



JPA with Hibernate

▲ Raoul Van den Berge

IT Consultant

raoul.vandenberge@infosupport.com

Graduated from KdG in 2020



Content

- **Spring Data JPA and Hibernate**
 - What am I actually using? Who has which responsibility?
- **Transaction management**
 - How does Spring manage transactions and how does it map to JPA/Hibernate?
- **Loading associations: best practices**
 - What is the best way to fetch associations?
- **Schema generation and validation**
 - How can I evolve my database?
- **JPQL**
 - How can I use JPQL to fix the *N+1 problem*? What can I do against the *cartesian product problem*?
- **Performance tips and common mistakes**
- **Further resources**



Spring Data JPA and Hibernate

What am I actually using?

Who has which responsibility?

▲ What is what?

- JPA
 - Jakarta Persistence API
 - Formerly known as Java Persistence API
 - Specification and interfaces.
- Hibernate
 - Implementation of the JPA API.
- Spring Data JPA
 - Layer on top of JPA which makes interacting with JPA less cumbersome.
 - Uses Hibernate behind the scenes.

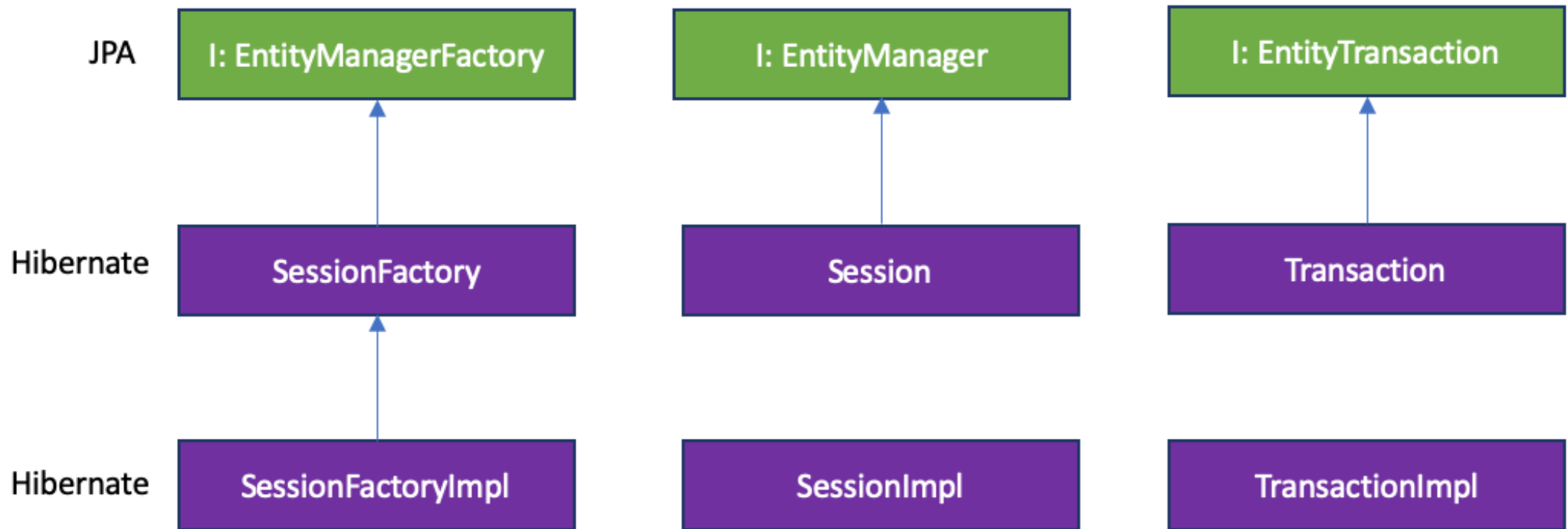


Transaction management

How does Spring manage transactions and how does it map to JPA/Hibernate?

