Class 6 – Spring Boot Microservice Enterprise Applications with Java

Contents

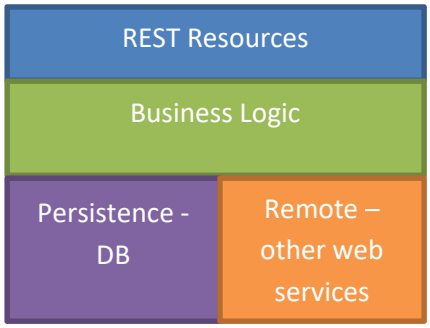
Review

- Class5 Homework and Assignment
- Q&A

Break

Microservices and Data

As we saw in Class 1 – there is a presentation layer, business layer and data layer. To present this for a REST microservice:



Generally:

- Database should NOT be shared across Microservices
- The database belongs to a Microservice
- If you need data from some other DB then call its Microservice (the remote service)
- Different Microservices may use different DB that suit them best – "polyglot" system

Let us look at data options – Persistence and Remote

Relational - Entity-Relationships

The "relational data" model using tabular relations[1] first described by Codd.

- One-One (rare – usually combine into same table)
  - One person has one passport
  - A car model is made by one company
  - A pair of jeans has one brand name
  - A book is published by one company
  - An apple comes from one source (a tree)
  - Santa Claus is associated with one holiday
  - A person has one driver's license
- One-Many/Many-One (Most common)
  - One person may have one or more bank accounts
  - One person may have one or more cars
  - One movie has many actors
- Many-Many
  - There are many customers in an online store and there are many products. The invoice/purchase is a many-many relationship

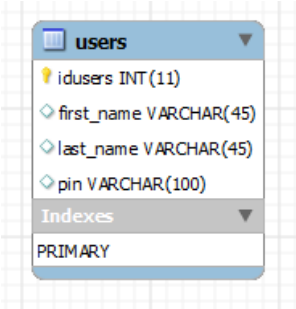
Goal: Organizing your table columns to minimize redundancy and dependency

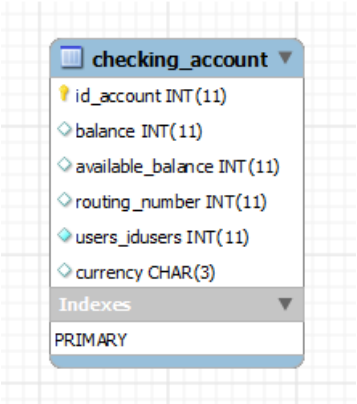
Normalization: http://en.wikipedia.org/wiki/Database_normalization

[1] http://en.wikipedia.org/wiki/Relational_database

ER Diagrams

We will use the MySQL Workbench (see installation in last class homework). Here is an example of two DBs for User Service and Checking Account Service:





What if:

- We need to keep a record of transactions for each account so we can see a statement of transactions and not just the balance. How could be accomplish that?
- How can we keep track of the check deposits ?

What about "NoSQL"?

More accurately these are non-tabular relational models and not a new thing! Generally trees, graphs and key-value pairs are used to define the relationships.

Java Persistence, Hibernate and Spring

The same database maybe used by any language or any framework. Java introduced JPA[2] in JSR-220 and it is an open standard with multiple implementations.

JPA

The Java Persistence Architecture API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects/classes and a relational database. JPA was defined as part of the EJB 3.0 and is now considered the standard industry approach for Object to Relational Mapping (ORM) in the Java.

[2] http://en.wikipedia.org/wiki/Java_Persistence_API

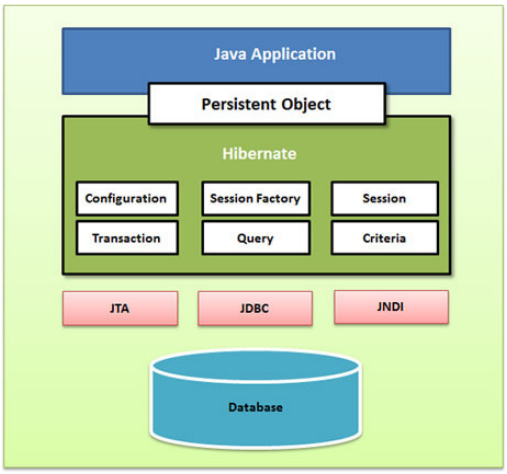
JPA itself is just a specification, not a product; it cannot perform persistence or anything else by itself. JPA is just a set of interfaces, and requires an implementation. There are open-source and commercial JPA implementations. JPA allows POJO (Plain Old Java Objects) to be easily persisted. JPA allows an object's object-relational mappings to be defined through standard annotations or XML defining how the Java class maps to a relational database table.

