let's move
the java world

Hibernate Puzzlers

Patrycja Wegrzynowicz

About Me

- 10+ years of professional experience as software developer, architect, and head of software R&D
- PhD in Computer Science
  - Patterns and anti-patterns, code analysis, language semantics, compiler design
- Speaker at JavaOne, Devoxx, OOPSLA, JavaZone, TheServerSide Symposium, Jazoon, others
- CTO of Yonita, Inc.
  - Bridge the gap between the industry and the academia
  - Automated detection and refactoring of software defects
  - Security, performance, concurrency, databases
- Twitter: @yonlabs

Today

- Five hibernate-related puzzles
  - Short program with curious behavior
  - Hibernate, JPA, general database issues
  - Correctness, performance
  - Question to you (multiple choice)
  - Mystery revealed
  - How to fix it
  - Lessons learned

Disclaimer

I do think Hibernate is a great tool!

#1: Volatile Warehouse

Volatile Warehouse

```
@Entity public class Warehouse {
        private Long id;

        private int maxCapacity;

        private int actualUtilization;


        @Id @GeneratedValue public Long getId() {...}

        protected void setId(Long id) {...}


        public int getMaxCapacity() { return this.maxCapacity; }

        protected void setMaxCapacity(int c) { this.maxCapacity = c; }


        public void setActualUtilization(int u) {

                        if (u > maxCapacity) throw new IllegalArgumentException("utilization exceeds capacity");

                        this.actualUtilization = u;

        }

}
// new EntityManager and new transaction
Warehouse warehouse = new Warehouse(20, 10); em.persist(warehouse);

// new EntityManager and new transaction
Warehouse found = em.find(Warehouse.class, warehouse.getId());
System.out.println(found.getMaxCapacity() + " " + found.getActualUtilization());
```

What Does It Print?

(a) 20 10
(b) 10 20
(c) Throws IllegalArgumentEx.
(d) Throws PersistentException
(e) None of the above

```
@Entity public class Warehouse {
        private Long id;

        private int maxCapacity;

        private int actualUtilization;


        @Id @GeneratedValue public Long getId() {...}

        protected void setId(Long id) {...}


        public int getMaxCapacity() { return this.maxCapacity; }

        protected void setMaxCapacity(int c) { this.maxCapacity = c; }


        public void setActualUtilization(int u) {

                        if (u > maxCapacity) throw new IllegalArgumentException("utilization exceeds capacity");

                        this.actualUtilization = u;

        }

}
// new EntityManager and new transaction
Warehouse warehouse = new Warehouse(20, 10); em.persist(warehouse);

// new EntityManager and new transaction
Warehouse found = em.find(Warehouse.class, warehouse.getId());
System.out.println(found.getMaxCapacity() + " " + found.getActualUtilization());
```

What Does It Print?

(a) 20 10

(b) 10 20

(c) Throws IllegalArgumentException

(d) Throws PersistentException

(e) None of the above

Hibernate sets the properties in the alphabetical order and maxCapacity is not initialized yet while setting actualUtilization.

IllegalArgumentException is wrapped by PersistentException.

Another Look

```
@Entity public class Warehouse {
        private Long id;

        private int maxCapacity;

        private int actualUtilization;


        @Id @GeneratedValue public Long getId() {...}

        protected void setId(Long id) {...}


        public int getMaxCapacity() { return this.maxCapacity; }

        protected void setMaxCapacity(int c) { this.maxCapacity = c; }     // called 2nd


        public void setActualUtilization(int u) {        // called 1st

                        if (u > maxCapacity) throw new IllegalArgumentException("utilization exceeds capacity");

                        this.actualUtilization = u;

        }

}
// new EntityManager and new transaction
Warehouse warehouse = new Warehouse(20, 10); em.persist(warehouse);

// new EntityManager and new transaction
Warehouse found = em.find(Warehouse.class, warehouse.getId());
System.out.println(found.getMaxCapacity() + " " + found.getActualUtilization());
```

