Effiziente Persistierung mit Hibernate

JUG Saxony 2016/02/25

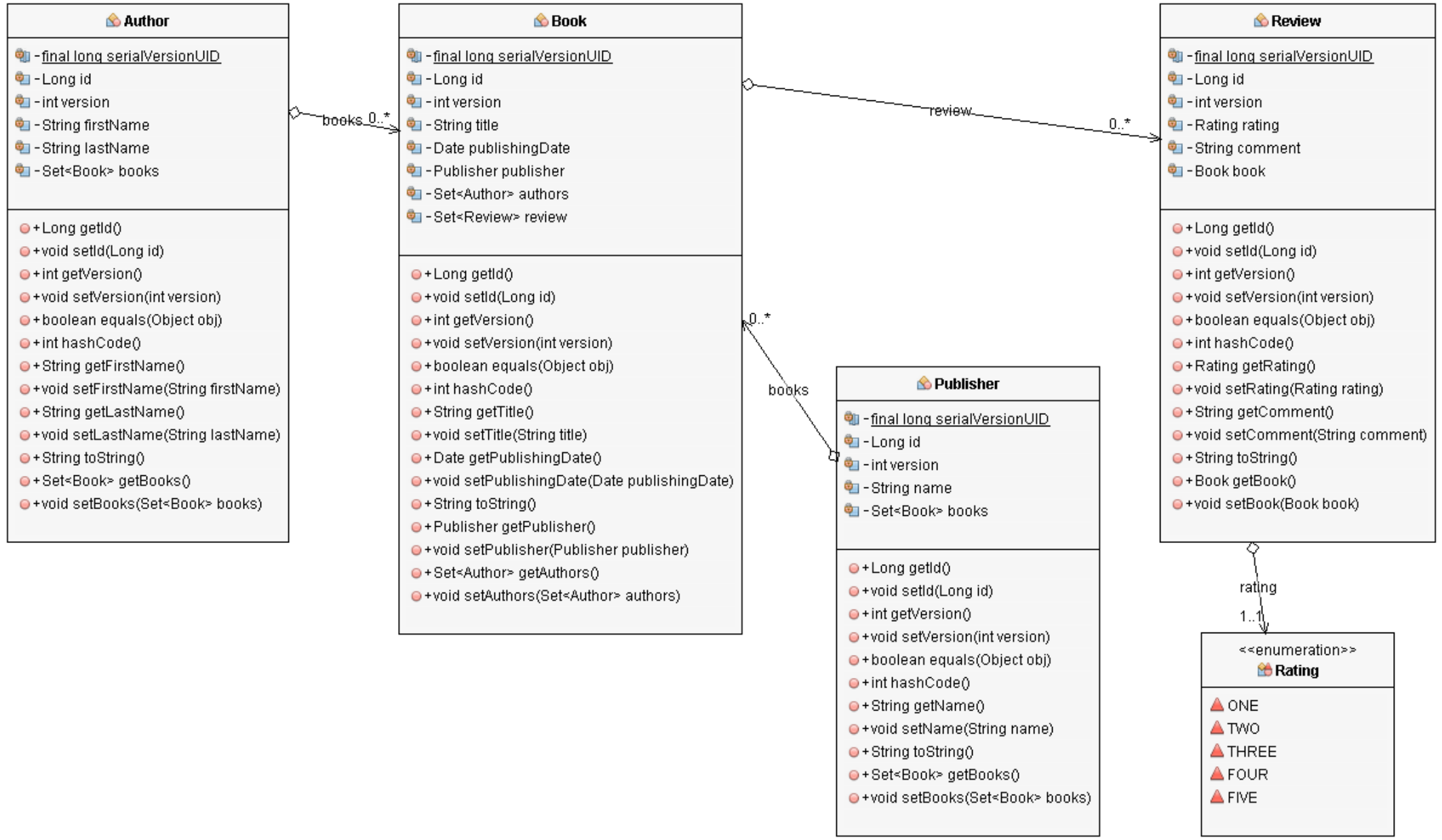
Thorben Janssen

- Independent author and trainer
- Senior developer and architect @ Qualitype GmbH

