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:

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

Disadvantages:

- A lot less ressources online
- More verbose APIs for often quite simple definitions
- Two seperate files to define a Table, which do need to be changed in lockstep to make sense (less maintable)
- 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 than 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 to 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 send 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

Pesemistic Locking requires the user to set a 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.

Pesemistic Locking can lead to poor performance, but might be prefered 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 databasses 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 proccess 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 witg replication which can address all problems by combining both solutions.

2.4.1. Java Transaction API

Consider the match method and how you would handle things different if you were to implement it with pessimistic or optimistic locking.

Solution

With optimistic locks the whole match would have been reexecuted with on conflicts. While I implemented it with optimistic locks we should have implemented it with pesimistcs to improve performance.

2.4.2. Remoting Technologies

Compare gRPC remoting and Web services. When would you use one technology, and when the other? Is one of them strictly superior? How do these technologies relate to other remoting technologies that you might know from other lectures (e.g., Java RMI, or even socket programming)?

Solution

gRPC seams a lot simpler, as it is clearly designed for calling functions on another machine. With Webservices you need to check, HTTP-Status codes, should verify the content header, the service itself should check your content accept header.

For services that need to communicate, I would prefer gRPC simply because of their simplicity. Compared to Java RMI it also has the big advantage that it isn't limiting to a single language (well, to the JVM).

For public facing APIs I would prefer Web Services (REST-ish), because even though it is not as simple, I know that the consumers can choose almost any programming language since they all can perform requests and can parse JSON.

If it is avoidable, I will avoid socket programming. From experience, it forces you to think about many low level aspects that are often irrelevant to your application. Maybe if latency is a extremely critical aspect of the application and it is currently bottlenecked by the network.

2.4.3. Class Loading

Explain the concept of class loading in Java. What different types of class loaders exist and how do they relate to each other? How is a class identified in this process? What are the reasons for developers to write their own class loaders?

Solution

Class loading is the process of loading compiled Java classes into the JVM.

There are three class loaders that come with Java:

1) Bootstrap Class Loader: Loads internal classes like `java.lang*`.
2) Extension Class Loader: Loads the JDK extensions.
3) System Class Loader: Loads classes from the current classpath.

Java class names are preserved in the *.jar and there fore the classloader can identify all classes in the jar.

There are multiple reasons why one might want to implement their own class loader:

- Modify existing classes (like we did with javaassist)
- Creating classes dynamically
- Loading different versions of the same class, depending on its use-case.

2.4.4. Weaving Times in AspectJ

What happens during weaving in AOP? At what times can weaving happen in AspectJ? Think about advantages and disadvantages of different weaving times.

Solution

Weaving is when the defined aspects get ”woven” into their target classes. There are three ways this can happen, during compile time, post-comile time and load time.

Compile time weaving is the first choice if the source code for the target is given and the aspect is known at compile time. Because the AspectJ compiler can rewrite the target classes here, this leads to the best performance.

Post compile time weaving also runs during compilation but here the source code can be unknown and it is possible to weave directly into the java bytecode. This can be handy for libraries, when only a jar is provided.

Load time weaving happens during runtime and can therefore modify already running programs which can be utialized for plugins.