Royalty Manager: Architecture and Technical Solution

Introduction

NOTE: This project has been implemented and documented by Jorge Ruiz Quiñones, in order provide a solution for a Code Challenge proposed by P.
The project is stored and accessible in a public github repository:
https://github.com/jorgeruizqui/pcc

This document tends to be a brief description of the architecture and technical solution implemented for the Royalty Manager System code challenge, from P. company.

A fully working and functional solution has been provided. Please, find more details in 'Running' chapter to know how to launch the application and tests provided with the solution.

Architecture

It has been required to implement a system that manage royalties for Studios, based on Episodes viewing by different customers. The main interface of the system within third party users is a REST API with different endpoints to provide the required functionality. The system will be in charge of storing and managing all data about Studios and its royalties and viewings, Episodes and customers. Find next the details about the architecture and continue reading this chapter for more details:

Royalty Manager Architecture Diagram

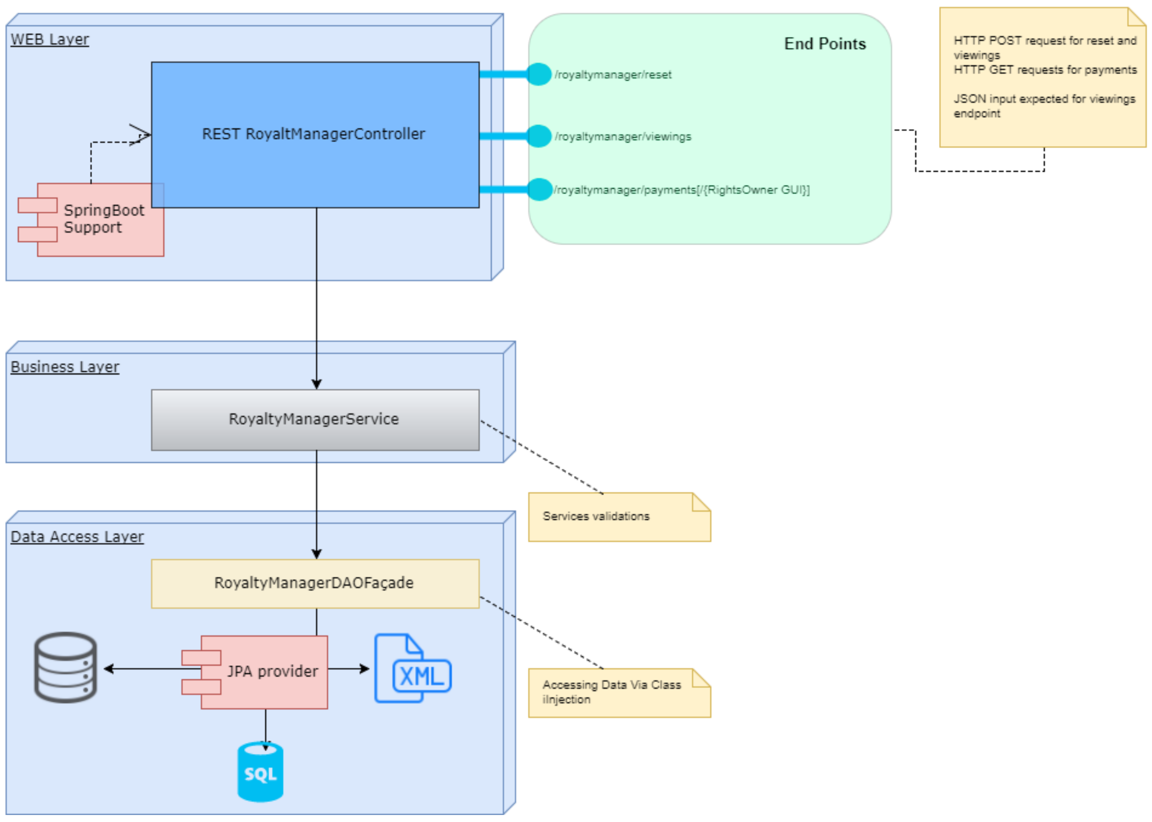
WEB Layer
REST RoyaltManagerController
SpringBoot Support
End Points
/royaltymanager/reset
/royaltymanager/viewings
/royaltymanager/payments[/{RightsOwner GUI}]
HTTP POST request for reset and viewings
HTTP GET requests for payments
JSON input expected for viewings endpoint
Business Layer
RoyaltyManagerService
Services validations
Data Access Layer
RoyaltyManagerDAOFaçade
JPA provider
XML
SQL
Accessing Data Via Class iInjection

