JPA Best Practices

Bruce Campbell

Outline

- mapping approaches
- primary keys
- sequences/generators
- orphans
- equals and hashcode
- fetch types/n+1 queries
- eager vs. lazy loading
- concurrency

Mapping Approaches

- Two approaches to mapping
  - Schema driven design
    - Build the object model based upon the database
  - Object Oriented driven design
    - Hibernate can generate database creation scripts from your object model

Mapping Approaches

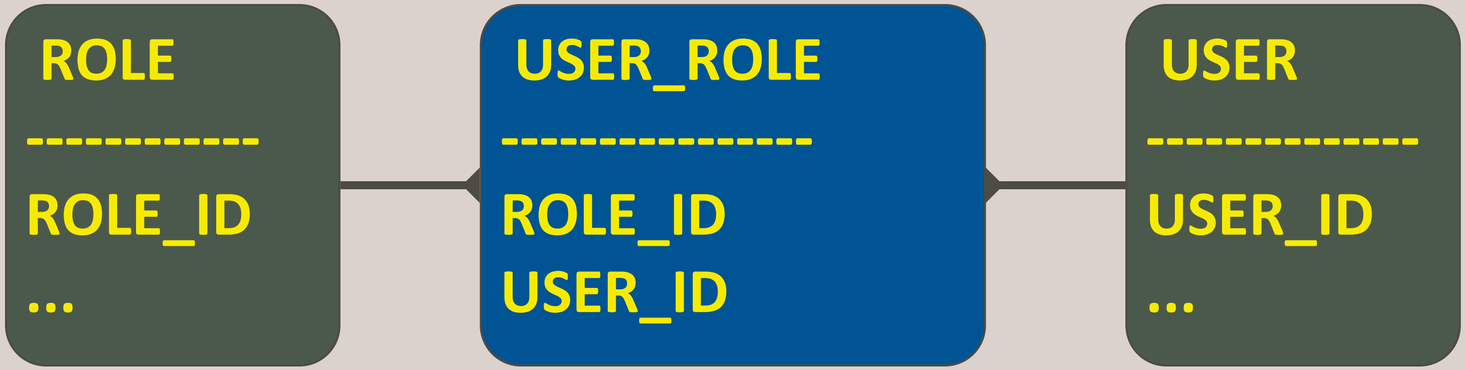
- avoid automatic schema generation if you
  - have an existing database
  - have certain database naming standards or data modeling best practices to conform to
  - want total control over the schema
- also note that
  - some claim the generated schema quality is sub-par
  - additional mapping work is required to specify objects names if you don't like the defaults

Primary Keys

- primary key: uniquely identifies a row in a table
- surrogate key: primary key that has no relation to the data
  - usually generated with a database sequence (1,2,3,4,5,6...)
  - or a GUID generator (5C37A7C133968DAFE040610A299461FB)

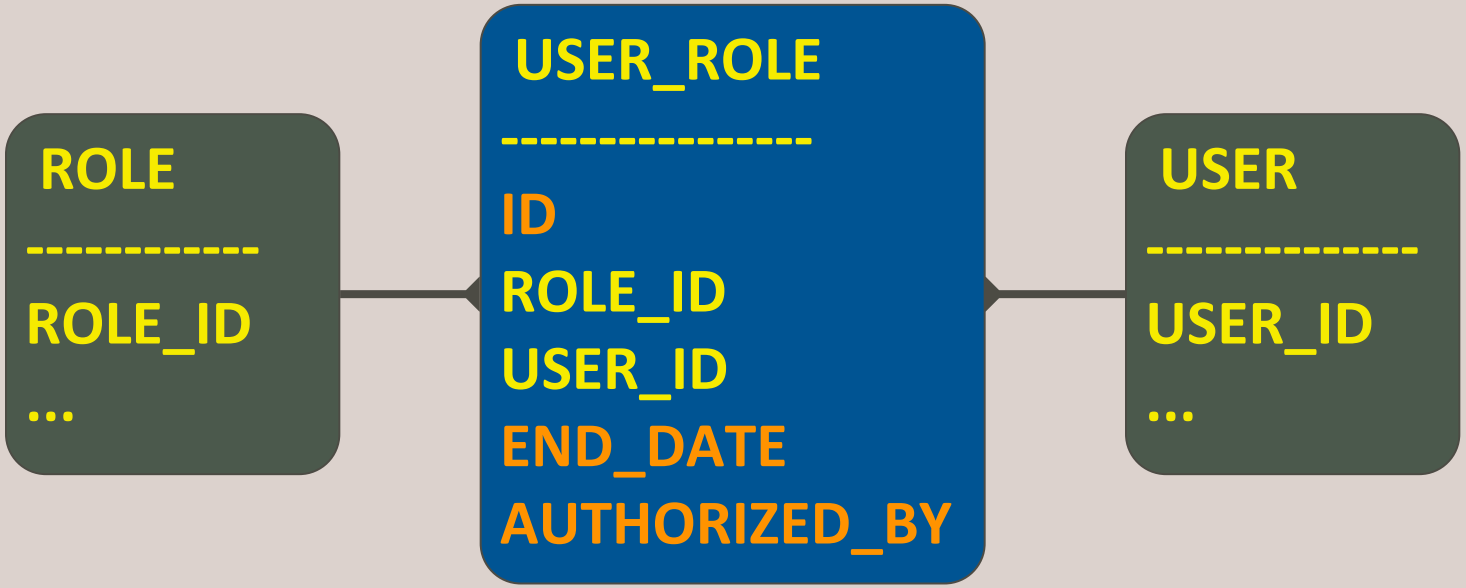
Primary Keys

- on pure bridge tables you can use a composite key consisting of the 2 foreign keys
- map with @ManyToMany