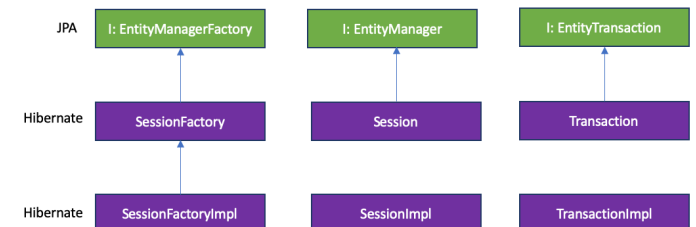


EntityManager vs Session



▲ Persistence context

- The persistence context tracks the entities in memory (the first level cache)
- In JPA: `EntityManager`
- In Hibernate: `Session`
- A database connection is bound to a `Session`.
- A `Session` can span multiple transactions.
 - Open Session In View (a `Session` per request)
- A `Session` can span a single business use case/transaction.
 - “Transaction Scoped Persistence Context”
 - A new `Session/EntityManager` per transaction.
 - `@Transactional`



Who does what?

What?	Project
<code>EntityManager</code>	JPA
<code>Session</code>	Hibernate
<code>CrudRepository</code>	Spring Data JPA
<code>@Transactional</code>	Spring Transaction
JPQL	JPA
JPQL/HQL (implementation)	Hibernate

▲ Accessing the persistence context

```
@PersistenceContext  
private EntityManager em;
```

If you are using Spring Data JPA, you don't need to use `EntityManager` directly (use a repository).

Transaction management in Spring

- JPA/Hibernate doesn't provide any type of declarative transaction management.
- Spring offers an API-neutral transaction platform.
 - Support for plain JDBC, JPA, etc.
 - `@Transactional`, `TransactionTemplate`, `TransactionManager`
- `TransactionManager` **manages transactions (and database connections)** and binds them to the current thread.



Loading associations: best practices

What is the best way to fetch associations?





▲ Demo setup

- Maven, Spring 3, Hibernate 6
- Primarily using tests
- Sometimes a little web...

▲ Problem

- Given:
 - A `Post` that can have multiple `Comments`.
- We want to:
 - Generate a summary of that `Post` with all its `Comments`.
- Problem:
 - How do we efficiently retrieve all `Comments` on a `Post`?



▲ Solution 1: Use lazy loading with

`FetchType.LAZY`

- Loads the association lazily when the getter on the entity is used.



2023-04-14 09:44:14.631 WARN 12456 --- [main] JpaBaseConfiguration\$JpaWebConfiguration :
spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during
view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning

▲ Open Session in View?!

- `findById` only keeps the database session open when retrieving the initial entity (the post).
- Open Session in View ensures that a database session is active **throughout the entire web request** – so that we can retrieve the lazy loaded collections in the same database session (the post comments).
- By default “on” in Spring Boot (`spring.jpa.open-in-view=true`)



