JPA with Hibernate

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

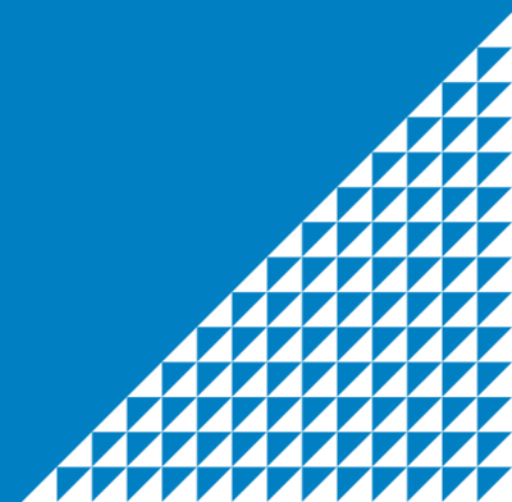
What am I actually using? Who has which responsibility?

What is what?

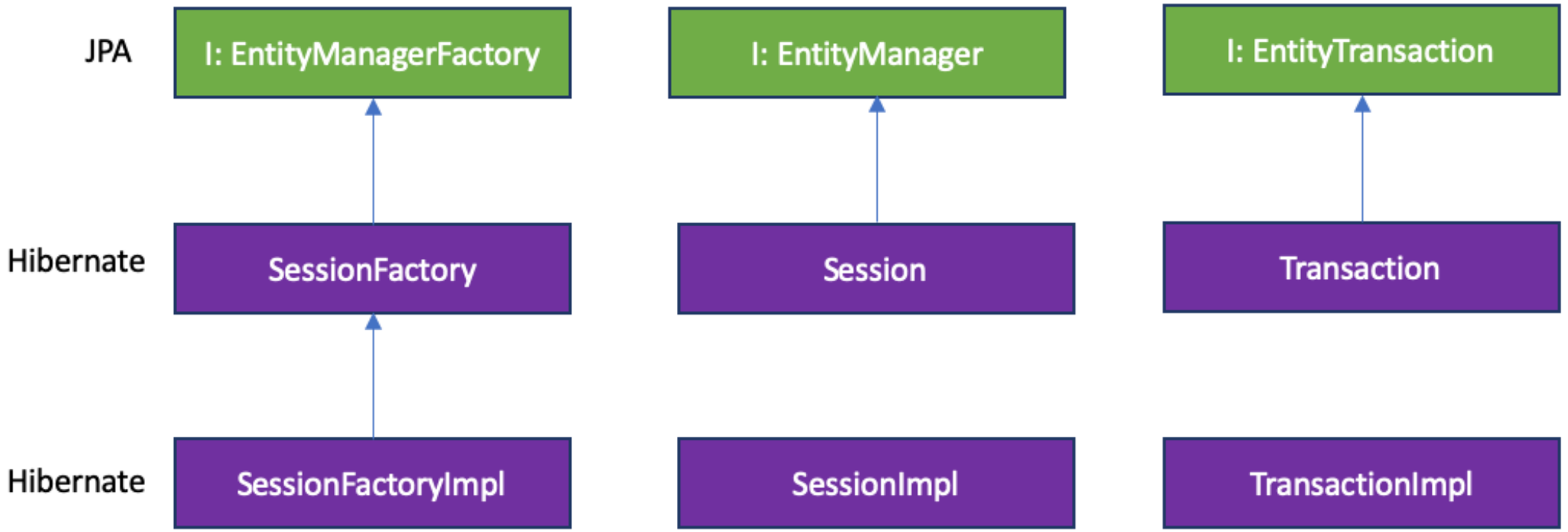
- JPA
  - Specification and interfaces.
- Hibernate
  - Implementation of the JPA API.
- Spring Data JPA
  - Layer on top of JPA which makes interacting with JPA less cumbersome.
  - Uses Hibernate behind the scenes.

