JPA with Hibernate

infoSupport
Solid Innovator

Raoul Van den Berge

IT Consultant

raoul.vandenberge@infosupport.com

Graduated from KdG in 2020

infoSupport
Solid Innovator

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?

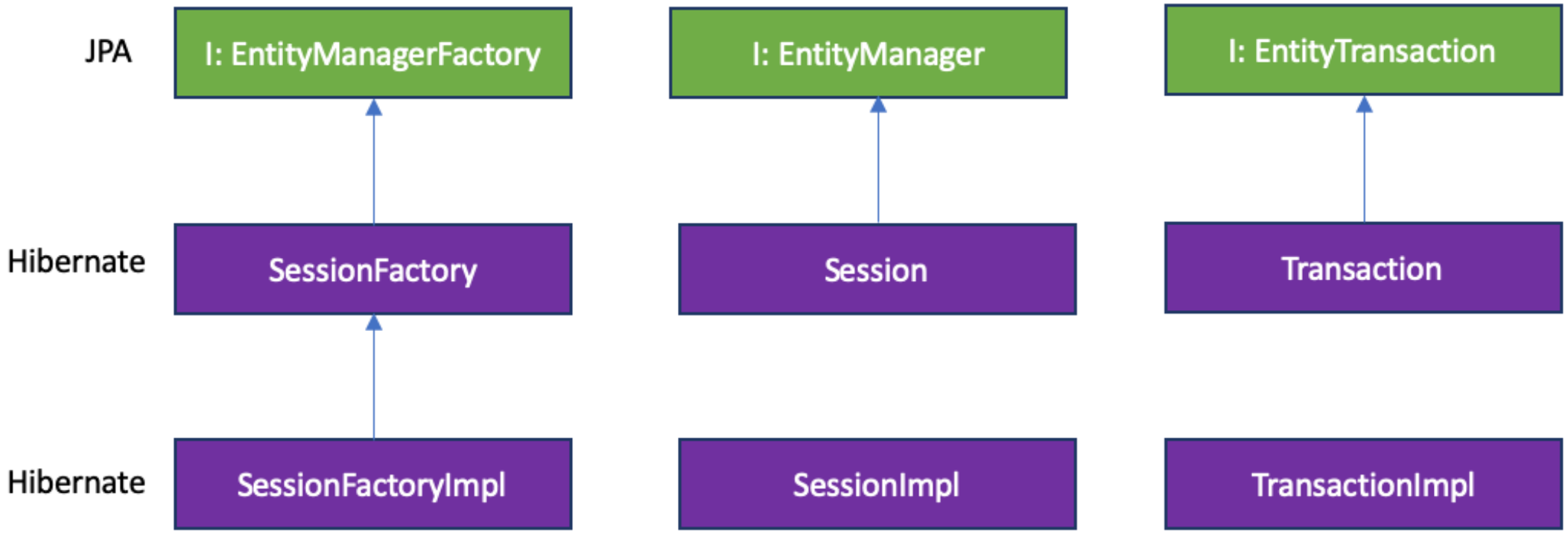
What is what?

- JPA
  - Java Persistence API
  - Now named: Jakarta 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

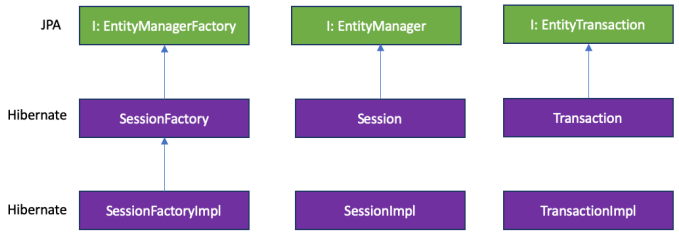
How does Spring manage transactions and how does it map to JPA/Hibernate?

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
`EntityManager`	JPA
`Session`	Hibernate
`CrudRepository`	Spring Data JPA
`@Transactional`	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