Figure 1: Royalty Manager System Architecture

As shown in Figure 1, a three layer architecture has been implemented for the solution:

1. The main interface with users (third party components) will be the Web Layer, providing the endpoints as REST Services via HTTP requests.
2. The business layer implements the Royalty Manager Service, which is in charge of providing all needed services to upper layers or other third party components. In this case, only the web layer will use the services. Separating this layer from the web layer allows:
    a. Make the business layer independent from the web layer
    b. Provides the possibility of different upper layers implementations making the system scalable as needed.
3. The Data Access layer which is in charge of providing the interface with the final storing service. Separating this layer from the business layer allows different implementations for Data Access Layer and make the system more maintainable and scalable.

Web Layer

This layer provides users the endpoints as described in the requirements. Web layer provides the following end-points:

- Viewing: Tracks a new viewing in the system for a concrete episode and studio (rights owner).
- Payments: Returns the list of payments for all rights owners. If right owner is specified, it will return just the information for that concrete studio.
- Reset: Restores the viewings information for all studios to zero.

Business Layer

Represents the Royalty Manager engine and the services provided to upper layers of the system or even third party users directly. Interface has been implemented according to requirements with some more options, useful for executing a fully working application or testing.

The provided interface encapsulates the Data Access Layer (only visible inside the Service) in order to provider scalability and independency of the layers. A brief resume of the services is detailed next:

- Loading services: Performs an initial (or not) loading of the Data Store
- Entities management: Manage different entities of PCC model (Studios, Episodes and Customer), providing access, save and remove operations.
- Provide Functional Services: Reset and viewing operations are provided by the Royalty Manager Service.

IMPORTANT NOTE: Take into account that this module is implemented as an engine and it has no sense alone by itself. No DAOs are configured and implemented in this module. It just defines the contracts (interfaces) to be accomplished and should the final applications configurations and implementations who are in charge of configuring and implementing the concrete DAO classes to use in consonance with the business layer. As is detailed later, a complete example of a DAO, working with memory concurrent maps, has been implemented for a fully working application and tests.

Data Access Layer

Using Factory and Façade patterns, Data layer can be configured and injected in order to provide flexibility for the system.

Any Persistence engine can be used and injected to perform the operations against a concrete data store.

Implementation

The solution has been implemented under the name of PCC (stands for P. Code Challenge). Following, all details about every of the component provided with the solution.

Justification

The main goal for the challenge has been getting a fully working and testing system according to the requirements.

Eclipse (Mars 3.2) IDE has been used to develop the project, as one of the most used IDEs in development world. Take into account that PCC projects (configured to be used with Maven) are independent of the IDE and can be used in every other IDE like NetBeans, for instance.

PCC projects are likely to be configured under maven (providing a Project Object Model for each of them), as one of the project management software more used and robust nowadays. Using maven takes advantage of the dependency management and the third party dependencies management through maven central repository. In a company oriented product (in a real situation), may be a mirror can be used to avoid direct internet connections and take advantage of the use of SNAPSHOTs versions for team integration development.

Java JDK 8 has been used for taking advantages of new features like collections API improvements, multi-catch blocks, importing static methods and functional interfaces, among others.