- CDI 2.0 expert group member

- Twitter: @thjanssen123
- Blog: www.thoughts-on-java.org

Application



Performance

Performance

- Recognize performance problems as early as possible
- Typical causes for performance problems
- Solving performance problems

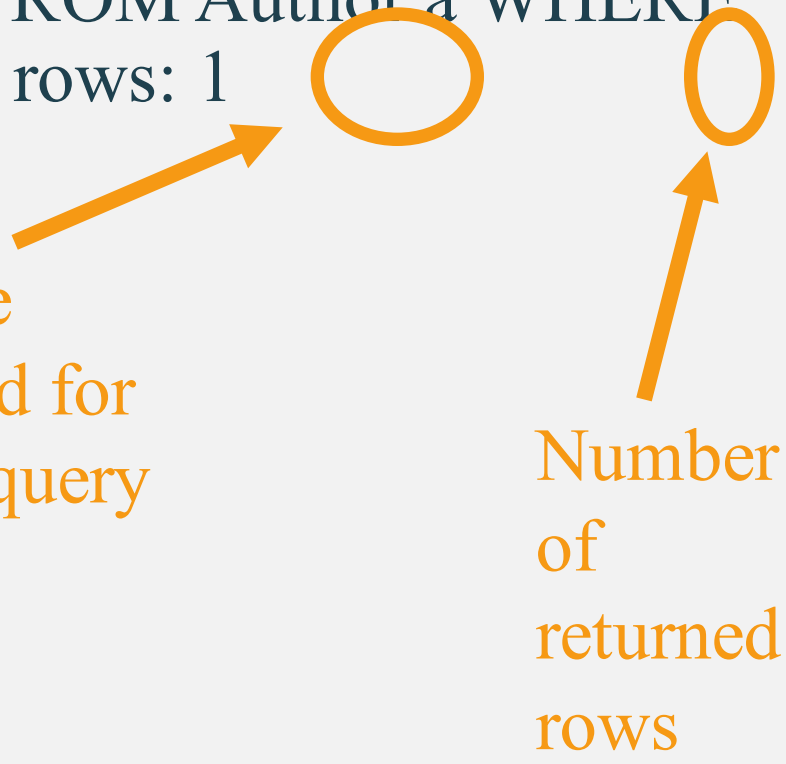
Hibernate Statistics

Hibernate Statistics

- Activate via system property
  - *hibernate.generate_statistics = true*
- Configure logging
  - *org.hibernate.stat = DEBUG*

Hibernate Statistics

```
08:24:10,916 DEBUG
[org.hibernate.stat.internal.ConcurrentStatisticsImpl] (default task-
1) HHH000117: HQL: SELECT a FROM Author a WHERE
a.lastName = :lastName, time: 6ms, rows: 1
```



Hibernate Statistics

```
08:08:04,960 INFO
[org.hibernate.engine.internal.StatisticalLoggingSessionEventListener]
(default task-2) Session Metrics {
    1191522 nanoseconds spent acquiring 4 JDBC connections;
    433875 nanoseconds spent releasing 4 JDBC connections;
    4404058 nanoseconds spent preparing 6 JDBC statements;
    12458725 nanoseconds spent executing 6 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    586896 nanoseconds spent executing 1 flushes (flushing a total of 2
entities and 2 collections);
    39434974 nanoseconds spent executing 1 partial-flushes (flushing a total
of 2 entities and 2 collections)
```

Time spend for SQL state-ments

Number of SQL statements

Cache usage

Typical causes for performance problems

Typical Causes

- Slow SELECT statements
- Wrong FetchType
- Load same data multiple times
- Delete or edit records one by one
- Put logic into the application instead of the database/query

Slow SELECT-statements

Slow SELECTs

- In general no „real“ JPA or Hibernate problem
  - Check generated SQL
  - Check execution plan of the statement
  - Check indexes
  - Optimize SELECT statement
    - Consider to use a native query

