
JBoss
Community

Hibernate ORM:
Tips, Tricks, and Performance Techniques

Brett Meyer

Senior Software Engineer

Hibernate ORM, Red Hat

HIBERNATE

JBoss Community

Brett Meyer

- Hibernate ORM
  - ORM 4 & 5 development
  - Hibernate OSGi
  - Developer community engagement
  - Red Hat enterprise support, Hibernate engineering lead
- Other contributions
  - Apache Camel
  - Infinispan
- Contact me
  - @brettemeyer or +brettmeyer
  - Freenode #hibernate or #hibernate-dev (brmeyer)

github.com/brmeyer
/HibernateDemos

slideshare.net/brmeyer

ORM? JPA?

- ORM: Object/Relational Mapping
  - Persistence: Data objects outlive the JVM app
  - Maps Java POJOs to relational databases
  - Supports OO concepts: inheritance, object identity, etc.
  - Navigate data by walking the object graph, not the explicit relational model
- JPA: Java Persistence API
- Hibernate ORM provides its own native API, in addition to full JPA support
- Annotations and XML

Overview

- Fetching Strategies
- 2nd Level Entity Cache
- Query Cache
- Cache Management
- Bytecode Enhancement
- Hibernate Search
- Misc. Tips
- **Q&A**

Caveats

- No “one strategy to rule them all”
- Varies greatly between apps
- Important to understand concepts, then apply as necessary
- Does not replace database tuning!

First, the antithesis:

```
public User get(String username) {
        final Session session = openSession();
        session.getTransaction().begin();
        final User user = (User) session.get( User.class, username );
        session.getTransaction().commit();
        return user;
}
public boolean login(String username) {
        return get(username) != null;
}
```

Clean? Yes. But...

EAGER Demo

- Prototypes start “simple”
  - EAGER
  - No caching
  - Overuse of the kitchen sink
- DAO looks clean
- Then you see the SQL logs

Fetching Strategies

Fetching Strategies

- By far, the most frequent mistake
- Also the most costly
- 2 concepts:
  - **WHEN** (fetch timing)
  - **HOW** (fetch style)

WHEN (fetch timing)

Fetch Timing: **EAGER**

- Immediate
- Can be convenient (in some ways)
- Significantly increases payload
- Enormous amount of unnecessary fetches for deep association tree

Fetch Timing: **LAZY**

- Fetched when first accessed
- Collections
  - LAZY by default
  - Utilizes Hibernate's internal concept of “persistent collections”
  - DEMO
- Single attribute (basic)
  - Requires bytecode enhancement
  - Not typically used nor beneficial

Fetch Timing: **LAZY** (cont'd)

- Single (ToOne) associations
  - Fetched when accessed
  - Proxy
    - Default
    - Fetched when accessed (except ID)
    - Handled internally by Hibernate
  - No-proxy
    - Fetched when accessed (including ID)
    - No visible proxy
    - Requires buildtime bytecode enhancement

Fetch Timing: **EXTRA LAZY**

- Collections only
- Fetch individual elements as accessed
- Does not fetch entire collection
- DEMO

HOW (fetch style)

Fetch Style: **JOIN**

- Join fetch (left/outer join)
- Great for ToOne associations
- Multiple collection joins
  - Possible for non-bags
  - Warning: Cartesian product! SELECT is normally faster
- DEMO

Fetch Style: **SELECT**

- Follow-up selects
- Default style for collections (avoids cartesian products)
- Can result in 1+N selects (fetch per collection entry)
- DEMO

Fetch Style: **BATCH**

- Fetches multiple entries when one is accessed
- Configurable on both class and collection levels (“batch-size”)
- Simple select and list of keys
- Multiple algorithms (new as of 4.2)
- Determines # of entries to fetch, based on # of provided keys

Fetch Style: **BATCH** (cont'd)

- **Legacy**: pre-determined sizes, rounded down
  - batch-size==25, 24 elements in collection
  - Fetches -> 12, 10, 2
- **Padded**: same as Legacy, size rounded up
- **Dynamic**: builds SQL for the # of keys, limited by “batch-size”
- DEMO

Fetch Style: **SUBSELECT**

- Follow-up select
- Fetches **all** collection entries when accessed for the 1st time
- Original root entry select used as subselect
- Performance depends on the DB itself
- DEMO

Fetch Style: **PROFILE**

- named profile defining fetch styles
- “activated” through the Session

```
@Entity
@FetchProfile(name = "customer-with-orders", fetchOverrides = {
        @FetchProfile.FetchOverride(entity = Customer.class,
                association = "orders", mode = FetchMode.JOIN)
})
public class Customer {
   ...

   @OneToMany
   private Set<Order> orders;
   ...
}

Session session = ...;
session.enableFetchProfile( "customer-with-orders" );
Customer customer = (Customer) session.get(
        Customer.class, customerId );
```