▲ Advice

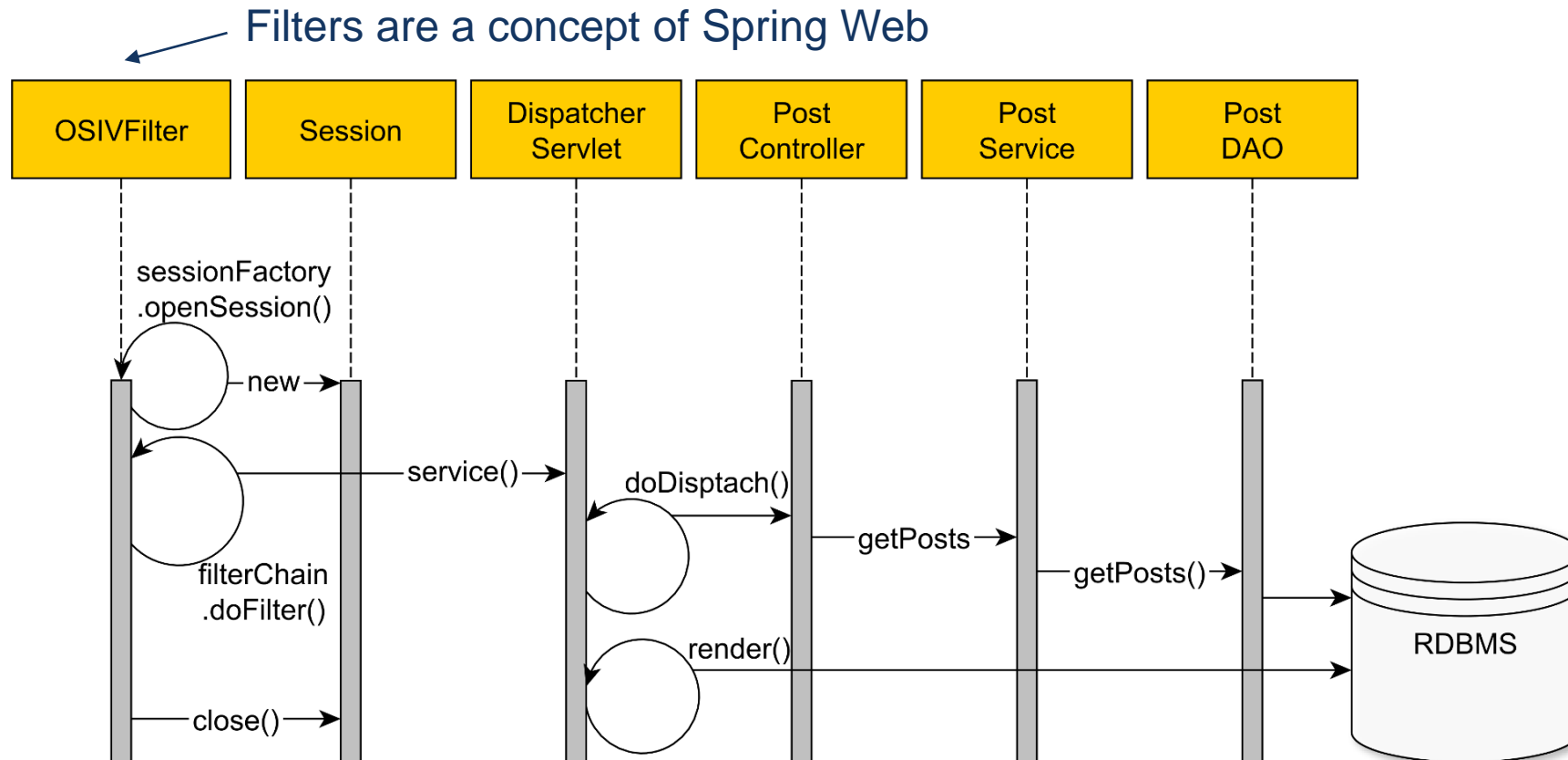
- Avoid Open Session in View, especially if you're not familiar with JPA.
- There is no separation of concerns since SQL statements can be generated at any point in the application (like the UI rendering process).
- Hard to get rid of in badly tested projects.
- It's easy to navigate associations at any point, which might cause performance issues later on.
- Database connection is held throughout the entire request, which increases connection lease times.

▲ Problem: `LazyInitializationException`

- A lazy association needs the `Session` to be opened in order to initialize the collection.
- Open Session in View kept a session open for us the entire request...
- The persistence context (`Session`) is closed after executing a method on a `JpaRepository`.
- If the persistence context is closed, when trying to access a non-initialized lazy association, the infamous `LazyInitializationException` is thrown.



Open Session in View (OSIV)



DEMO

▲ Solution 2: Use eager loading with `FetchType.EAGER`

▲ I was getting the same error for a one to many relationships for below annotation.

