JVA-000
Java Persistence with Hibernate

Module 7
Hibernate JPA Performance Tips

Objectives

- Observe performance issues the JPA can bring up
- Learn hot to avoid performance issue in JPA application
- Observer JPA and Hibernate specific facilities for improving application performance

The double-edged sword of JPA

A great thing about JPA is that it abstracts your interaction with the underlying database…

… you can write database access code very easily and get most of the general database operations out of the box without having to write all that tedious JDBC code

A bad thing about JPA is that it abstracts your interaction with the underlying database…

…you need to have some knowledge of what's going on behind the scenes or you will be in for some unpleasant surprises

Performance areas

What you need to know about JPA to avoid performance issues:

- What is Lazy Loading and its pros/cons
- How to use Fetching Strategies
- What is Pagination and how to use it
- Caches (1L, 2L and query) and Caching
- Performance monitoring

Lazy Loading

Lazy loading means delaying the loading of related data, until you specifically request it

Hibernate supports fetch types:

- `FetchType.LAZY`
- `FetchType.EAGER`

Used in:

- `@ManyToMany, @ManyToOne, @OneToMany, @OneToOne`
- `@Basic`

Lazy Loading - Example

Example: `Department` entity contains `Company` as a complex property/field.

```
@Entity
public class Company {
    @Id
    private String name;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}

@Entity
public class Department {
    @Id
    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    private Company company;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public Company getCompany() { return company; }
    public void setCompany(Company company) { this.company = company; }
}
```

An many-to-one association has fetch type defined as LAZY.

It means the additional SELECT will be issued for loading Company data from db.

Lazy Loading - Example

If we load a `Department` entity:

```
Department department = em.find(Department.class, "IT");
```

The following SQL statement will be executed:

```
select
    department0_.name as name1_1_0_,
    department0_.company_name as company_2_1_0_
from
    Department department0_
where
    department0_.name=?
```

After access the `company` field the SQL statement will be run:

```
select
    company0_.name as name1_0_0_
from
    Company company0_
where
    company0_.name=?
```

Eager Loading - Example

If we load a `Department` entity:

```
Department department = em.find(Department.class, "IT");
```

The following SQL statement will be executed:

```
select
    department0_.name as name1_1_0_,
    department0_.company_name as company_2_1_0_,
    company1_.name as name1_0_1_
from
    Department department0_
left outer join
    Company company1_
        on department0_.company_name=company1_.name
where
    department0_.name=?
```

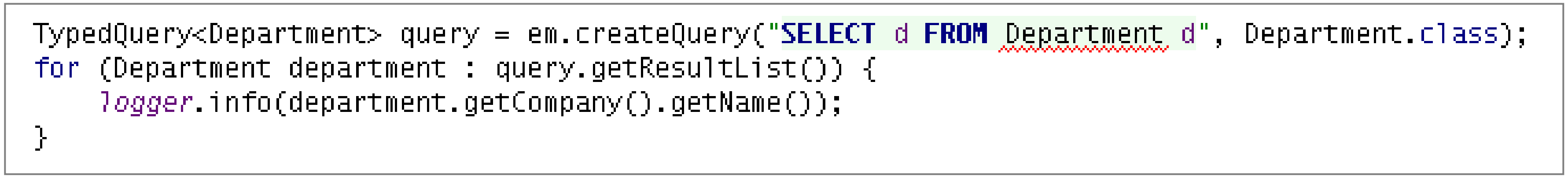
And no additional queries will be run for loading data.