Hibernate is one implementation of JPA and a very popular choice.

Hibernate[3]

Hibernate takes care of the mapping from Java classes to database tables and also provides data query and retrieval facilities. It can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC. Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier.

Look in detail:



Configuration - The Configuration object is the first Hibernate object you create in any Hibernate application and usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate. The Configuration object provides two keys components:

- Database Connection: This is handled through one or more configuration files supported by Hibernate. These files are hibernate.properties and hibernate.cfg.xml (we will use the Spring wrapper instead)

[3] http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single

- Class Mapping Setup: This component creates the connection between the Java classes and database tables (XML or annotation based).

SessionFactory - Configuration object is used to create a SessionFactory object which in turn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application. The SessionFactory is heavyweight object so usually it is created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So if you are using multiple databases then you would have to create multiple SessionFactory objects.

Session - A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object. The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

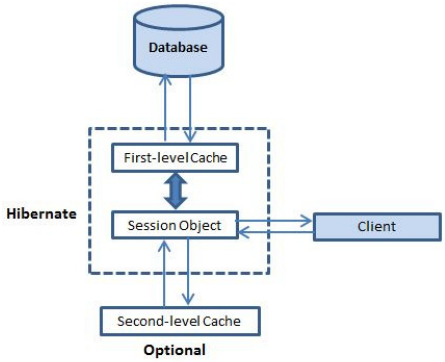
Transaction - A Transaction represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction from JDBC.

Query - Query objects use SQL or Hibernate Query Language (HQL) string to retrieve data from the database and create objects. A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

Criteria - Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

Caching

One of the most important benefits of Hibernate is the cache.



First-level cache (L1): The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database. **If you issue multiple updates to an object, Hibernate tries to delay doing the update as**

long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

Second-level cache (L2): Second level cache is an **optional** cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions. Any third-party cache can be used with Hibernate. An org.hibernate.cache.CacheProvider interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation. EHCache, JBoss Cache etc are some of these third party providers. You can also use distributed cache like Memcache, Oracle Coherence etc.

Query-level cache: Hibernate also implements a cache for query result sets that integrates closely with the second-level cache. This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

Entities and Search

Entities and relationship maybe defined via annotations or in an XML file (similar to Spring bean definitions!)

Queries – this is a very important aspect of any application and efficient queries can make or break an application. In Hibernate there are 3 ways to define queries: HQL, Criteria and Native SQL.

Native – as you may have guessed this is native SQL. This may end up being specific to Oracle/MySQL/MS-SQL etc so your application may not be portable across databases (in reality that is not an issue). You may end up not using the Hibernate caching capabilities so that is the major disadvantage of this approach.

HQL – Hibernate query language – similar to native but with its own nuances

Criteria – Queries are written in Java with a SQL like syntax.

```
Criteria criteria = session.createCriteria(StockDailyRecord.class);
criteria.add(Restrictions.like("stockName", "GOOG"));
criteria.add(Restrictions.eq("volume", 10000));
criteria.add(Restrictions.between("date", startDate, endDate));
criteria.addOrder(Order.asc("date"));
```

Spring Data

The spring-data provides a Spring wrapper over Hibernate and provide transaction management. We will look at this in our example. Spring data also provides a wrapper over other DBs like Mongo, Redis etc. Spring Data provides:

- Standard Crud Repository

- Paging/Sorting support
- Custom Finder method - converted to SQL – based on method name

We will be using this in our examples as the crud repository and finder methods are easy to implement on a simple Microservice. If you have multiple tables then a more complex and custom repository may be necessary.

Break

Example – TBTF Bank

Using JPA annotations and Spring JDBC template

Example implementation: user-serivce-2. Let us look at this code.

- Repository/DAO

Test dependency on Database – Strictly speaking we are not "unit testing" as we are using external dependencies so it is an end-to-end test. You can Mock out the repository interface and eliminate this external database dependency.

Using external service: checking-account-service-2

- RestTemplate

Homework for Class 6

Integrate DB into your services