23

```
@OneToMany(mappedBy="department", cascade = CascadeType.ALL)
```

▼

🔄 Changed as below after adding `fetch=FetchType.EAGER`, it worked for me.

```
@OneToMany(mappedBy="department", cascade = CascadeType.ALL, fetch=FetchType.EAGER)
```

Share Improve this answer Follow



[Ahmed Ashour](#)
4,418 ● 10 ● 32 ● 49



[Smruti R Tripathy](#)
733 ● 1 ● 9 ● 15

edited Nov 30, 2017 at 18:18 answered Sep 20, 2017 at 10:21

44 Yes it may fix it but now you are loading the whole tree of data. This will have negative performance impacts in most cases – [astro8891](#) Jun 11, 2018 at 1:39

solved my problem thank you so much – [crispengari](#) Dec 18, 2021 at 23:01

Add a comment

DEMO

▲ Lazy loading vs eager loading

- `FetchType.EAGER` is a code smell.
- Most often it's used for simplicity sake without considering the long-term performance penalties.
- The fetching strategy should never be the entity mapping responsibility.
- **Once a relationship is set to be eagerly fetched, it cannot be changed to being fetched lazily on a per-query basis.**
- **Each business use case has different entity load requirements and therefore the fetching strategy should be delegated to each individual query.**



▲ Advice

- Use lazy associations.
- Using lazy associations gives you the flexibility of changing the fetching strategy at query time with the `FETCH HQL` directive.

Advice

ASSOCIATION TYPE	DEFAULT FETCHING POLICY
@OneToMany	LAZY
@ManyToMany	LAZY
@ManyToOne	EAGER
@OneToOne	EAGER

Good practice: always set the fetching policy explicitly in your entity mapping.



▲ Solution 3: use `@Transactional`

- Ensures that a transaction is active within a scope using aspect-oriented-programming (AOP).
- At the location where you initialize a collection, ensure that it is wrapped in `@Transactional`.





▲ Where to use @Transactional

- The service layer determines the transaction boundaries.
- Avoid it in the web layer: it increases database connection lease times (see OSIV)
- Repositories require a transaction, but this should propagate from the service layer.

▲ When to use `@Transactional`

- When using repositories, `@Transactional` is applied to repository scope automatically.
- You'd lose the session after getting a result back from the repository.
- You'd get a `LazyInitializationException` when using Lazy Loading out of repository scope because the session is lost.
- You'll have to use it when using Lazy Loading.
- Good practice: always use it to clearly define transaction boundaries!

▲ Solution 4: custom query with JPQL JOIN FETCH

- Retrieves an entity with the flexibility of choosing the fetching strategy for an association.



▲ JOIN FETCH with projections

- Allows you to only select (“project”) the columns that you want into a Domain Transfer Object (DTO)
 - You should fetch just as much data you need to fulfill the requirements of a given use case.
 - Fetching too many columns than necessary has an impact.
- Often used for read-only use-cases (no entity)





Schema generation and validation

How can I evolve my database?

Schema generation

- Let Hibernate update the schema?
- Don't do this in production!

4. Schema generation

Hibernate allows you to generate the database from the entity mappings.



Although the automatic schema generation is very useful for testing and prototyping purposes, in a production environment, it's much more flexible to manage the schema using incremental migration scripts.



▲ Schema generation

- Write your own “schema migration” scripts.
- The scripts will reside in version control along with your codebase.
- When you check out a branch, you can recreate the whole schema from scratch (and so can your tests).
- Flexibility of writing your own migration logic.





▲ Schema validation

- Use `ddl-auto validate` mode.
- Validates the real database model against your entities.
- This doesn't make any changes to the database!





▲ Integration testing

- My advice: avoid using embedded databases like H2 on more complex projects.
- Use *Testcontainers* with your real production database.
- Use the same Liquibase/Flyway migrations.





JPQL

How can I use JPQL to fix the *N+1 problem*?
What can I do against the *cartesian product problem*?



▲ Why use JPQL?

- Very SQL like.
- Supports many features.
- Database independency.



▲ JPQL vs auto-generated repository methods

- Use JPQL when a query can't easily be expressed in a repository method name.
- Use JPQL when your repository method name becomes too long.
- Use JPQL when you think it will improve readability.

▲ The N+1 query problem

- The N+1 query problem happens when the data access framework executed **N additional SQL statements to fetch the same data that could have been retrieved when executing the primary SQL query.**
- Eager loading is prone to this issue.
- Lazy loading is prone to this issue.