Spring (and concretely Spring-boot) framework has been used in order to:

- Take advantage mainly of the Dependency Injection and Model View Controller provided by spring.
- Using spring-boot allows easily develop a fully working Web Services application, without having dependences with web servers or containers and takes advantages of all spring features.
- JUnit testing framework has been used for testing both, the Royalty Manager Service and the web application to make sure every component and individual method works as expected.

Quality Assurance

In order to develop a clean, maintainable, readable and clear code, some tools and standards have been used in PCC project:

- Checkstyle: Configured as an eclipse plugin, allow to make sure that all code is well formed, documented, naming according to java standards, etc… Is configured using file /config/pcc-cs-rules.xml.
- Formatter: Java files and XML formatter to maintain all files with the same format and structure and used as well to remove blank lines, spaces, braces position, etc… Is configured using the /config/pcc-formatter.xml file
- Cleanup: Rules applied to projects when cleaning code or saving files, making sure the code is well formatted, has the correct indentation, among other features. Is configured using file /config/pcc-cleanup.xml

Modules

pcc-def

This module does not contain any code. Is just the Maven parent project for the rest of the modules. It defines the project features and version and also is in charge of managing all dependences for all modules.

pcc-core

This module contains the implementation of the Business Layer, providing services to other modules and third party modules. The main class implementing the Service class is `es.jrq.pcc.core.service.RoyaltyManagerService`, which is a singleton class encapsulating all related to Data Access Layer. During the service class instantiation, the DAO Factory will be in charge of reading configuration for obtaining the Persistence classes implementation.

In package `es.jrq.pcc.core.dao` are located the Interfaces definition for each of the DAOs used in the system, and also the DAO Factory and Façade. The Façade is in charge of providing the Service class all DAOs instantiated in the system.

The model entities of the systems are implemented under the `es.jrq.pcc.core.model` package. Classes represents basically the instances of the Data Store model, with some additional points:

- As requirements say, every studio (right owner) will calculate royalties independently, so, an interface `es.jrq.pcc.core.model.royalty.IRoyaltyCalculator` has been defined and is included as a field in the `Studio` class. This class can be injected at running-time, so every Studio can have its own Royalty Calculator implementation.
- In order to make the example fully working, a basic implementation of a Royalty Calculator has been developed (`es.jrq.pcc.core.model.royalty.DefaultRoyaltyCalculator`). This basic implementation read a system property called “`defaultRoyaltyAmount`” (can be set with -D application parameters) and that value will be the amount added to the global studio payment after every registration of a viewing. If the parameter hasn’t been set in application properties, a default amount defined as a constant will be used instead.

pcc-dao-map

This module contains the implementation of a concrete Data Access Layer. In this case, a Concurrent Memory Map has been implemented as data store (see Topics Not Covered section for more information about other possibilities and justification).
Basically, the implementation consists in a implementation of the DAO interfaces defined in pcc-core module, each of them managing the access to data store (memory in this case).

A configuration is provided in order to use the system with theses DAO implementations to be used.

pcc-web

This module contains the implementation of the web layer. The solution, as detailed in the Justification section, has been developed used Spring and Springboot framework. The main aspects of this project and components according to the framework are following detailed:

`es.jrq.pcc.web.RoyaltyManagerWebLauncher`: Is a main Java class used to launch the server, configure the spring application, instantiate the PCC Service and loading initial data into the configured data store. See Running section for more details.

`es.jrq.pcc.web.RoyaltyManagerSpringBootApp`: Is annotated as an spring boot application, defining the packages to scan in order to discover spring components and beans.

`es.jrq.pcc.web.controller.json.RoyaltyManagerController`: Is the REST controller in charge of managing the `/royaltymanager/*` HTTP requests received by the application. Implements the endpoints specified in requirements with the corresponding HTTP method and is in charge of returning the corresponding HTTP code as defined in requirements.

`es.jrq.pcc.web.http.json`: This package contains the implementation of the web layer for JSON as specified in requirements. A spring Bean (`es.jrq.pcc.web.http.json.JSONHttpMessageConverter`) is configured to make sure that requests will be mapped to/from JSON format objects. By default, spring-boot is able to chose the JSON converter if it is in the classpath, but defining the bean explicitly in the project is more powerful because provides the possibility of set every configuration option according our needs.