Transaction management

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

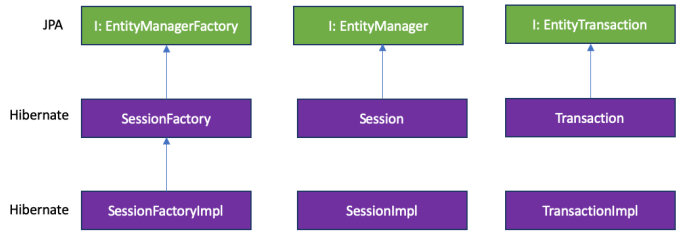


EntityManager vs Session



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;
```

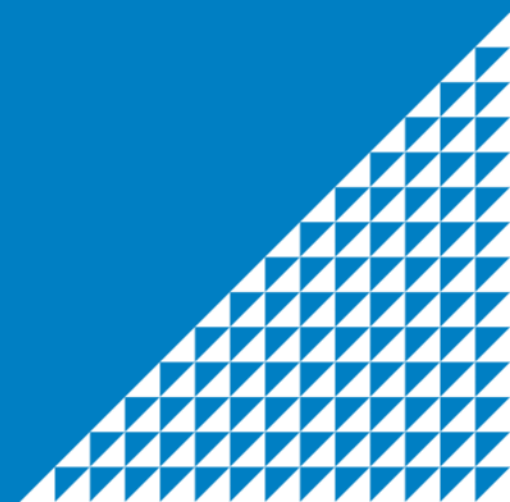
If you are using Spring Data JPA, you don’t need to use `EntityManager` directly (use a repository).

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 assocations: 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`?

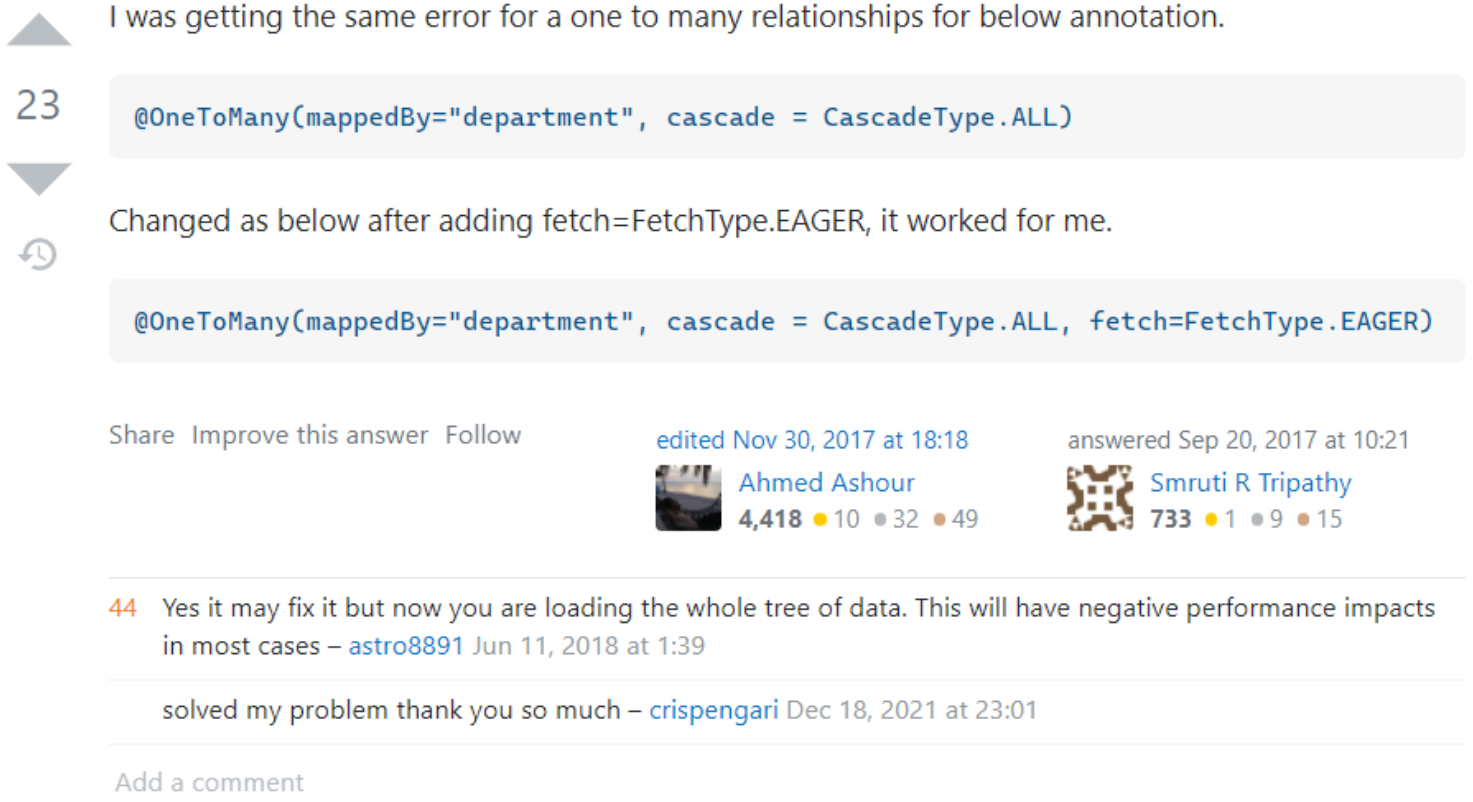
Solution 1: Use lazy loading with `FetchType.LAZY`

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

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`



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

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

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