Loading collections – Batch Fetch

There is N+1 issue when we are working with collections properties

Running the following code produces the N+1 SQL statements to be run against database:

- One to load list of `Department` entities
- N to load Country for each `Department` entity

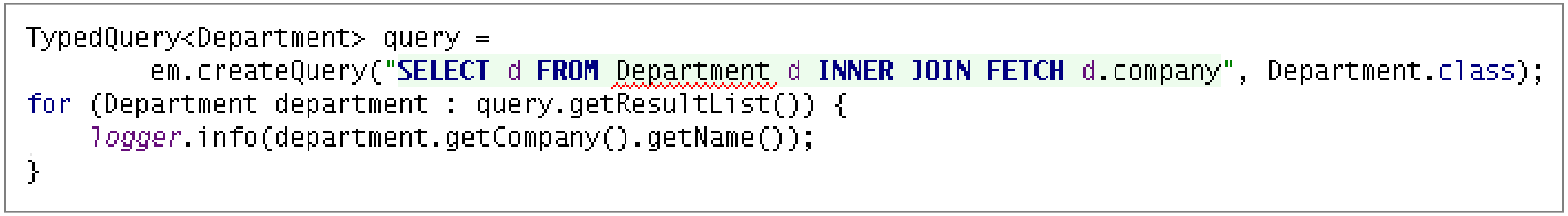


```
TypedQuery<Department> query = em.createQuery("SELECT d FROM Department d", Department.class);
for (Department department : query.getResultList()) {
    logger.info(department.getCompany().getName());
}
```

Loading collections – Batch Fetch

To avoid N+1 issue you can use **Batch Fetch** technique (when loading `Department` entities)

Use JPQL `JOIN FETCH` instruction to enable batch fetch when using JPA



```
TypedQuery<Department> query =
        em.createQuery("SELECT d FROM Department d INNER JOIN FETCH d.company", Department.class);
for (Department department : query.getResultList()) {
    logger.info(department.getCompany().getName());
}
```

This forces Hibernate to use single SQL statement with `JOIN` to load the data and use them for entities construction

Pagination

Paginating your results is probably one of the best ways to increase the performance of your JPA application

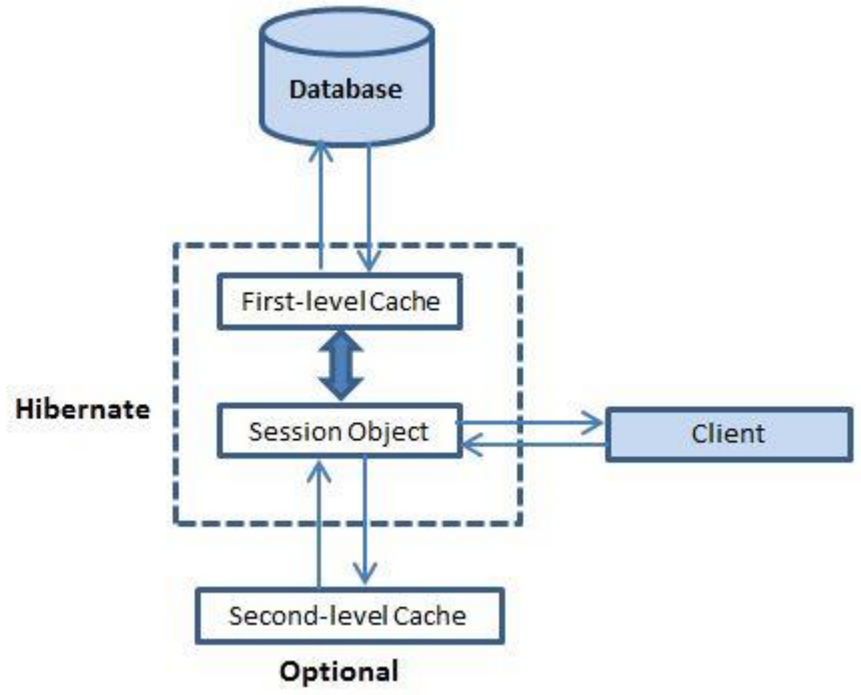
Pagination is supported in JPA via the Query `setFirstResult()` and `setMaxResults()` methods

This avoids:

- Having to read the entire database
- And also should (in theory) make the persistence context more optimized by reducing the number of objects it needs to process together

Caching

Cache sits between application and database to avoid the number of database hits as many as possible to give a better performance for application.