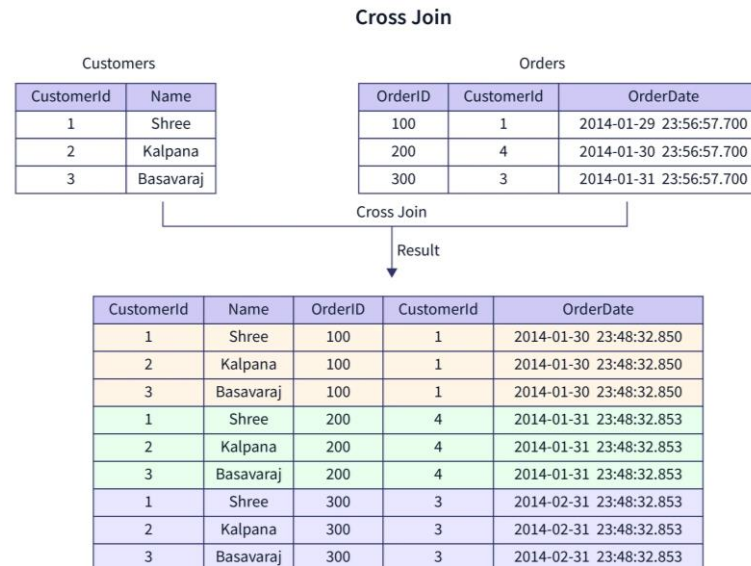
▲ JOIN FETCH directive in JPQL


- Solution for the N+1 problem.
- Avoid lazily navigating associations, or eagerly retrieving associations that you don't need.
- Careful! If you forget to “JOIN FETCH” properly, the persistence context will run queries on your behalf while you navigate the lazy associations (the *N+1 query problem*).
- When using JOIN FETCH we create a new problem: *The cartesian product problem*.



Cartesian product problem

- JOINS lead to big datasets.
- Reading and parsing a big dataset from the database takes some time.
- Hibernate needs to de-duplicate in-memory: could cause performance issues.





▲ What solution do I use to fix the cartesian product problem?

- It depends.
- Could avoid the `JOIN` and use lazy loading.
- Fetching associations with lazy loading causes more database traffic and latency but *could* be faster than loading and deduplicating a massive result set.

▲ MultipleBagFetchException

- Hibernate can no longer differentiate between *information that is supposed to be duplicated* and *information that was duplicated by the cartesian product*.
- Solution 1: Use `Set` instead of `List` (small datasets only)
- Solution 2: Use lazy loading.

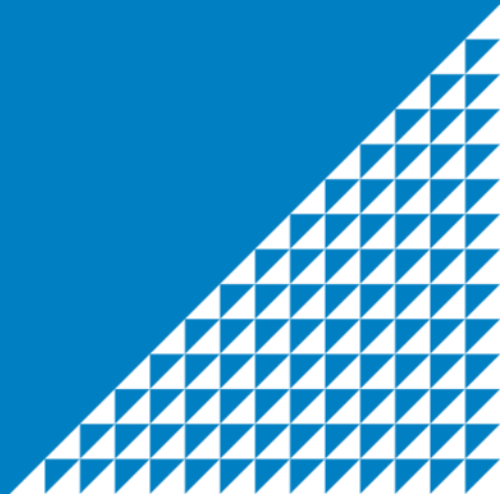


▲ Difference between JOIN FETCH and eager loading

- JOIN FETCH **affects the query** (will use a JOIN operation)
- Eager loading **affects the point in time** when a collection is loaded



Performance tips and common mistakes





▲ Tip 1: Avoid entity overhead

- Entities come with a lot of overhead (dirty checking, persistence context)

▲ Tip 1: Avoid entity overhead

- If your use-case doesn't require propagating changes to the database, use **read-only transactions**.
- `@Transactional(readOnly=true)`
- It eliminates dirty-checking.
- It eliminates loading the entity in the persistence context.

▲ Tip 1: Avoid entity overhead

- You can also eliminate entities by using **DTO projections**.
- Same result as using `@Transactional(readOnly=true)` but allows you to only extract fields that are required.