- Primarily using tests
- Sometimes a little web…

Problem

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

Solution 1: Use lazy loading with `FetchType.LAZY`

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

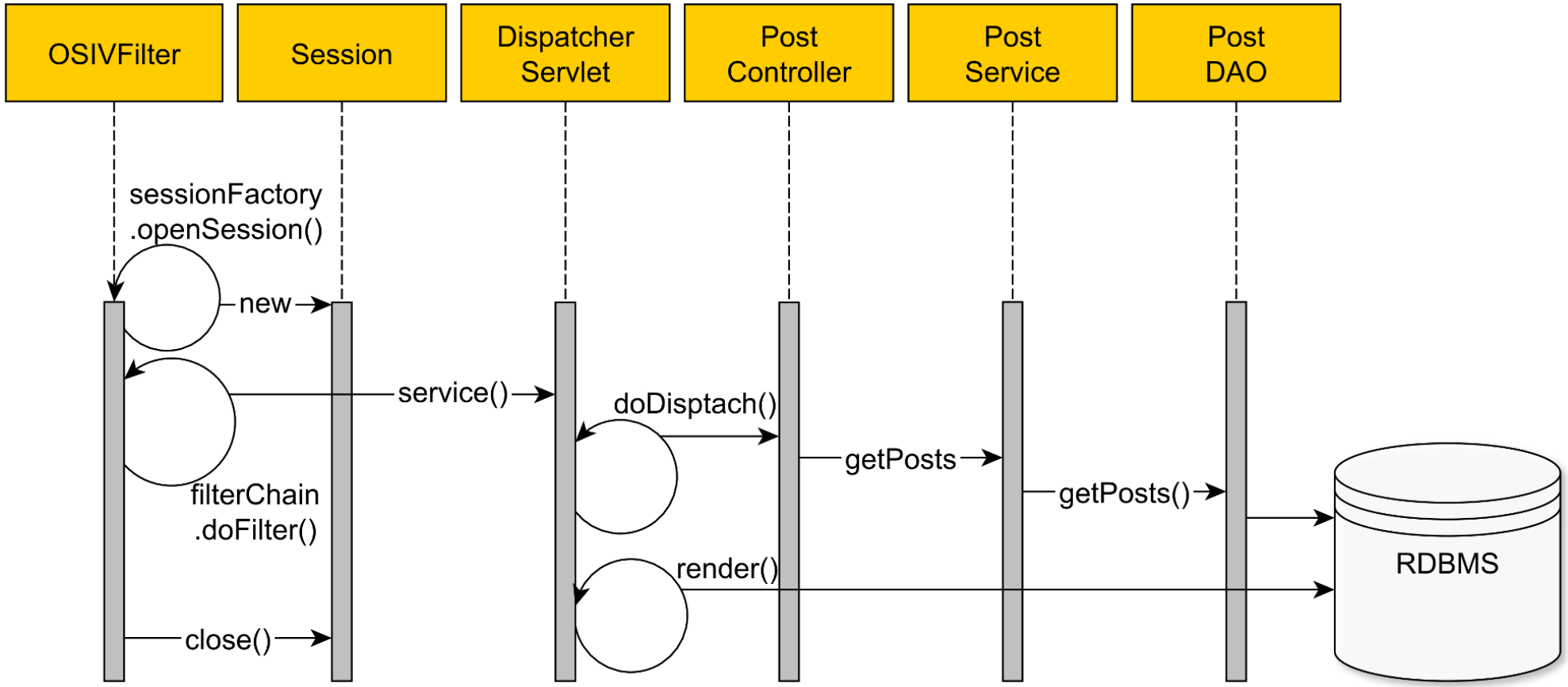
DEMO

```
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?!

- Ensures that a database session is active throughout the entire request.
- Instead of letting the business layer decide how it’s best to fetch all the associations that are needed by the View layer, it forces the Persistence Context to stay open so that the View layer can trigger the lazy-loaded collection initialization.
- By default “on” in Spring Boot (`spring.jpa.open-in-view=true`)

Open Session in View (OSIV)



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.
- The persistence context (`Session`) is closed after executing a method on a `JpaRepository` *(if not in a transaction).*
- If the persistence context is closed, when trying to access a non-initialized lazy association, the infamous `LazyInitializationException` is thrown.

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

23

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

```
@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

edited Nov 30, 2017 at 18:18
Ahmed Ashour
4,418 • 10 • 32 • 49

answered Sep 20, 2017 at 10:21
Smruti R Tripathy
733 • 1 • 9 • 15

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 site where you initialize a collection, ensure that it is wrapped in `@Transactional`.

`@Transactional` FAQ

- Where to use?
  - 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.

`@Transactional` FAQ

- When to use?
  - When using repositories, `@Transactional` is applied to repository scope automatically.
  - You’d lose the connection after getting a result back from the repository.
  - You’d get a `LazyInitializationException` when using Lazy Loading out of repository scope because the connection 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

- Often used in combination with *DTO projections* for read-only datasets.
- DTO = Data Transfer Object
- You should fetch just as much data you need to fulfill the requirements of a given business logic use case.
- Fetching too many columns than necessary has an impact, and that's why entities are not good candidates for read-only views.
- Good practice: make a separate model for reading and writing (CQRS).