Caching – 1L Cache

The **first-level cache** is the session cache and is a **mandatory cache** through which all requests must pass (cannot be disabled)

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.

Caching – 2L Cache

Second-level cache is an optional cache

The **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 **responsible for caching objects across sessions**.

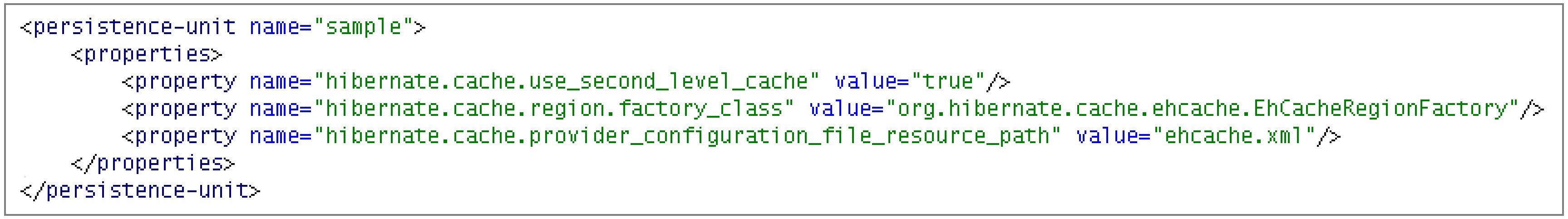
Any third-party cache can be used with Hibernate (ex. Ehcache)

Caching – 2L Cache

The second-level cache **implementation follows** the `org.hibernate.cache.spi.RegionFactory` Java interface

Configuration properties:

- `hibernate.cache.region.factory_class` – cache implementation class
- `hibernate.cache.use_query_cache` – `true/false` to enable/disable using of the second-level cache



```xml
<persistence-unit name="sample">
    <properties>
        <property name="hibernate.cache.use_second_level_cache" value="true"/>
        <property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcache.EhCacheRegionFactory"/>
        <property name="hibernate.cache.provider_configuration_file_resource_path" value="ehcache.xml"/>
    </properties>
</persistence-unit>
```

Caching – 2L Cache

By default, entities are not part of the second level cache.

Entity caching is managed with help of:

- `@Cacheable` annotation for an entity class
- Cache mode (to be defined for Persistence Unit)

`@Cacheable` – specifies whether an entity should be cached if caching is enabled:

- `TRUE` – entity should be cached
- `FALSE` – entity should not be cached

Caching – 2L Cache

Cache modes:

- `ALL` – all entities will be cached
- `NONE` – no entities will be cached
- `ENABLE_SELECTIVE` – entities are not cached unless explicitly marked as cacheable *(default)*
- `DISABLE_SELECTIVE` – entities are cached unless explicitly marked as not cacheable

Caching – 2L Cache

The second-level cache also **can be accessed programmatically** via `javax.persistence.Cache`

- `contains(Class,Object)` – checks if cache contains the given entity
- `evict(Class,Object)` – removes given entity from cache
- `evict(Class)` – removes entities of given type from cache
- `evictAll()` – clears the cache

Reference to `Cache` can be obtained via call of `EntityManagerFactory.getCache()` method.

Caching – Query Cache

Hibernate implements a cache for query results that integrates closely with the second-level cache

Query cache keeps identifiers of entities affected by the query. Then the second-level cache is used to get the entities by the cached identifiers.