Primary Keys

- use a surrogate key when attributes of the relationship exist
- Map the bridge table explicitly and use @OneToMany from each direction



Sequences

- map sequences with @SequenceGenerator
- bind the sequence to a column with @GeneratedValue

```
...
@Id
@SequenceGenerator(name="UserSequence",
                   sequenceName="USER_PK",
                   allocationSize=1)
@GeneratedValue(strategy=GenerationType.SEQUENCE,
                generator="UserSequence")
private Long id;
...
```

Sequences

Don’t forget the allocation size

Sequences

- Sequence with classic generator
  - default allocation size is 50
  - query the database sequence (.nextval)
  - multiply the database sequence value by the allocation size
  - use that value to insert the row
  - increment by 1 internally until the block is exhausted
  - then return to the database sequence

Sequences

- example of classic generator using
  - allocation size: 50 (default)
  - database increment: 1 (default)
  - create sequence example_seq [increment by 1];

Database Sequence Returns	Hibernate Uses
1	50, 51, 52, 53... 99
2	100, 101, 102... 149
3	150, 151, 152... 199

Sequences

- Sequence with enhanced generator
  - optional in v3.2.3+ (stack 3.2 uses this)
  - default allocation size of 50
  - query the database sequence (.nextval)
  - subtract the allocation size from the database sequence value and add 1
  - use that value to insert the row
  - increment by 1 internally until the block is exhausted
  - then return to the database sequence

Sequences

- example of enhanced generator using
  - allocation size: 50 (default)
  - database increment: **50, start with 50**
  - create sequence example_seq increment by 50;

Database Sequence Returns	Hibernate Uses
50	1, 2, 3... 50
100	51, 52, 53... 100
150	101, 102, 103... 150

Sequences

- These behaviors (both classic and enhanced) boost performance by avoiding trips to the database
- This is good if...
  - you need the performance boost
  - holes in the id field are acceptable
  - all entry points into the table use the same algorithm... OTHERWISE...
    ORA-00001 unique constraint violated

Sequences

- use
  - explicit allocation size of 1
  - database sequences that increment by 1
- unless
  - you need the performance boost
  - all entry points use the same algorithm

https://tech.lds.org/wiki/Hibernate,_JPA,_and_Sequences

Orphans

- orphans occur when a parent record is missing
  - Cause: the parent record is not in the database
  - Why?
    - missing or un-enforced foreign keys
    - mapping views instead of base tables
- counter orphan tactics
  - um, use foreign keys, hello? why aren't you?
  - avoid mapping views
    - if you must, isolate them, don't map their relationships

Equals and Hashcode

- normally, most Java objects provide default equals() and hashCode() methods based on the object's identity
- they work great for objects that stay in memory, but hibernate marshals them in and out
- if you want to store entities in a List, Map or Set implement equals AND hashCode on the entity

Equals and Hashcode

- the identifier doesn’t work because the key doesn’t work until the object has been persisted*
- implement equals and hashCode using the business key works
- a workaround is to save and flush after creating a new object (performance? prone to forgetting)

http://www.hibernate.org/109.html

Fetch Types

- Eager vs. Lazy
- The default fetch type is often correct however..
- This is the first place you should look when tuning performance
- Watch your console for repeating queries
- that's an N+1 - hibernate is returning to the database for each record retrieved in a previous query

Fetch Types

- fix N+1 problems
- they are hardly noticed when developing against a local or near by database
- but the problem is amplified when latency between the application server and the database server is introduced - firewalls, distance, etc.

Fetch Types

- The opposite of N+1 is the “load up the world” problem
- Hibernate is eager loading too many relationships
- Stick with the default fetch type (Lazy) on *ToMany relationships, only switch to Eager when N+1’s are occuring

Concurrency

JPA supports both optimistic and pessimistic locking

Just an overview, see the docs for details

Concurrency

- Pessimistic locking
  - use with moderate or less contention
  - must be inside a transaction
  - prevents collision up front
  - **em.find(Example.class, exampleId, LockModeType.PESSIMISTIC_WRITE)**
  - translates into a “SELECT FOR UPDATE”

Concurrency

- Optimistic locking
  - use with any level of contention
  - must be inside of a transaction
  - use @Version annotation on the version column
  - JPA check the version before writing out changes
  - throws OptimisticLockException if the row was modified by another transaction since the last read
  - write your application to either automatically recover or allow the user to verify and re-try