We can make it working! ᴍ

```
@Entity
public class Warehouse {
        private Long id;

        private int maxCapacity;

        private int utilization;


        @Id @GeneratedValue

        public Long getId() {...}

        protected void setId() {...}


        public void setUtilization(int u) {
                        if (u > maxCapa                                  capacity”);
                        this.utilization =
        }
}
// new EntityManager and new transaction
Warehouse warehouse = new Warehouse(20, 10); em.persist(warehouse);

// new EntityManager and new transaction
Warehouse found = em.find(Warehouse.class, warehouse.getId());
System.out.println(found.getMaxCapacity() + “ “ + found.getUtilization());
```

It’s not a good fix!
Think about maintanability.

Better Fix

```
@Entity
public class Warehouse {
        @Id @GeneratedValue

        private Long id;

        private int maxCapacity;

        private int actualUtilization;


        public Long getId() {...}

        protected void setId() {...}


        public void setActualUtilization(int u) {

                        if (u > maxCapacity) throw new IllegalArgumentException("utilization exceeds capacity");

                        this.actualUtilization = u;

        }

}
// new EntityManager and new transaction
Warehouse warehouse = new Warehouse(20, 10); em.persist(warehouse);

// new EntityManager and new transaction
Warehouse found = em.find(Warehouse.class, warehouse.getId());
System.out.println(found.getMaxCapacity() + " " + found.getActualUtilization());
```

Lessons Learned

- Property access mapping and a rich domain model do not get along very well
  - JPA unspecified behavior
- Use field access mappings
  - Fields initialized without calling setters

#2: Where’s My Head?

Where's My Head?

```
@Entity public class Person {
        @Id @GeneratedValue private Long id;

        @Basic private String name;

        @Embedded private Head head = new Head();

        @Embedded private Hand left = new Hand(), right = new Hand():

}
@Embeddable public class Head {
        private String thought;

        public String toString() { return "head"; }

}
@Embeddable public class Hand {
        public String toString() { return "hand"; }

}
// new EntityManager and new transaction
Person patrycja = new Person("patrycja");
em.persist(patrycja);
// new EntityManager and new transaction
Person found = em.find(Person.class, patrycja.getId());
System.out.println(found.getHead()+" "+found.getLeft()+" "+found.getRight());
```

What Does It Print?

(a) head hand hand
(b) head null null
(c) null null null
(d) None of the above

```
@Entity public class Person {
        @Id @GeneratedValue private Long id;

        @Basic private String name;

        @Embedded private Head head = new Head();

        @Embedded private Hand left = new Hand(), right = new Hand():

}
@Embeddable public class Head {
        private String thought;

        public String toString() { return “head”; }

}
@Embeddable public class Hand {
        public String toString() { return “hand”; }

}
// new EntityManager and new transaction
Person patrycja = new Person(“patrycja”);
em.persist(patrycja);
// new EntityManager and new transaction
Person found = em.find(Person.class, patrycja.getId());
System.out.println(found.getHead()+“ “+found.getLeft()+” “+found.getRight());
```

What Does It Print?

(a) head hand hand

(b) head null null

(c) null null null

(d) None of the above

Hibernate does not distinguish between null embedded object and not-null embedded object with all fields null.

Database Dependency

- Some databases treat **empty strings** as nulls!
- Side effect: your not-null embedded objects with empty strings are retrieved as nulls!

Lessons Learned

- What you store is NOT always what you get!
- Test edge cases
- Test target environments

#3: Heads of Hydra

Heads of Hydra

```
@Entity
public class Hydra {
	private Long id;

	private List<Head> heads = new ArrayList<Head>();


	@Id @GeneratedValue

	public Long getId() {...}

	protected void setId() {...}

	@OneToMany(cascade=CascadeType.ALL)

	public List<Head> getHeads() {

			return Collections.unmodifiableList(heads);

	}

	protected void setHeads() {...}

}

// new EntityManager and new transaction: creates and persists the hydra with 3 heads

// new EntityManager and new transaction
Hydra found = em.find(Hydra.class, hydra.getId());
```

How Many Queries in 2nd Tx?