DEMO

Schema generation and validation

How can I evolve my database?

Schema generation

- Let Hibernate update the schema?
- Don’t do this in production!
- Better write your own patches and migrations.

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

- 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).
- The incremental scripts can be included in your test setup.
- Flexibility of writing your own migration logic.

DEMO

Schema validation

- Use ddl-auto `validate` mode.
- Validates the real database model against your entities.
- This doesn’t do 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.

DEMO

`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*.

DEMO

Cartesian product problem

- `JOIN`s lead to big datasets.
- `JOIN`s lead to a data set with duplicates.
  - Solution 1: Fetch associations independently.
    - Downside: the N+1 problem is back.
  - Solution 2: Use `DISTINCT`.
    - Downside: Hibernate will de-duplicate all data in memory (`QueryTranslatorImpl needsDistincting`).

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

- It depends.
- Fetching associations with lazy loading causes more database traffic and latency but can be faster than deduplicating a massive result set with `DISTINCT`.

The cartesian product problem is a database problem

A fetch join has the same join semantics as the corresponding inner or outer join, except that the related objects specified on the right-hand side of the join operation are not returned in the query result or otherwise referenced in the query. Hence, for example, if department 1 has five employees, the above query returns five references to the department 1 entity.

More info:

- https://jakarta.ee/specifications/persistence/3.0/jakarta-persistence-spec-3.0.html#a4931
- https://developer.jboss.org/docs/DOC-15782#jive_content_id_Hibernate_does_not_return_distinct_results_for_a_query_with_outer_join_fetching_enabled_for_a_collection_even_if_I_use_the_distinct_keyword
- https://vladmihalcea.com/jpql-distinct-jpa-hibernate/
- https://in.relation.to/2016/08/04/introducing-distinct-pass-through-query-hint/
- https://thorben-janssen.com/hibernate-tips-apply-distinct-to-jpql-but-not-sql-query/

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
- `enable_lazy_load_no_trans`
- `FetchType.EAGER`
- Not using `JOIN FETCH` directive if necessary to avoid the N+1 problem.

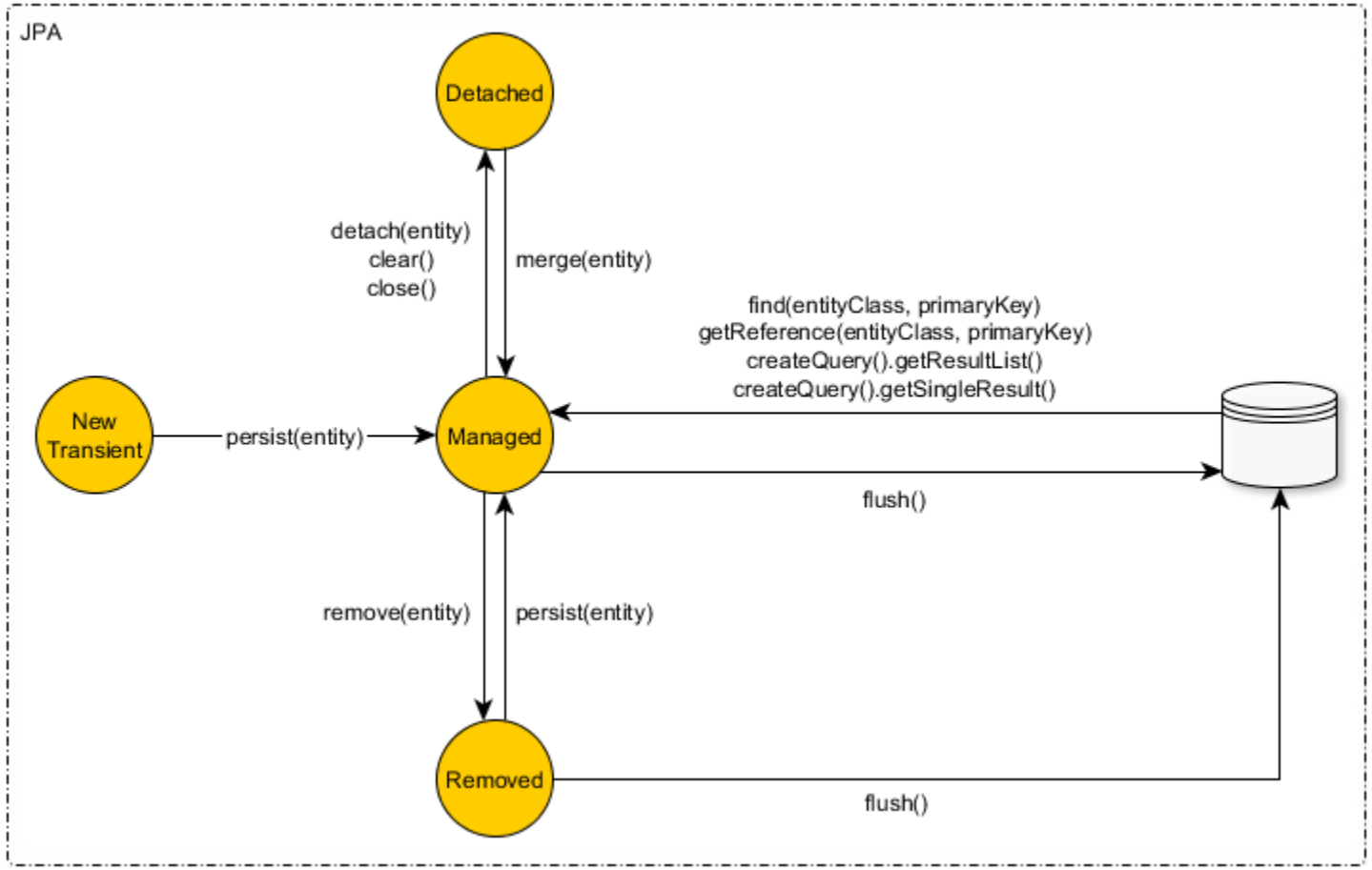
Tip 6: Use the same database system in your tests