Native Query

- Reasons to use native queries:
  - JPQL supports only a subset of SQL
  - Database specific features
- Native queries return an Object[] for each row
  - Needs to be mapped programmatically or declaratively

ResultSetMapping

- Declarative mapping of query results

```
@SqlResultSetMapping(
        name        = "myResultMapping ",
        entities    = {@EntityResult(...), …},
        classes     = {@ConstructorResult (…), …},
        columns     = {@ColumnResult(…), …}
)
```

```
this.em.createNativeQuery(“Select …", "myResultMapping")
```

ResultSetMapping

```
@SqlResultSetMapping(
        name            = "myResultMapping",
        entities        = {@EntityResult(
                                entityClass     = Author.class,
                                fields          = {
                                        @FieldResult(name = "id", column = „autId"),
                                        …}
                                ),
                           …},
        classes         = {@ConstructorResult(…), …},
        columns         = {@ColumnResult(…), …}
)
```

ResultSetMapping

```
@SqlResultSetMapping(
        name            = "myResultMapping",
        entities        = {@EntityResult(...), …},
        classes         = {@ConstructorResult(
                                targetClass    = BookPublisherValue.class,
                                columns        = {
                                        @ColumnResult(name = "title"),
                                        …}
                                ),
                           …},
        columns         = {@ColumnResult(…), …}
)
```

ResultSetMapping

```
@SqlResultSetMapping(
        name            = "myResultMapping",
        entities        = {@EntityResult(...), …},
        classes         = {@ConstructorResult(...), …},
        columns         = {@ColumnResult(
                                name  = "bookCount",
                                type    = Long.class),
                           …},
)
```

FetchType

FetchType

- Defines when the relationship will be fetched
- Static definition in entity mapping

```
@ManyToMany(mappedBy="authors", fetch = FetchType.EAGER)
private Set<Book> books;
```

FetchType

- Lazy
  - Relationship gets loaded at first access
  - Default for to-many relationships
- Eager
  - Loads relationships immediately
  - Default for to-one relationships

Recommendations

- To-many relationships
  - Stick to the default mapping (FetchType.LAZY)
  - Use eager fetching for specific queries, if required
- To-one relationships
  - Check individually
  - Default is fine in most of the cases

Query Specific Fetching

N+1 Select?

- Most common cause for performance problems
- Lazy fetching of related entities creates too many queries

```
List<Author> authors = this.em.createQuery("SELECT a FROM Author a",
                                                    Author.class).getResultList();

for (Author a : authors) {
        System.out.println("Author " + a.getFirstName() + " " + a.getLastName()
                                + " wrote " + a.getBooks().size() + " Books."));
}
```

Query specific fetching

- Fetch all required entities with one query
  - Fetch Joins
  - @NamedEntityGraph
  - EntityGraph

Fetch Join

Fetch Join

- Use JOIN FETCH instead of JOIN in JPQL query

```
List<Author> authors = this.em.createQuery(
                    "SELECT DISTINCT a FROM Author a JOIN FETCH a.books b",
                    Author.class).getResultList();
```

Fetch Join

- Advantages
  - Relationships gets loaded in same query
- Disadvantages
  - Requires a special query for each use case
  - Creates cartesian product

NamedEntityGraph

NamedEntityGraph

- Introduced in JPA 2.1
- Declaratively defines a graph of entities which will be loaded
- Graph is query independent

- Define NamedEntityGraph

```
@NamedEntityGraph(
        name = "graph.AuthorBooksReviews",

        attributeNodes =
                @NamedAttributeNode(value = "books", subgraph = "books"),

        subgraphs =
                @NamedSubgraph(
                        name = "books",
                        attributeNodes = @NamedAttributeNode("reviews")
                )
)
```

- Provide entity graph es hint

```
EntityGraph graph = this.em.getEntityGraph("graph.AuthorBooks");

this.em.createQuery("SELECT DISTINCT a FROM Author a")
                .setHint("javax.persistence.loadgraph", graph);
```