(a) 1 select
(b) 2 selects
(c) 1+3 selects
(d) 2 selects, 1 delete, 3 inserts
(e) None of the above

During commit hibernate checks whether the collection property is dirty (needs to be re-created) by comparing Java identities (object references).

IllegalArgumentException is wrapped by PersistentException.

Another Look

```
@Entity
public class Hydra {
	private Long id;

	private List<Head> heads = new ArrayList<Head>();


	@Id @GeneratedValue

	public Long getId() {...}

	protected void setId() {...}

	@OneToMany(cascade=CascadeType.ALL)

	public List<Head> getHeads() {

			return Collections.unmodifiableList(heads);

	}

	protected void setHeads() {...}

}
// new EntityManager and new transaction: creates and persists the hydra with 3 heads

// new EntityManager and new transaction
// during find only 1 select (hydra)
Hydra found = em.find(Hydra.class, hydra.getId());
// during commit 1 select (heads),1 delete (heads),3 inserts (heads)
```

Lessons Learned

- Expect unexpected ;-)
- Prefer field access mappings
- Operate on collection objects returned by hibernate
  - Don’t change collection references unless you know what you’re doing

#4: Plant a Tree

Plant a Tree

```
@Entity
public class Forest {
        @Id @GeneratedValue

        private Long id;

        @OneToMany

        private Collection<Tree> rees = new HashSet<Tree>();


        public void plantTree(Tree tree) {

                        return trees.add(tree);

        }

}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree(“oak”);
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

How Many Queries in 2nd Tx?

(a) 1 select, 2 inserts
(b) 2 selects, 2 inserts
(c) 2 selects, 1 delete, 10.000+2 inserts
(d) 2 selects, 10.000 deletes, 10.000+2 inserts
(e) Even more ;-)

```
@Entity
public class Forest {
        @Id @GeneratedValue

        private Long id;

        @OneToMany

        private Collection<Tree> trees = new HashSet<Tree>();


        public void plantTree(Tree tree) {

                        return trees.add(tree);

        }

}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree("oak");
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

How Many Queries in 2nd Tx?

(a) 1 select, 2 inserts

(b) 2 selects, 2 inserts

(c) 2 selects, 1 delete, 10.000+2 inserts

(d) 2 selects, 10.000 deletes, 10.000+2 inserts

(e) Even more ;-)

The combination of **OneToMany** and **Collection** enables a bag semantic. That's why the collection is re-created.

Plant a Tree Revisited

```
@Entity
public class Orchard {
        @Id @GeneratedValue

        private Long id;

        @OneToMany

        private List<Tree> trees = new ArrayList<Tree>();


        public void plantTree(Tree tree) {

                        return trees.add(tree);

        }

}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree("apple tree");
em.persist(tree);
Orchard orchard = em.find(Orchard.class, id);
orchard.plantTree(tree);
```

STILL BAG SEMANTIC

Use OrderColumn or IndexColumn for list semantic.

OneToMany Mapping

Semantic	Java Type	Annotation
Bag semantic	java.util.Collection java.util.List	@ElementCollection || @OneToMany || @ManyToMany
Set semantic	java.util.Set	@ElementCollection || @OneToMany || @ManyToMany
List semantic	java.util.List	(@ElementCollection || @OneToMany || @ManyToMany) && (@OrderColumn || @IndexColumn)

OneToMany Mapping

Semantic	Add element	Remove element	Update element
Bag semant	re-create:	re-create:	date
Set semanti			date
List semantic	1 insert + M updates	1 delete + M updates*	1 update