Fetch Style Tips

- Best to leave defaults in mappings
- Define the style at the query level
- Granular control

2nd Level Entity Cache (2LC)

2LC

- differs from the 1st level (Session)
- SessionFactory level (JVM)
- can scale horizontally on commodity hardware (clustered)
- multiple providers
- currently focus on Infinispan (JBoss) & EHCache (Terracotta)

2LC (cont'd)

- disabled by default
  - can enable globally (not recommended)
  - enable for individual entities and collections
  - configurable at the Session level w/ CacheMode
- cache all properties (default) or only non-lazy

2LC (cont'd)

- Since JVM-level, concurrency is an issue
- Concurrency strategies
  - Read only: simplest and optimal
  - Read-write
    - Supports updates
    - Should not use for serializable transaction isolation
  - Nonstrict-read-write
    - If updates are occasional and unlikely to collide
    - No strict transaction isolation
  - Transactional:
    - Fully transactional cache providers (ie, Infinispan and EHCache)
    - JTA required

2LC (cont'd)

- DEMO

Query Cache

Query Cache

- Caches query result sets
- Repetitive queries with identical params
- Different from an entity cache
  - Does not maintain entity state
  - Holds on to sets of identifiers
- If used, best in conjunction with 2LC

Query Cache (cont'd)

- Disabled by default
- Many applications do not gain anything
- Queries invalidated upon relevant updates
- Increases overhead by some: tracks when to invalidate based on commits
- DEMO

Cache Management

Cache Management

- Entities stored in Session cache when saved, updated, or retrieved
- Session#flush syncs all cached with DB
  - Minimize usage
- Manage memory:
  - Session#evict(entity) when no longer needed
  - Session#clear for all
  - Important for large dataset handling
- Same for 2LC, but methods on SessionFactory#getCache

Bytecode Enhancement

Bytecode Enhancement

- Not just for no-proxy, ToOne laziness
- Reduced load on PersistenceContext
  - EntityEntry
    - Internally provides state & Session association
    - “Heavy” operation and typically a hotspot (multiple Maps)
  - ManagedEntity
    - Reduced memory and CPU loads
    - Entities maintain their own state with bytecode enhancement
    - (ManagedEntity)entity.$$_hibernate_getEntityEntry();
  - 3 ways:
    - Entity implements ManagedEntity and maintains the association
    - Buildtime instrumentation (Ant and recently Gradle/Maven)
    - Runtime instrumentation

Bytecode (cont'd)

- dirtiness checking
  - Legacy: “dirtiness” determined by deep comparison of entity state
  - Enhancement: entity tracks its own dirtiness
  - Simply check the flag (no deep comparison)
  - Already mentioned…
    - Minimize amount of flushes to begin with
    - Minimize the # of entities in the cache -- evict when possible
- Many additional bytecode improvements planned

Hibernate Search

Hibernate Search

- Full-text search on the DB
  - Bad performance
  - CPU/IO overhead
- Offload full-text queries to Hibernate Search engine
  - Fully indexed
  - Horizontally scalable
- Based on Apache Lucene

Hibernate Search (cont'd)

- Annotate entities with @Indexed
- Annotate properties with @Field
  - Index the text: index=Index.YES
  - “Analyze” the text: analyze=Analyze.YES
    - Lucene analyzer
    - Chunks sentences into words
    - Lowercase all of them
    - Exclude common words (“a”, “the”)
- Combo of indexing and analysis == performant full-text searches!

Misc. Tips

Misc. Tips

- Easy to overlook unintended fetches
  - Ex: Fully implemented toString with all associations (fetches everything simply for logging)
- Use @Immutable when possible
  - Excludes entity from dirtiness checks
  - Other internal optimizations

Misc. Tips (cont'd)

- Use Bag or List for inverse collections, not Set
  - Collection#add & #addAll always return true (don't need to check for pre-existing values)
  - I.e., can add a value without fetching the collection

```
Parent p = (Parent) session.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.addChild(c);
...
// does *not* fetch the collection
p.getChildren().add(c);
```

Misc. Tips (cont'd)

- One-shot delete (non-inverse collections)
  - Scenario: 20 elements, need to delete 18 & add 3
  - Default: Hibernate would delete the 18 one-by-one, then add the 3
  - Instead:
    - Discard the entire collection (deletes all elements)
    - Save a new collection with 5 elements
    - 1 bulk delete SQL and 5 inserts
  - Important concept for large amounts of data

How to Help:

hibernate.org /orm/contribute

QUESTIONS?

- Q&A
- #hibernate or #hibernate-dev (brmeyer)
- @brettemeyer
- +brettmeyer