Stock Price Monitoring Application: Design and Implementation Details

Database Design

Our objective is to create a database schema in which we can

- Store a list of stock symbols
- For each stock symbol, maintain a history of the trading price at various times

To monitor and collect information about last trading price of various stock symbols, the following database design is used.

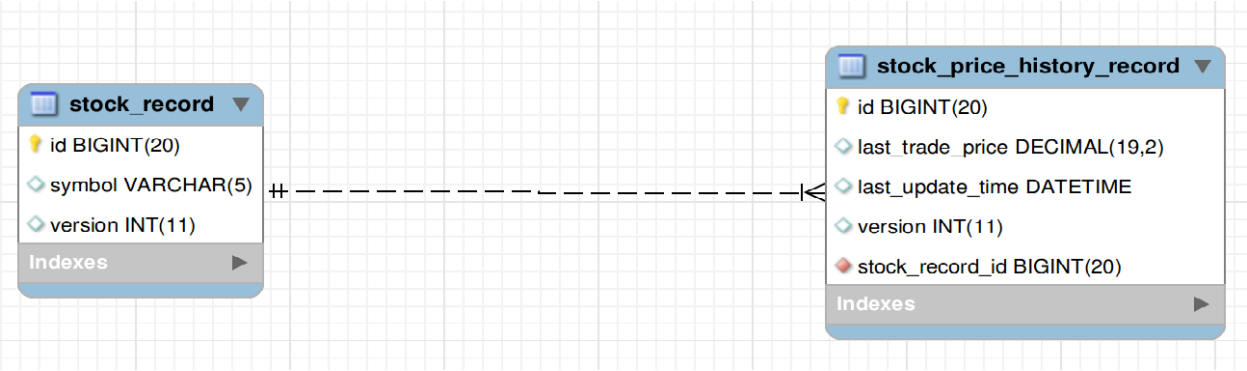
First, a **stock_record** table stores the **id** (a numeric field acting as primary key for the table) and the **symbol** (a varchar2 field of size 10, with a constraint enforcing unique values per row).

Second, a **stock_history_price_record** stores the **id** (a numeric field acting as primary key for the table), the **last_trade_price** as retrieved from a service provider (a numeric field), the time of retrieval (a timestamp field).

A stock record will have multiple price records retrieved at various time intervals. As a result, there is a **One-to-Many** relation between **stock_record** and **stock_history_price_record.** Thus, a foreign key **stock_record_id** in **stock_history_price_record** references the primary key in **stock_record**.

I will use **Java Persistence API (JPA)** in the application for Object-Relational Mapping (**ORM).** Per JPA standards, a **version** field is also present in both tables. I will also take advantage of MySQL's auto increment feature and JPA's primary key generation strategy for generating primary keys.

The final database diagram of our system is given below.



