**📄 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:

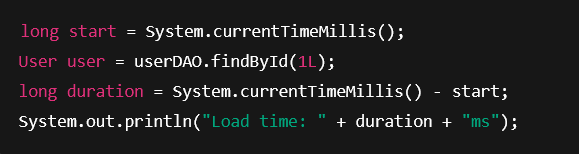
* @Embeddable and @Embedded for value objects (Address)
* @OneToMany and @ManyToOne for entity relationships
* Default Lazy fetching, configurable to Eager for experiments

**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.

**SWAM Project Report**

**Course:** Software Architecture and Methodologies  
**Term:** Spring 2025  
**Group members:**

* Valdemar Børresen
* Rikke [Etternavn]
* Anne Line [Etternavn]

**1. Introduction**

In this project, we aim to explore and analyze various mapping techniques in JPA (Jakarta Persistence API), including **basic mappings, embedded types, inheritance, collections, associations,** and the impact of **lazy vs eager loading strategies**. Additionally, we evaluate how these mappings affect the **structure, navigability, and performance** of the application.

To facilitate the experimentation, we developed a layered application using **Java 17**, **Maven**, **Hibernate (as JPA provider)**, and **H2** as an in-memory database. We also used **CDI (Contexts and Dependency Injection)** to manage dependencies between DAOs, services, and infrastructure components.

Our project focuses primarily on:

* Designing a meaningful domain model with associations and embedded types.
* Using JPA annotations to explore different mapping strategies.
* Conducting runtime experiments to compare performance and behavior under different configurations.
* Documenting the outcomes of these experiments with explanations and analysis.

**2. Technologies Used**

* **Programming language:** Java 17
* **Build tool:** Apache Maven
* **Persistence:** Jakarta Persistence API (JPA)
* **ORM:** Hibernate
* **Dependency Injection:** Jakarta CDI
* **Database:** H2 (in-memory)
* **Testing framework:** JUnit 5
* **IDE:** Visual Studio Code

**3. Domain Model and Mapping Strategies**

**3.1 Entities**

We modeled the following core entities:

* **User**
  + Fields: id, name
  + Embedded: Address (city, street, zipcode)
  + One-to-many: List of StoreOrders
* **StoreOrder**
  + Fields: id, orderDetails
  + Many-to-one: User (as buyer)
  + Many-to-one: Item
* **Item**
  + Fields: id, name, price
* **Address**
  + Value object with no identity
  + Embedded in User using @Embeddable and @Embedded

**3.2 Annotations Used**

* @Entity, @Id, @GeneratedValue
* @Embeddable, @Embedded
* @OneToMany, @ManyToOne, @JoinColumn
* @Table and @Column to avoid name conflicts with reserved SQL keywords
* @RequestScoped, @ApplicationScoped, and @Produces for CDI
* @ElementCollection (tested separately)

**4. CDI and Layered Architecture**

Our architecture follows a **clean separation of concerns** with four layers:

1. **Model layer** – JPA entities
2. **DAO layer** – Data access objects extending BaseDAO<T>
3. **Service layer** – Business logic components using DAOs
4. **Infrastructure** – EntityManagerProducer providing CDI-managed persistence context

All injections are handled with @Inject, and EntityManager instances are scoped to the request using @RequestScoped.

**5. Experiments and Benchmarking**

**5.1 Experiment Goals**

We wanted to test and compare the effects of:

* **FetchType.LAZY** vs **FetchType.EAGER** on performance and SQL behavior
* **Direct annotation-based mapping** vs **JPQL with JOIN FETCH**
* The behavior of Hibernate when managing associations and caching

**5.2 Experiment Setup**

* Created 100 users with 2 orders each
* Used System.currentTimeMillis() for timing
* Cleared the EntityManager after creation to force fresh loading
* Activated SQL logging via hibernate.show\_sql=true

**5.3 Code Sample**