**📄 SWAM Group Project Report**

**Software Architecture and Methodologies – Spring 2025  
Group members: Valdemar Børresen, Anne Line Hval, Jonas Bakke**

**1. Introduction**

The goal of this project is to explore the impact of different JPA mapping choices on performance and database behavior.  
We implement a domain model with Users, StoreOrders, and Items, utilizing JPA, CDI, and an H2 in-memory database.  
Our focus is to experiment with:

* Embedded types (@Embeddable, @Embedded)
* Collections and associations (@OneToMany, @ManyToOne, @ElementCollection)
* Lazy vs Eager fetching
* Cascading operations
* Inheritance mappings (planned for possible extension)

The application is written in Java 17, structured as a Maven project, and follows best practices for clean layering (Entities → DAOs → Services → Presentation/Testing).

**2. Architecture Overview**

Key technologies:

* Jakarta Persistence API (JPA)
* Hibernate ORM (as JPA implementation)
* Jakarta CDI (Contexts and Dependency Injection)
* H2 Database (memory mode for easy testing)
* JUnit 5 (for unit and integration testing)

Layered structure:

* Model layer: JPA entities (User, StoreOrder, Item, Address)
* DAO layer: Generic BaseDAO and concrete DAOs
* Service layer: UserService, OrderService
* Infrastructure: CDI producers for EntityManager and EntityManagerFactory
* Testing/Presentation: Main class for running experiments and benchmarks

Dependency Injection is used across the DAO and Service layers to decouple components.

**3. Domain Model**

Entities:

* User
  + id, name
  + Embedded Address
  + One-to-many relationship with StoreOrder
* StoreOrder
  + id, orderDetails
  + Many-to-one relationship to User and Item
* Item
  + id, name, price
* Address
  + city, street, zipcode (embedded in User)

Mapping Strategies:

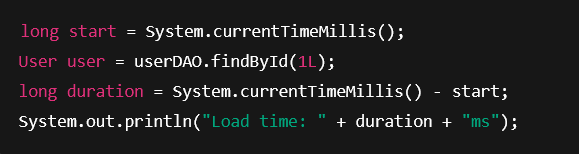
* @Embeddable and @Embedded for value objects (Address)
* @OneToMany and @ManyToOne for entity relationships
* Default Lazy fetching, configurable to Eager for experiments

**4. Experiments and Benchmarking**

We created experiments to compare LAZY and EAGER fetching:

Benchmark Method:

* Use System.currentTimeMillis() before and after EntityManager operations
* Example:



Observations:

* LAZY fetching performs better when only primary entity data is needed.
* EAGER fetching leads to larger queries and more memory use when related entities are large.

**5. Testing**

* Created JUnit 5 test cases for UserDAO and StoreOrderDAO.
* Verified persistence lifecycle: save, update, delete.
* Verified that cascading works for User → StoreOrder.
* Verified that @Embedded Address correctly persists and retrieves.

**6. Conclusion**

The project demonstrates practical application of JPA and CDI patterns:

* Clean layering and separation of concerns.
* Effective use of dependency injection for flexibility and testability.
* Awareness of performance trade-offs with Lazy vs Eager fetching.

By conducting benchmarks and observing database queries, we validated the impact of JPA annotations and ORM behavior on system performance**.**

**7. Source Code**

The complete project is available at:  
[GitHub repository link her]

**8. Future Work (optional)**

Potential improvements could include:

* Adding inheritance hierarchies (@Inheritance) for the Item entity.
* Using JPQL queries for more complex data retrieval.
* Extending the benchmarks to measure memory and CPU usage.