In order to use a different implementation that JSON, the Viewing data received by the viewing request has been defined as an interface, so different implementations can be developed in the future, accomplishing this interface.

Software Design and Architecture Patterns

This "*not so simple*" code challenge allows to apply different and useful software design and architecture patterns. Some of them are present in the solution in order to take advantages provided by each of the patterns.

Note that pcc-web is based on spring framework, as explained before. Although spring implements and uses itself lots of software design patterns, for instance providing the capability of creating beans by default implementing the singleton pattern, pcc-core project is outside the scope of spring. It has been done this way consciously in order to have enough flexibility for implementing explicitly some of those patterns.

Following, a brief description of the used/implemented patterns at architecture and software design level:

- Service Oriented Architecture: As the application is focused on REST HTTP Services
- Layer Architecture: Providing different and independent layers allowing scalability and maintainability projects.
- Singleton pattern: Royalty Manager Service is implemented as singleton to make sure only one instance of the class is instantiated in the system
- Interface/Implementation: It is used for instance in the DAO interfaces, defining the contracts to be accomplished by the implementations, in this case, the implemented Map DAOs.
- Façade: Is used in the business layer to provide access to all DAOs configured in the system and used by the service.
- Factory: DAOs are constructed and instantiated using a Factory class that creates the instances based on configuration.
- Injection: DAOFactory reads configuration and injects the implementations to the interfaces fields. Royalty calculations use as well this kind of pattern in order to allow different implementations of Royalty Calculations for different studios.

Code Documentation

The javadoc of PCC projects are available in /doc/solution/pcc-javadoc folder and compressed into RAR file /doc/solution/pcc-javadoc.rar

Running

Running application under IDE

The main class for running the application to get a fully working example is launching the class `es.jrq.pcc.web.RoyaltyManagerWebLauncher` from the pcc-web project. This class is in charge of creating the Springboot context, launching the server, instantiate the REST controller and everything depending on the web layer. Also, it initializes the Royalty Manager Service to allow accessing data and make the example fully functional.

Running JUnit tests

Several JUnit tests suites have been developed to test and provide robust and correct code. Following the standards, these tests are under src/test/java folders, where tests can be found in these packages:

- `es.jrq.pcc.core.service`: Test suite for Royalty Manager Service in charge of testing all service interface
- `es.jrq.pcc.web.controller`: Test suite for Royalty REST Controller in charge of testing the full system (from web layer to data layer)

For running tests, just right click the the package and select Run as JUnit.

Running application as a fully working standalone java application

Topics not covered

As is not specified in the requirements and taking into account that the time is limited and the code challenge is focused in the implementation of the provided REST API, some important topics are not included in the given solution. To mention some of them:

- Physical Data Store (SQL Database, XMLs, NoSQL, etc…): The solution, as seen before, is provided with a Data Access Management implemented with Concurrent Maps, working and persisting data in memory. Obviously, this is just a basic implementation of Data Persistence not suitable for a real environment. Many other solutions (clearly, better solutions for persisting data) are not implemented, mainly, because of time to develop the challenge and also because the software is absolutely ready to accept any other implementation (if needed) due to the DAO implementation injection during execution time from configuration.
- Exceptions management: Exceptions are not in the scope of the solution. Data access layer and business layer should have its own exceptions definitions and they should be treated as needed.
- Logging management: Only Console Appender logger from Spring boot has been used. No Logger has been configured for the solution, understanding that log debug, warning, info and error messages should be added to complete the implementation.
- Software engineering patterns: As explained in previous chapters, some important and well known software development patterns have been applied to the given solution, but is important to notice that some others, like Event/Listener, publisher/subscriber or Builder pattern, are not in the scope of the implementation for this solution. It has been attempted to use as many patterns as possible according with the difficulty of the challenge.