Tip 2: Read SQL logs

- Don't use `hibernate.show_sql` (statements are always logged to console)
- Use the logging framework instead:

```
logging:
```

```
  level:
```

```
    org.hibernate.SQL: debug
```

```
    org.hibernate.type.descriptor.sql: trace
```


▲ Tip 3: Don't use `@Transactional` in tests

- Why would you use `@Transactional` in a test?
 - To clean up data and ensure a deterministic test suite.
 - Behavior of `@Transactional` in tests = rollback
 - › In the Spring `TestContext` framework, transactions are managed by the `TransactionalTestExecutionListener`.
- What is the biggest problem when doing this?
 - Using `@Transactional` in tests is dangerous as it can hide production issues.
- Solution:
 - Clean up manually (`@AfterEach`).



▲ Tip 4: Use bulk operations

- Anti-pattern: retrieving entities and updating/deleting them one by one.
- Try to create an update/delete query for all relevant rows.
- If not possible to write a general query: think about batch processing and Hibernate memory usage.
 - https://docs.jboss.org/hibernate/core/3.6/reference/en-US/html_single/#batch

Tip 5: Avoid association fetching anti-patterns

- Open Session in View
- `FetchType.EAGER`
- Not using `JOIN FETCH` directive if necessary to avoid the N+1 problem.



▲ Tip 6: Use the same database system in your tests

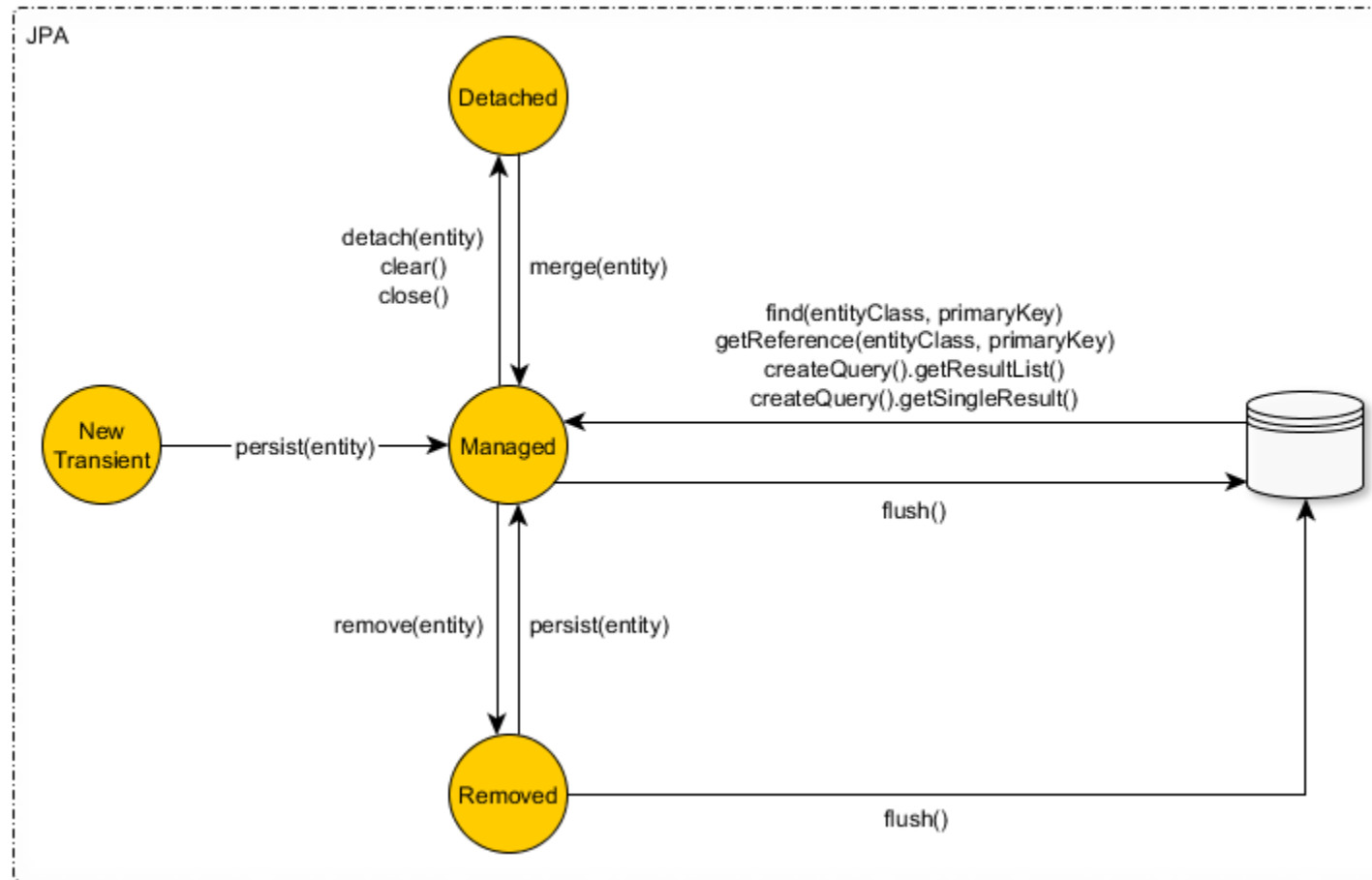
- No need for H2, use *Testcontainers* for integration tests.
- Ensures that your tests are representative for production.
- Allows you to use more database specific features.

