

## Theory Questions

### 1.6.1. Annotation vs. XML Declarations

In the previous tasks you already gained some experiences using annotations and XML. What are the benefits and drawbacks of each approach? In what situations would you use which one? Hint: Think about maintainability and the different roles usually involved in software projects.

#### Solution

Advantages:

- Separation of relationships and all logic
- More readable for complex definitions

Disadvantages:

- A lot less resources online
- More verbose APIs for often quite simple definitions
- Two separate files to define a Table, which do need to be changed in lockstep to make sense (less maintainable)
- Less obvious where the relationships are defined (you need to read the pom.xml config)

I would almost always opt for the annotations, except for in projects where there is a DB Architect. But even then I as a developer might still write the files and the expert might only define the schema. Often I might start out with annotations and later change to XML if the relations become too complex to be readable in Java Annotations.

### 1.6.2. Entity Manager and Entity Lifecycle

What is the lifecycle of a JPA entity, i.e., what are the different states an entity can be in? What EntityManager operations change the state of an entity? How and when are changes to entities propagated to the database?

#### Solution

- 1) New: A new entity is created and it is not yet associated to the DB in any way.
- 2) Managed: The new entity was submitted to the DB (with something like `EntityManager.persist()`). Any changes that are now made to the entity will be stored and sent to the DB.
- 3) Detached: The entity is now no longer connected to the DB, so all changes made to it won't be communicated to the DB (with something like `EntityManager.detach()`)
- 4) Removed: The item was removed from the DB (with something like `EntityManager.remove()`)

### 1.6.3. Optimistic vs. Pessimistic Locking

The database systems you have used in this assignment provide different types of concurrency control mechanisms. Redis, for example, provides the concept of optimistic locks. The JPA EntityManager allows one to set a pessimistic read/write lock on individual objects. What are the main differences between these locking mechanisms? In what situations or use cases would you employ them? Think of problems that can arise when using the wrong locking mechanism for these use cases.

#### Solution

**Pessimistic Locking** requires the user to set and hold a lock while they are reading and modifying data on an item.

**Optimistic Locking** doesn't provide explicit locks. A user simply reads, and modifies an item. The modification however can fail if in the item was changed by another actor since the reading. In which case the whole transaction fails and has to be tried again.

Pessimistic Locking can lead to poor performance, but might be preferred in scenarios which many overlapping locks which is a worst-case scenario for optimistic locking.

### 1.6.4. Database Scalability

How can we address system growth, i.e., increased data volume and query operations, in databases? Hint: vertical vs. horizontal scaling. What methods in particular do MongoDB and Redis provide to support scalability?

#### Solution

Like most services, databases can scale vertically. However there is a limit at which point, throwing more storage, cpu, ram won't increase performance.

Luckily, many databases also support horizontal scaling (not all, especially not in memory databases like SQLite or H2). MongoDB and Redis have two horizontal scaling approaches, replication and partitioning.

**Replication** can help with many (read) query operations as the data is stored on many replicants which all can process read queries. However, it does not help with many write queries or increased data volume.

**Partitioning** also known as Sharding, splits the data over multiple servers, so that each server only needs to hold a subset. This helps with increased data volume and write queries. But it might happen that a complex read query needs to be run on all partitions and can therefore increase CPU utilization.

MongoDB also supports sharding with replication which can address all problems by combining both solutions.