (if any) as you see fit.

Implement the Plugin pattern so that different persistence provider implementations can be plugged into your server without needing to recompile the server. This will require you to provide a mechanism for registering external JAR files with your server. When the server is started, the user should be able to specify the name of the persistence provider they want to use during that execution of the server. The server should load, initialize, and use the specified persistence provider until it shuts down. Once the persistence provider has been initialized, the rest of the server should be unaware of what particular persistence technology is being used (i.e., all interaction with the persistence subsystem should be through the abstract provider and DAO interfaces).

Plugins should be packaged as JAR files. Plugin implementations should be dynamically loaded from their JAR files, and not compiled into the server. Your server must provide a way for plugins to register themselves with the server. This is typically done by using a configuration file that lists the name and JAR file path for each plugin. Another approach is to create a special subdirectory where plugin JAR files are placed, and have the server scan this directory for plugins.

Implement at least two different persistence provider plugins that use different underlying technologies. One of the implementations should be a relational database. The other implementation is your choice, but must be something other than a relational database.

You should store the state of each game as a BLOB (i.e., serialize the game's state, and store the serialized data). For a relational database, this will drastically reduce the number of tables you will need -- you shouldn't need more than about three or four tables in your schema.

In your persistence providers, rather than saving the entire state of a game each time it changes, you should use a "checkpoint plus deltas" approach. Here is the basic idea:

1. When a game is created, store a BLOB containing its initial state. (This is the initial checkpoint.)
2. Store the sequence of Commands executed in the game. Each time a Command is executed, append it to the list.
3. The "current" state of a game can be computed by starting with the initial checkpoint and replaying all of the Commands executed during the game. However, in order to speed up this process, you should update the game's state BLOB after every N commands (where N is a command-line parameter to your server). This will allow the "current" game state to be computed by starting with the most recent checkpoint, and replaying only those Commands that have been executed since the checkpoint was last updated.

Your server should accept two command-line parameters:

1. The name of the persistence provider plugin to use during this execution of the server.
2. The number of Commands between game state checkpoints.

You should also provide a way to 'clear' the data in your persistence provider. You can do this by adding another command-line parameter to the server ('-wipe' for example).

Design

Create UML class diagrams for the following:

1. Persistence provider and DAO interfaces and classes that will implement them.
2. Interfaces and classes related to your Plugin implementation.
3. Any other interfaces and classes central to your design.

Provide a write-up of how your two persistence provider implementations will work. Which technologies will you be using? Explain how you will store all user account and game information. Explain in detail how you will implement the "checkpoint plus deltas" approach for storing game state. For your relational database provider, include the schema for your tables (i.e., CREATE TABLE statements).

Provide a write-up explaining how you will implement the Plugin pattern to support pluggable persistence providers. How will plugins be registered with your server? How will it load them?

How will the current persistence provider be accessed by classes that need to interact with the persistence subsystem?

Turn your design into the TAs in the same way you turned in the design for phase 3.

Implementation

Implement your design.

In addition, the command line for our server is required to have two parameters: "persistence_type" and "number_of_commands-between-checkpoints"

For example,

java ticketToRideServer sqlite 10

Deliverables

Design

1. Design documentation

Implementation and Testing

1. Fully working server that supports pluggable persistence subsystems
2. At least two working persistence providers (one relational, one something else)
3. A way to 'clear' out the data in your persistence providers from the command line.