- Fetch graph
  - Eager loading for all elements of the graph
  - Lazy loading for all other attributes

- Load graph
  - Eager loading for all elements of the graph
  - Loads all other attributes with their defined FetchType

- **Hibernate always uses a load graph**
  - HHH-8776

- Advantages
  - Query specific EAGER loading
  - Definition of the graph is independent of the query
- Disadvantages
  - Creates cartesian product

EntityGraph

EntityGraph

- Introduced in JPA 2.1
- Dynamic version of @NamedEntityGraph
- Definition via Java API
- Graph is query independent

- Define and use EntityGraph

```
EntityGraph graph = this.em.createEntityGraph(Author.class);
Subgraph<Book> bookSubGraph = graph.addSubgraph(Author_.books);
bookSubGraph.addSubgraph(Book_.reviews);
```

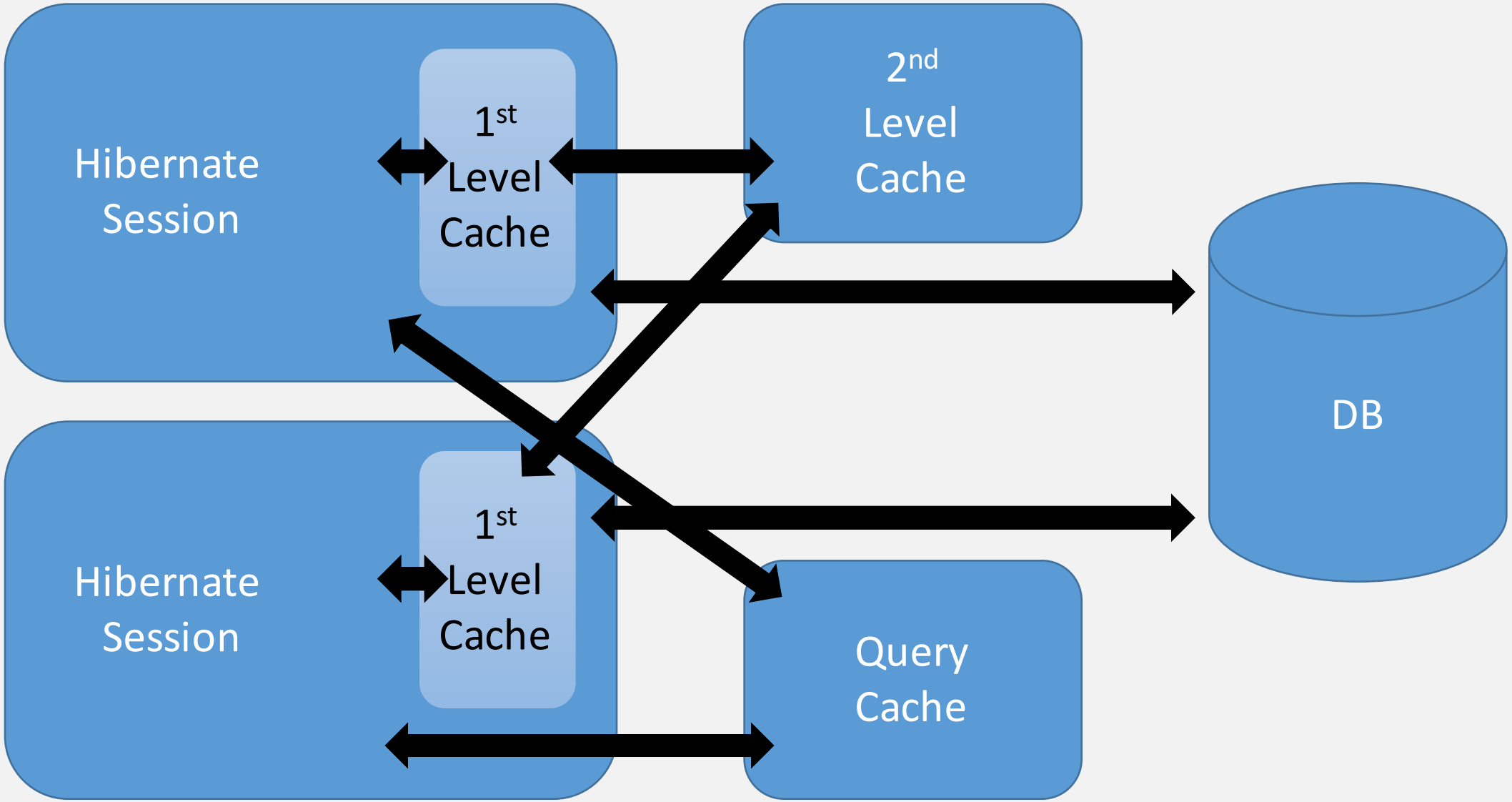
```
this.em.createQuery("SELECT DISTINCT a FROM Author a")
                .setHint("javax.persistence.loadgraph", graph);
```