▲ Tip 7: Understand @Transactional semantics

- Spot the problems:
 - Retrieve all comments (using @Transactional internally)
 - Give managers a raise



Tip 7: Understand @Transactional semantics





▲ Tip 8: Use DTOs in the web layer, not entities

- Always map entities to a *data transfer object* (DTO).
- Avoid security leaks and have a separation between the database model and the web model.



▲ Tip 9: Use `getReferenceById` instead of `findById` if you don't need the entity contents

- `getReferenceById` returns a proxied *reference* to an entity, it doesn't go to the database.
- You can use getters on the proxied reference, but this triggers lazy loading.
- Perfect when you only need the entity for establishing a relationship.
 - Example: Inserting a `PostComment` for a `Post`.



Tip 10: Don't trust Stack Overflow blindly



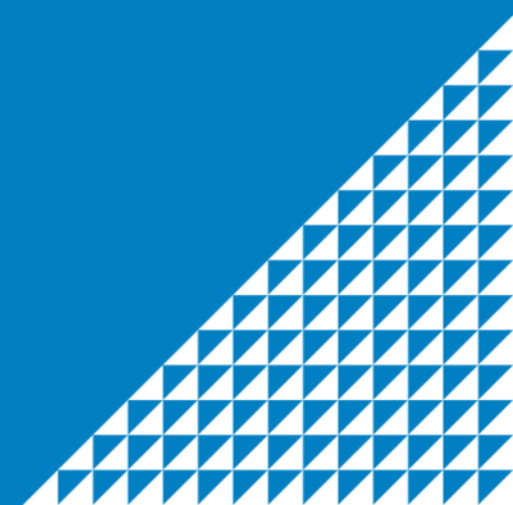


▲ In conclusion...

- Important to realize that using JPA comes with a performance impact.
- Don't be afraid to utilize JPA!
- Awareness is the most important: know what JPA does for you behind the scenes.



Further resources





▲ Further resources

Slides and demos are available on GitHub.

<https://github.com/raoulvdberge/jpa-with-hibernate>

▲ Further resources

- Vlad Micalcea (Hibernate contributor and expert)
 - <https://vladmihalcea.com/>
 - <https://vladmihalcea.com/the-open-session-in-view-anti-pattern/>
 - <https://vladmihalcea.com/eager-fetching-is-a-code-smell/>
- Hibernate User Guide
 - https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html
- JPA specification
 - <https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html>
- How does Spring Transactional work?
 - <https://dzone.com/articles/how-does-spring-transactional>
- JPA Join Types
 - <https://www.baeldung.com/jpa-join-types>