This is an optional feature and requires two additional physical cache regions that hold:

- Cached query results
  (org.hibernate.cache.internal.StandardQueryCache)
- Timestamps when table was last updated
  (org.hibernate.cache.spi.UpdateTimestampsCache)

Caching – Query Cache

Useful for queries that are run frequently with the same parameters

When entity gets updated the cached query results for this entity become invalid and need to be reloaded from database

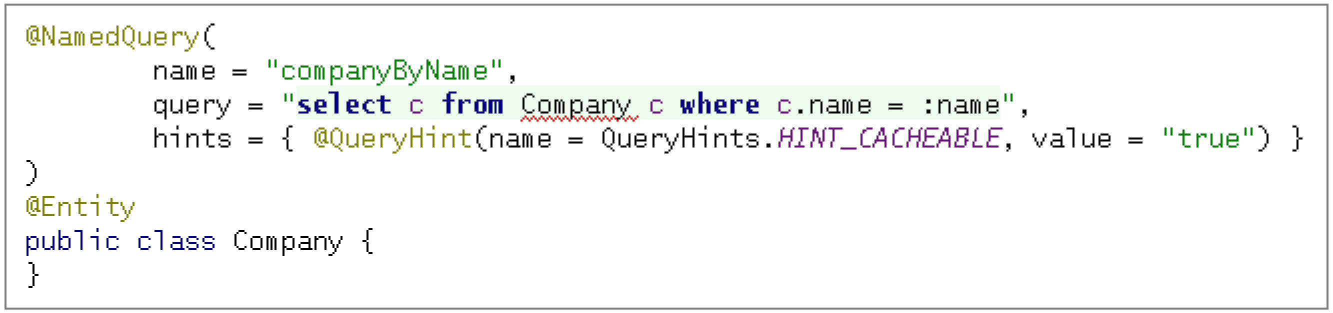
The `hibernate.cache.use_query_cache` property is used to enable/disable query cache (`true`/`false`)

Caching – Query Cache

To let Hibernate know that query results should be cached the hint `org.hibernate.cacheable=TRUE` should be specified for the query

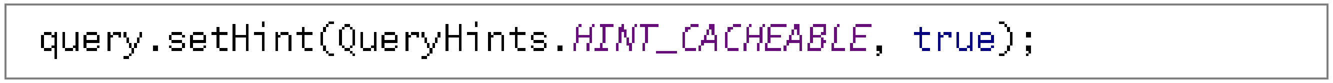
Ways to specify a hint:

- Specify `hints` attribute for `@NamedQuery` annotation



```
@NamedQuery(
        name = "companyByName",
        query = "select c from Company c where c.name = :name",
        hints = { @QueryHint(name = QueryHints.HINT_CACHEABLE, value = "true") }
)
@Entity
public class Company {
}
```

- Use `setHint()` method for the `Query` object



```
query.setHint(QueryHints.HINT_CACHEABLE, true);
```

Metrics

Hibernate provides a number of metrics

All available counters are described in the `org.hibernate.stat.Statistics` interface, in three categories:

- Metrics related to the general usage (such as number of open sessions, retrieved JDBC connections, etc.)
- Metrics related to the entities, collections, queries, and caches as a whole
- Detailed metrics related to a particular entity, collection, query or cache region

Property `hibernate.generate_statistics` is used to enable/disable statistic gathering by Hibernate

Metrics

Example of how to obtain Hibernate `Statistics` object and print caches usage information:

```
private static void printHibernateStatistic(EntityManagerFactory emf) {
    final Statistics stat =
            ((org.hibernate.jpa.internal.EntityManagerFactoryImpl) emf).getSessionFactory().getStatistics();

    logger.info("StatisticsEnabled=" + stat.isStatisticsEnabled());

    logger.info("QueryCacheHitCount=" + stat.getQueryCacheHitCount());
    logger.info("QueryCacheMissCount=" + stat.getQueryCacheMissCount());
    logger.info("QueryCachePutCount=" + stat.getQueryCachePutCount());

    logger.info("SecondLevelCacheHitCount=" + stat.getSecondLevelCacheHitCount());
    logger.info("SecondLevelCacheMissCount=" + stat.getSecondLevelCacheMissCount());
    logger.info("SecondLevelCachePutCount=" + stat.getSecondLevelCachePutCount());
}
```