EntityGraph

- Advantages
  - Query specific EAGER loading
  - Definition of the graph is independent of the query
  - Dynamic creation at runtime
- Disadvantages
  - Creates cartesian product

Caching

Caches


Hibernate Session
1st Level Cache
Hibernate Session
1st Level Cache
2nd Level Cache
Query Cache
DB

1st Level Cache

1st Level Cache

- Activated by default
- Linked to the Hibernate session
- Stores all entities that were used within a session
- Transparent usage

2nd Level Cache

2nd Level Cache

- Session independent entity store
- Needs to be activated
  - persistence.xml or EntityManagerFactory
- Transparent usage
- PersistenceProvider doesn‘t need to provide it
  - Not always portable

2nd Level Cache

- Shared Cache Mode
  - ALL — cache all entities
  - NONE — cache no entities
  - ENABLE_SELECTIVE — cache needs to be activated for specific entities
  - DISABLE_SELECTIVE — cache can be deactivated for specific entities
  - UNSPECIFIED — use default settings of the PersistenceProvider

2nd Level Cache

- Cache configuration
  - Cache Retrieve Mode
    - How to read entities from the cache
  - Cache Store Mode
    - How to write entities to the cache
  - Concurrency Strategy
    - How to handle concurrent access

Query Cache

Query Cache

- Hibernate specific
- Stores query result session independent
- Needs to be activated (persistence.xml)
  - hibernate.cache.use_query_cache = true
- Activate caching for a specific query
  - org.hibernate.Query.setCacheable(true)
  - @NamedQuery(… hints = @QueryHint(name="org.hibernate.cacheable", value="true"))

Query Cache

- Stores query results for a query and its parameters
  - [„FROM Author WHERE id=?“, 1] → [1]
- Stores only entity references or scalars
  - Always use together with 2nd Level Cache

Recommendations

- Only cache data that is seldom updated
- Always benchmark the application when adding or changing caching
- Use Query Cache together with 2nd Level Cache
  - Configuration has to fit to each other

Resources

Resources

- JSR 338: JavaTM Persistence API, Version 2.1 http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html
- Hibernate Reference Documentation http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/
- Hibernate Developer Guide http://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/
- Hibernate ORM: Tips, Tricks and Performance Techniques by Brett Meyer http://de.slideshare.net/brmeyer/hibernate-orm-performance-31550150

Resources

- Java Persistence with Hibernate Second Edition by Christian Bauer, Gaving King, Gary Gregory
- Java Platform, Enterprise Edition: The Java EE Tutorial https://docs.oracle.com/javaee/7/tutorial/index.html
- Hibernate: Truly Understanding the Second-Level and Query Caches http://www.javalobby.org/java/forums/t48846.html

Slides & Cheat Sheet

www.thoughts-on-java.org

Slides & Cheat Sheet

Get this presentation and a handy cheat sheet with all the discussed performance tuning tips at

http://www.thoughts-on-java.org/hpt-jug-saxony/