- 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:
  - Demo: retrieve all comments
  - Demo: give managers a raise

Tip 7: Understand `@Transactional` semantics



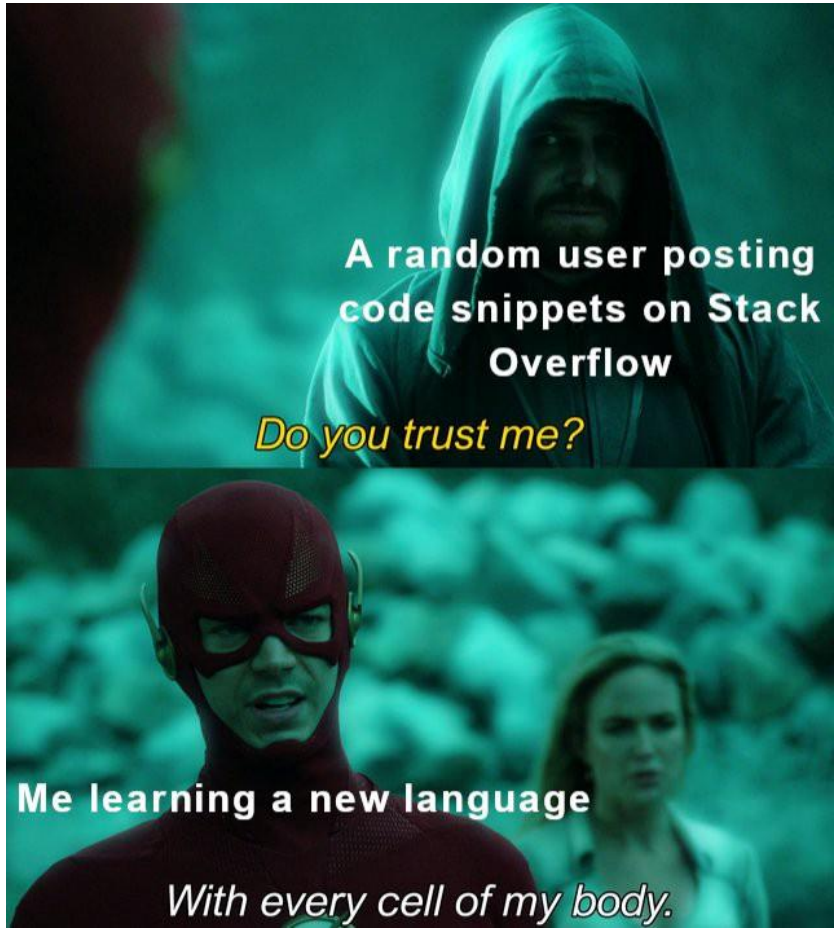
Tip 8: Use DTOs in the web layer, not entities

- 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 `getById` instead of `findById` if you don't need the entity contents

- `getById` 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…

- All my advice is a nuanced story…
- 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

Slides and demos

- 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