Application Design

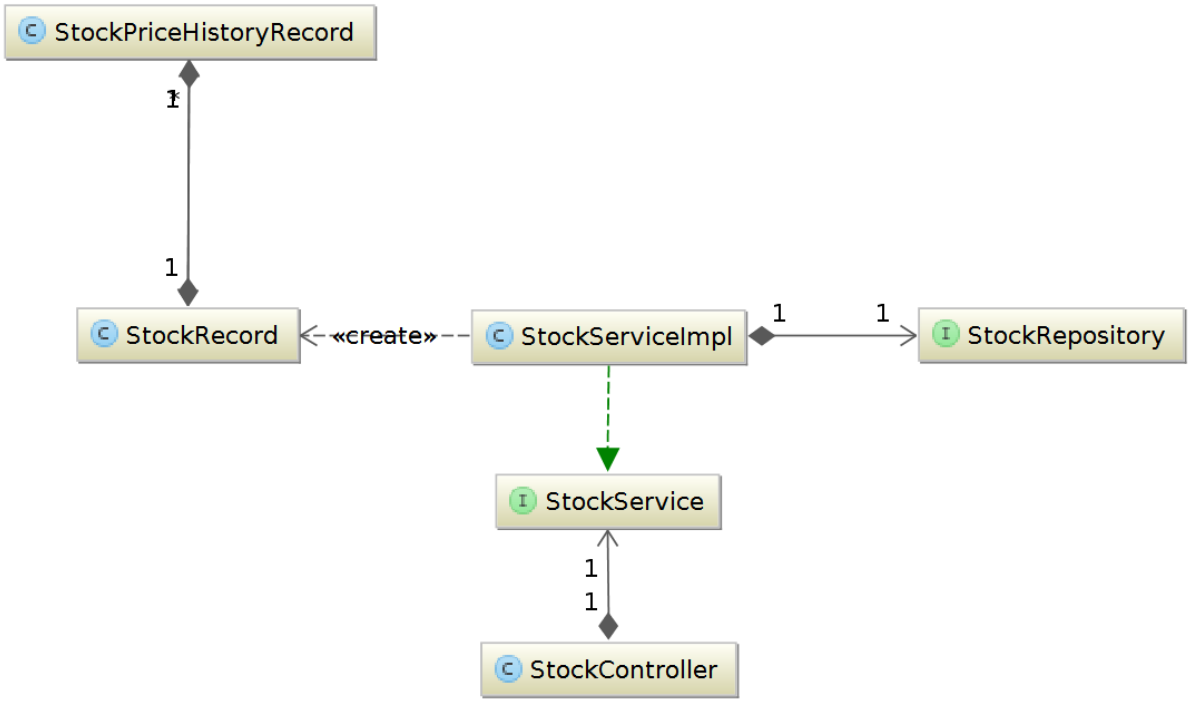
REST API

My REST API is as follows:

	GET (Read)	POST(Create)	DELETE(Delete)
/stocks	Lists all Stock symbols in the database		
/stocks/GOOGL		Create a record in the database for a stock having the symbol GOOGL.	Delete the record in the data for a stock having the symbol GOOGL. Also delete corresponding price record history for GOOGL as well.
/stockhistoryrecords/GOOGL	Return all price records associated with GOOGL in the database		

Class Design

The following class diagram represents the underlying architecture of the above REST API.



The architecture is based on the standard Model-View-Controller (MVC) pattern for a web application. I also decouple business and database logic into separate components. This also leaves the design open to future extension, as well as making each component very easy to unit test.

Firstly, **StockRecord** and **StockPriceHistoryRecord** are classes mapping to the relational tables **stock_record** and **stock_history_price_record** respectively. The **StockRecord** class has a **List** of associated s**tockPriceHistoryRecord**s.

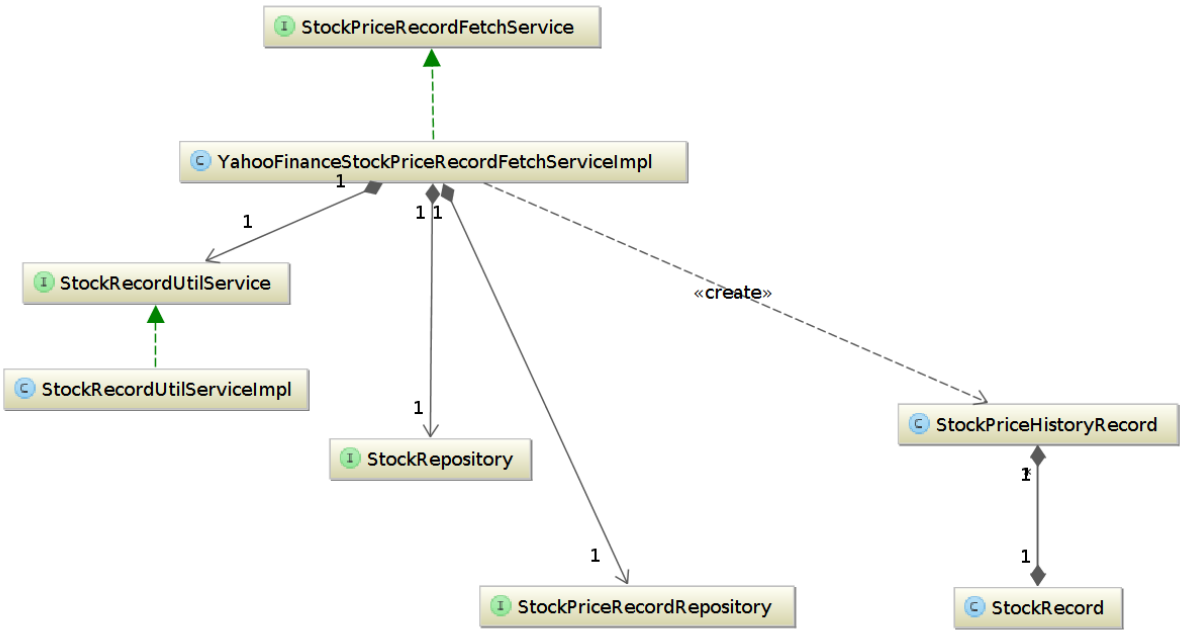
StockController services all calls to the REST API. Each possible call to the REST API i.e. each HTTP request is mapped to one method in the **StockController**.

StockController has a reference to a **StockService**. **StockService** is an interface containing methods to execute all the required business logic to service calls to the REST API.

In the current design, **StockService** is implemented **StockServiceImpl. StockServiceImpl** provides an implementation of the business logic specified in **StockService**. This particular implementation services the calls to the REST API by finding, creating and deleting instances of **StockRecord**. However, it does not interact with the database directly, but has a reference to a **StockRepository** instance, which handles the underlying calls to database.

Class Design of Stock Price Fetch Service

The design for the standalone Stock Price Fetch Service is given below.