Ooops, we have a problem.
(list: removal of nth element, n < size-2)

```
Hibernate: update Forest_Tree set trees_id=? where Forest_id=? and trees_ORDER=?
2832 [main] WARN  org.hibernate.util.JDBCExceptionReporter  - SQL Error: 1062, SQLState: 23000
2832 [main] ERROR org.hibernate.util.JDBCExceptionReporter  - Duplicate entry '10' for key 'trees_id'
Exception in thread "main" javax.persistence.RollbackException: Error while committing the transaction
    at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:93)
    at com.yonita.examples.jpapuzzles.JPA.execute(JPA.java:23)
    at com.yonita.examples.jpapuzzles.puzzle5.collection.Forest.main(Forest.java:68) <5 internal calls>
Caused by: javax.persistence.PersistenceException: org.hibernate.exception.ConstraintViolationException
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1387)
    at org.hibernate.ejb.AbstractEntityManagerImpl.convert(AbstractEntityManagerImpl.java:1315)
    at org.hibernate.ejb.TransactionImpl.commit(TransactionImpl.java:81)
    ... 7 more
Caused by: org.hibernate.exception.ConstraintViolationException: Could not execute JDBC batch update
```

@OneToMany (no cascade option)

GeeCON

Plant a Tree

```
@Entity
public class Forest {
        @Id @GeneratedValue

        private Long id;

        @OneToMany

        private Set<Tree> trees = new HashSet<Tree>();


        public void plantTree(Tree tree) {

                        return trees.add(tree);

        }

}

// new EntityManager and new transaction: creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
Tree tree = new Tree("oak");
em.persist(tree);
Forest forest = em.find(Forest.class, id);
forest.plantTree(tree);
```

1. Collection elements loaded into memory
2. Possibly unnecessary queries
3. Transaction and locking schema problems: version, optimistic locking

Plant a Tree

```
@Entity public class Forest {
        @Id @GeneratedValue

        private Long id;

        @OneToMany(mappedBy = „forest”)

        private Set<Tree> trees = new HashSet<Tree>();


        public void plantTree(Tree tree) {

                        return trees.add(tree);

        }

}

@Entity public class Tree {
                @Id @GeneratedValue

        private Long id;

        private String name;

        @ManyToOne

        private Forest forest;


        public void  setForest(Forest forest) {

                        this.forest  = forest;

                        Forest.plantTree(this);

        }

}
```

Set semantic on the inverse side forces of loading all trees.

Plant a Tree

```
@Entity public class Forest {
      @Id @GeneratedValue

      private Long id;

      @OneToMany(mappedBy = „forest”)

      private Collection<Tree> trees;


      public void plantTree(Tree tree) {

                  return trees.add(tree);

      }

}

@Entity public class Tree {
            @Id @GeneratedValue

      private Long id;

      private String name;

      @ManyToOne

      private Forest forest;


      public void  setForest(Forest forest) {

                  this.forest  = forest;

                  Forest.plantTree(this);

      }

}
```

Plant a Tree

```
@Entity public class Forest {
    @Id @GeneratedValue

    private Long id;

    @OneToMany(mappedBy = „forest”)

    private Collection<Tree> trees;


}
```

```
    ... 7 more
Caused by: java.sql.BatchUpdateException: Cannot delete or update a parent row: a foreign key constraint fails (`jpapuzzles`.`tree`
    at com.mysql.jdbc.PreparedStatement.executeBatchSerially(PreparedStatement.java:2020)
    at com.mysql.jdbc.PreparedStatement.executeBatch(PreparedStatement.java:1451)
    at org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:70)
    at org.hibernate.jdbc.AbstractBatcher.executeBatch(AbstractBatcher.java:268)
```

```
@E
    private String name;

    @ManyToOne

    private Forest forest;

}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
em.remove(forest);
```

