LazyInitializationException

Hibernate throws a *LazyInitializationException* if you try to access a not initialized relationship to another entity without an active session.

The best way to fix it is to initialize the required relationship with a @NamedEntityGraph.

```
@NamedEntityGraph(name = "graph.AuthorBooks",
                  attributeNodes = @NamedAttributeNode("books"))
```

You can then provide the graph as a hint to define which relationships shall be initialized with a given query.

```
EntityGraph<?> graph =
                        em.getEntityGraph("graph.AuthorBooks");
HashMap<String, Object> properties = new HashMap<>();
properties.put("javax.persistence.fetchgraph", graph);

Author a = em.find(Author.class, 1L, properties);
```

OptimisticLockException

Hibernate throws an OptimisticLockException when you use optimistic locking and it detects a conflicting update of an entity. That most often happens for one of two reasons:

1. 2 users try to update the same entity at nearly the same point in time.

   You can't do much to avoid this without introducing pessimistic locking which would sacrifice the performance of your application. Just try to update the entity representations in the client as often as possible and to keep the update operations as short as possible.

2. 1 user performs 2 updates of the same entity, and you didn't refresh the entity representation in the client so that the version value wasn't updated after the first update.

   This one is a bug. You need to make sure that your client always updates its representation of the entity after the user triggered any change on the entity. And your client application should also not cache the entity or any value object representing it.

org.hibernate.AnnotationException: Unknown Id.generator

This one is caused by a missing @SequenceGenerator annotation which allows you to provide additional information about the sequence Hibernate shall use.

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
                        generator = "authorSequence")
@SequenceGenerator(name = "authorSequence",
                                sequenceName = "author_seq",
                                initialValue = 1000)
@Column(name = "id", updatable = false, nullable = false)
private Long id;
```

QuerySyntaxException: Table is not mapped

The most common reason for this Exception is that the default table name for an entity doesn't match the database table name. You can fix it with a *@Table* annotation and specify the table and schema name.

```
@Entity
@Table(name = "author", schema = "bookstore")
public class Author implements Serializable {

        …
}
```

org.hibernate.PersistentObjectException: detached entity passed to persist

This Exception can have multiple reasons and all of them are bugs:

1. You try to persist a new entity and provide a primary key value, but the entity mapping defines a strategy to generate it.

   This one is easy to fix, don't provide a primary key value or remove the primary key generation strategy ;)

2. You try to persist a new entity and the persistence context already contains an entity with the given id.

   This one should only occur if you manage the primary key values yourself, and your algorithm creates duplicates. My preferred approach to fixing this issue is to let Hibernate use a database sequence to generate the primary key values instead of implementing my own algorithm. But that's not always possible and in these cases, you have to test and debug the algorithm you use to generate the primary key values.

3. You try to persist a detached entity instead of merging it.

   And this one often happens, when you use entities in your client, and the client calls the wrong server method which persists new entities instead of updating the existing ones. The obvious way to fix this error is to fix the call in the client.

4. But there are also a few things you can do on the server side to avoid these kinds of issues, like using specific value objects for create use cases and not handling create and update use cases in the same server method. This makes it easier for the client developer to find and call the right method and to avoid these kinds of issues.