StockPriceRecordFetchService is an interface specifying a method for fetching latest trade price of all stocks. **YahooFinanceStockPriceRecordFetchServiceImpl** is an implementation of this specification based on the database schema mentioned above and usage of Yahoo Finance's API. In this implementation, a list of all **StockRecord**s are obtained using **StockRepository**. A call is made to YahooFinance's API to fetch the latest stock price for these records. Finally, for each new price fetched, a new **StockPriceHistoryRecord** is created and saved using **StockPriceRecordRepository**. Some additional processing of the data structures is necessary during this process. For purposes of decoupling of logic/purpose and increasing testability, these were moved to a separate interface, **StockRecordUtilService** which in turn was implemented by **StockRecordUtilServiceImpl.**

To ensure that **YahooFinanceStockPriceRecordFetchServiceImpl** runs periodically at regular intervals, it is made into a **scheduled task** through the help of Spring Framework. The schedule is configurable by a cron expression, present in an application.properties file.

Design Patterns and good practices used

- MVC design pattern was used in designing the REST API to ensure less coupling between various layers and ensure separation of concerns
- I used Object-Relational Mapping through JPA to
    - Be able to model domain objects based on actual project concepts instead of database structure
    - Achieve reduction in having to write boilerplate SQL query related code
- I used Inversion of Control wherever possible. This allowed me to
    - Decouple components and layers in the system
    - Prevent having to manage a component's dependencies in the component itself
    - Perform unit testing of components through mocking of dependencies
- Inversion of Control was ensured by using Spring Framework for dependency injection
- I also used "Convention over Configuration" extensively throughout the project. I used Spring Boot and Spring Data JPA extensively throughout the project.
    - All of my database queries were standard database queries. So I used Spring Data JPA's convention over configuration to just declare my required interface, and Spring Data JPA provided the runtime implementation on the fly.
    - Most of my required Spring Framework configurations were standard configurations, so I used Spring Boot in the project to automatically perform the configuration for me.
- Used a property file to centralize application configuration in one place

Implementation

Prerequisites

The following must be pre-installed in order to build and run this project:

- JDK 1.8 (can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html)
- Maven 3.0+ (can be downloaded from https://maven.apache.org/download.cgi)
- MySQL (can be downloaded from https://www.mysql.com/downloads/)

Technologies/Libraries/Frameworks Used

The following technologies were used in the implementation of this project:

- Language used – Java 1.8
- Build Tool - Maven
- Database - MySQL
- ORM - JPA
- Unit Testing - JUnit, Mockito
- Frameworks - Spring Framework, Spring Data JPA, Spring Boot
- SCM – Git, repository hosted on Github
- Continuous Integration – TravisCI, integrated with Github repository

Pre-build setup

1. Install all the software (Java 1.8, Maven 3.0+, MySQL) mentioned in the prerequisites section
2. Create a database user in MySQL with the necessary privileges
3. Create a database in MYSQL which to store the data
4. The project is currently configured such that the JPA implementation (Hibernate) will automatically update the schema based on the JPA annotated classes, so manually creating the tables is not necessary.
5. Add the necessary database credentials in the **application.properties** file. It can be found in $PROJECT_HOME/src/main/resources directory. The necessary credentials needed to be provided are:
    - spring.datasource.url=dburl (dburl will be be of the form jdbc:mysql://{dbhostname or IP}:{port on which MYSQL is running, typically 3306}/{dbname i.e. name of the database created in step 3})
    - spring.datasource.username=dbuser (name of the user created in step 2)
    - spring.datasource.password=dbuserpassword (password of the user created in step2)
6. This concludes pre-build setup.

Building, Testing, Committing

1. Open a terminal/command-line
2. Traverse to project directory
3. The project can be built by running the command "**mvn clean package**" in the terminal/command-line from the project directory. This will also automatically run all the unit tests, and build will only complete if all unit tests are passed.
4. Finalized changesets are committed in Git and pushed to remote Github repo
5. The TravisCI tool is set up to run all unit tests and build the project when a new commit is pushed to the Github repo, and sends an email notification for both successful and unsuccessful builds.
6. There are integration tests for testing the REST API as well. The Maven POM file is configured to ignore these during normal build to make build process fast. Also, since the integration tests make direct calls to the MySQL database, they will fail when TravisCI will try to execute them. That is another reason why integration tests are disabled normally. However, integration tests can be run by executing the command "**mvn integration-test –P integration"** in the terminal/command-line.

Deployment

1. The project contains both the REST API and stock price fetching service in a single jar.
2. Running the jar will deploy the REST API, and the stock price fetching service will run independently to the REST API on an interval specified in the application.properties (currently every 30 minutes).
3. The project can be run directly from the terminal/command-line in the project directory by running the command "**mvn spring-boot:run**".
4. Also, the packaged jar, found in ${PROJECT_HOME}/target directory after building the project can also be run from the terminal/command-line using the command "**java –jar stock-price-monitor.jar**".