Plant a Tree

```
@Entity public class Forest {
        @Id @GeneratedValue

        private Long id;

        @OneToMany(mappedBy = „forest”)

        private Collection<Tree> trees;


}

@Entity public class Tree {
                        @Id @GeneratedValue

        private Long id;

        private String name;

        @ManyToOne

        private Forest forest;

}

// creates and persists a forest with 10.000 trees

// new EntityManager and new transaction
for (Tree tree : forest.getTrees()) { tree.setForest(null); }
em.remove(forest);
```

10.000 UPDATES
1 DELETE

Lessons Learned

- Big data cause big problems
  - Standard mappings don’t handle large datasets well
  - Smart model, bulk processing, projections
- Analyze your use cases and usage patterns to adjust your collection types properly

#5: Fashionable Developer

Fashionable Developer

```
@Entity public class Developer {
	@Id @GeneratedValue

	private Long id;

	private String mainTechnology;

	public boolean likesMainTechnology() {

			return „Java".equalsIgnoreCase(mainTechnology);

	}

}
// creates and persists a developer that uses hibernate as mainTechnology
// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{"Ceylon", „Kotlin", "Scala"}) {
	dev.setMainTechnology(tech);

	// othersAreUsingIt:

	// select count(*) from Developer where mainTechnology = ? and id != ?

	if (othersAreUsingIt(tech, dev) && dev.likesMainTechnology()) {

			foundCoolStuff = true; break;

	}

}
if (!foundCoolStuff) {
	// still use plain old Java

	dev.setMainTechnology("java");

}
```

Fashionable Developer

```
@Entity public class Developer {
	@Id @GeneratedValue

	private Long id;

	private String mainTechnology;

	public boolean likesMainTechnology() {

			return "hibernate".equalsIgnoreCase(mainTechnology);

	}

}
// creates and persists a developer that uses hibernate as mainTechnology
// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{"HTML5", "Android", "Scala"}) {
	dev.setMainTechnology(tech);

	// othersAreUsingIt:

	// select count(*) from Developer where mainTechnology = ? and id != ?

	if (othersAreUsingIt(tech, dev) && dev.likesMainTechnology()) {

			foundCoolStuff = true; break;

	}

}
if (!foundCoolStuff) {
	// still use hibernate

	dev.setMainTechnology("hibernate");

}
```

(a) 2 select
(b) 4 selects
(c) 4 selects, 1 update
(d) 4 selects, 4 updates
(e) None of the above

How Many Queries in 2nd Tx?

(a) 2 selects

(b) 4 selects

(c) 4 selects, 1 update

(d) 4 selects, 4 inserts

(e) None of the above

Hibernate must guarantee correctness of executed queries, therefore in certain situations it must perform flushes during a transaction.

Fashionable Developer

```
@Entity public class Developer {
	@Id @GeneratedValue

	private Long id;

	private String mainTechnology;

	public boolean likesMainTechnology() {

			return „Java".equalsIgnoreCase(mainTechnology);

	}

}
// creates and persists a developer that uses hibernate as mainTechnology
// new EntityManager and new transaction
Developer dev = em.find(Developer.class, id);
boolean foundCoolStuff = false;
for (String tech : new String[]{„Ceylon", „Kotlin", "Scala"}) {
	dev.setMainTechnology(tech);

	// othersAreUsingIt:

	// select count(*) from Developer where mainTechnology = ? and id != ?

	if (/*othersAreUsingIt(tech, dev) && */dev.likesMainTechnology()) {

			foundCoolStuff = true; break;

	}

}
if (!foundCoolStuff) {
	// still use hibernate

	dev.setMainTechnology(„Java");

}
```

NO UPDATES

Lessons Learned

- Flushes can be performed during the transaction, not only at the commit time
- Temporary changes persisted into the database unnecessarily but can also destroy the semantic of your app

Conclusion

- Hibernate is reasonably simple and elegant
- But it has several sharp corners
- Pay more attention to:
- Actual queries executed
- Collections
- Transaction management
- Consider Yonita to detect such issues :-)

Contact

- email: patrycja@yonita.com
- twitter: @yonlabs
- http://www.yonita.com