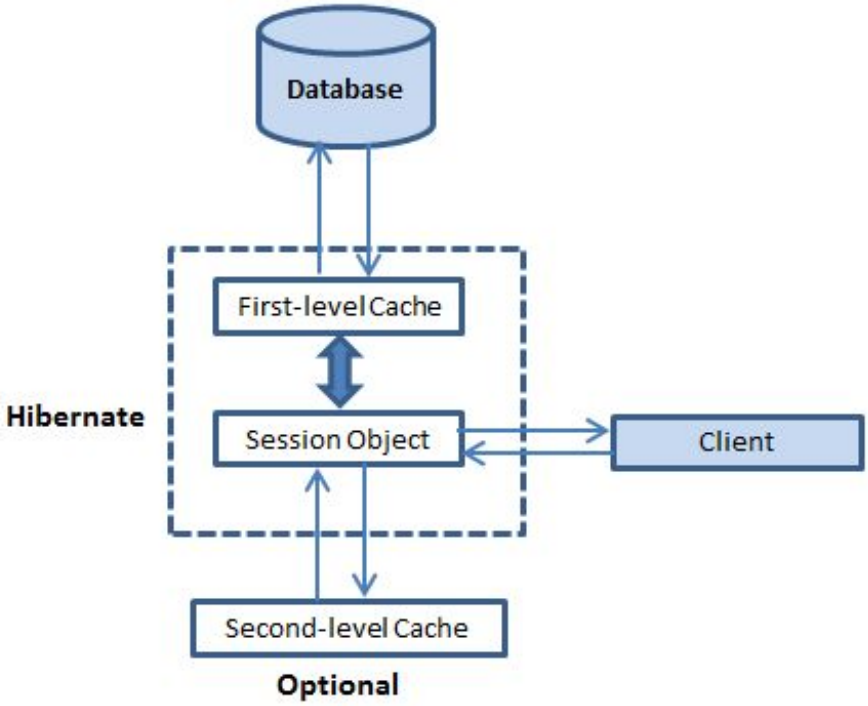
HIBERNATE CACHING

Caching is all about application performance optimization and it sits between your application and the database to avoid the number of database hits as many as possible to give a better performance for performance critical applications.

Caching is important to Hibernate as well which utilizes a multilevel caching schemes as explained below:



First-level cache:

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:

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.

Query-level cache:

Hibernate also implements a cache for query resultsets 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.

The Second Level Cache:

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

Concurrency strategies:

A concurrency strategy is a mediator which responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

- **Transactional:** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Read-write:** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions, in the rare case of an update.
- **Nonstrict-read-write:** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.
- **Read-only:** A concurrency strategy suitable for data which never changes. Use it for reference data only.

If we are going to use second-level caching for our **Employee** class, let us add the mapping element required to tell Hibernate to cache Employee instances using read-write strategy.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD//EN"
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <cache usage="read-write"/>
      <id name="id" type="int" column="id">
         <generator />
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

Cache provider:

Your next step after considering the concurrency strategies you will use for your cache candidate classes is to pick a cache provider. Hibernate forces you to choose a single cache provider for the whole application.

S.N.	Cache Name	Description
1	EHCache	It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache.
2	OSCache	Supports caching to memory and disk in a single JVM, with a rich set of expiration policies and query cache support.

3	warmCache	A cluster cache based on JGroups. It uses clustered invalidation but doesn't support the Hibernate query cache
4	JBoss Cache	A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported

Every cache provider is not compatible with every concurrency strategy. The following compatibility matrix will help you choose an appropriate combination.

Strategy/Provider	Read-only	Nonstrictread-write	Read-write	Transactional
EHCache	X	X	X	
OSCache	X	X	X	
SwarmCache	X	X		
JBoss Cache	X			X

You will specify a cache provider in hibernate.cfg.xml configuration file. We choose EHCache as our second-level cache provider:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>
   <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

Now, you need to specify the properties of the cache regions. EHCache has its own configuration file, **ehcache.xml**, which should be in the CLASSPATH of the application. A cache configuration in ehcache.xml for the Employee class may look like this:

```
<diskStore path="java.io.tmpdir"/>
<defaultCache
maxElementsInMemory="1000"
```

```
eternal="false"
timeToIdleSeconds="120"
timeToLiveSeconds="120"
overflowToDisk="true"
/>

<cache name="Employee"
maxElementsInMemory="500"
eternal="true"
timeToIdleSeconds="0"
timeToLiveSeconds="0"
overflowToDisk="false"
/>
```

That's it, now we have second-level caching enabled for the Employee class and Hibernate now hits the second-level cache whenever you navigate to a Employee or when you load a Employee by identifier.

You should analyze your all the classes and choose appropriate caching strategy for each of the classes. Sometime, second-level caching may downgrade the performance of the application. So it is recommended to benchmark your application first without enabling caching and later on enable your well suited caching and check the performance. If caching is not improving system performance then there is no point in enabling any type of caching.

The Query-level Cache:

To use the query cache, you must first activate it using the **hibernate.cache.use_query_cache="true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Next, to use the query cache, you use the setCacheable(Boolean) method of the Query class. For example:

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
List users = query.list();
SessionFactory.closeSession();
```

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
query.setCacheRegion("employee");
List users = query.list();
SessionFactory.closeSession();
```

This code uses the method to tell Hibernate to store and look for the query in the employee area of the cache.