Advice

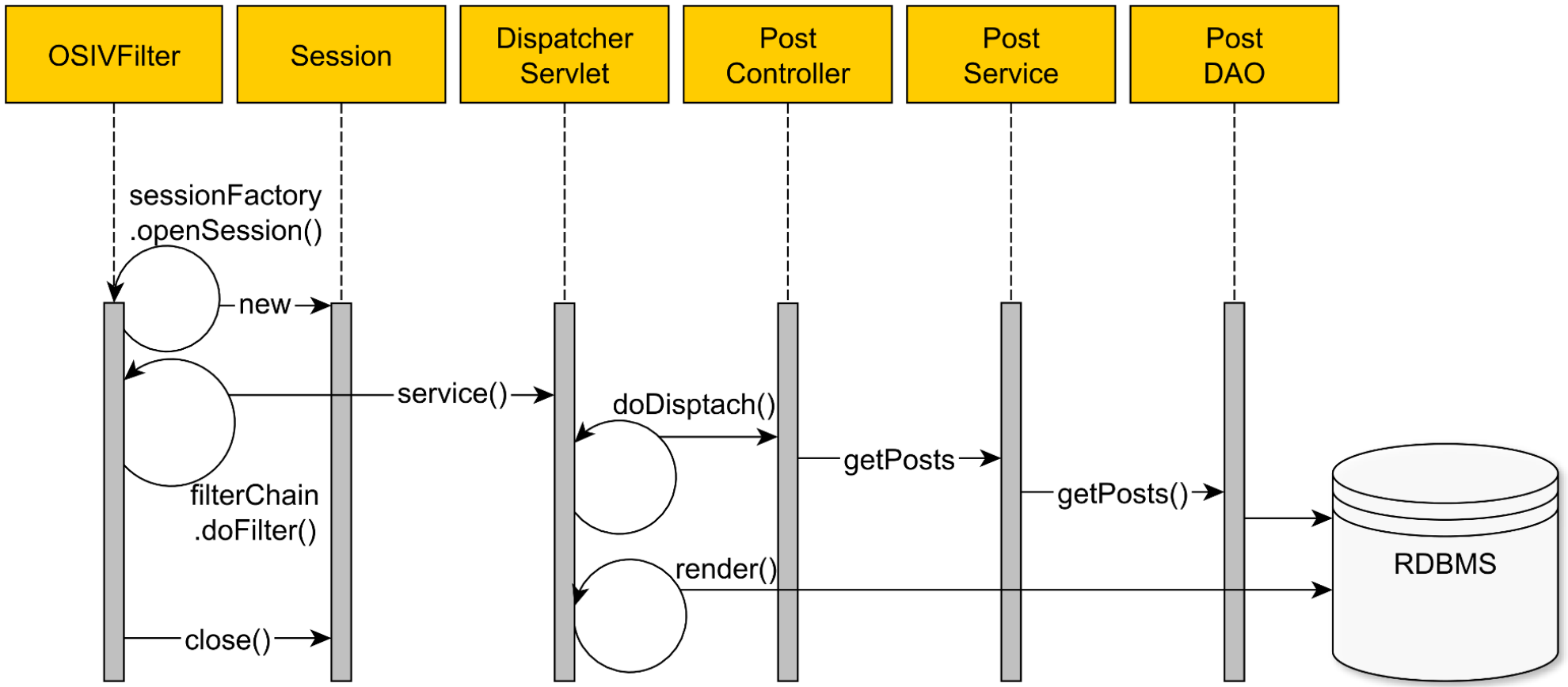
ASSOCIATION TYPE	DEFAULT FETCHING POLICY
@OneToMany	LAZY
@ManyToMany	LAZY
@ManyToOne	EAGER
@OneToOne	EAGER

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

Solution 3: Open session in view (OSIV)

- The most suggested “fix” for `LazyInitializationException`.
- Ensures that a 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 collection initialization.
- By default “on” in Spring Boot (`spring.jpa.open-in-view=true`)

Solution 3: Open session in view (OSIV)



Advice

- Don’t use OSIV!
- 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.

Solution 4: 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`.

Solution 5: custom query with HQL `FETCH`

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

`FETCH` with DTO projections

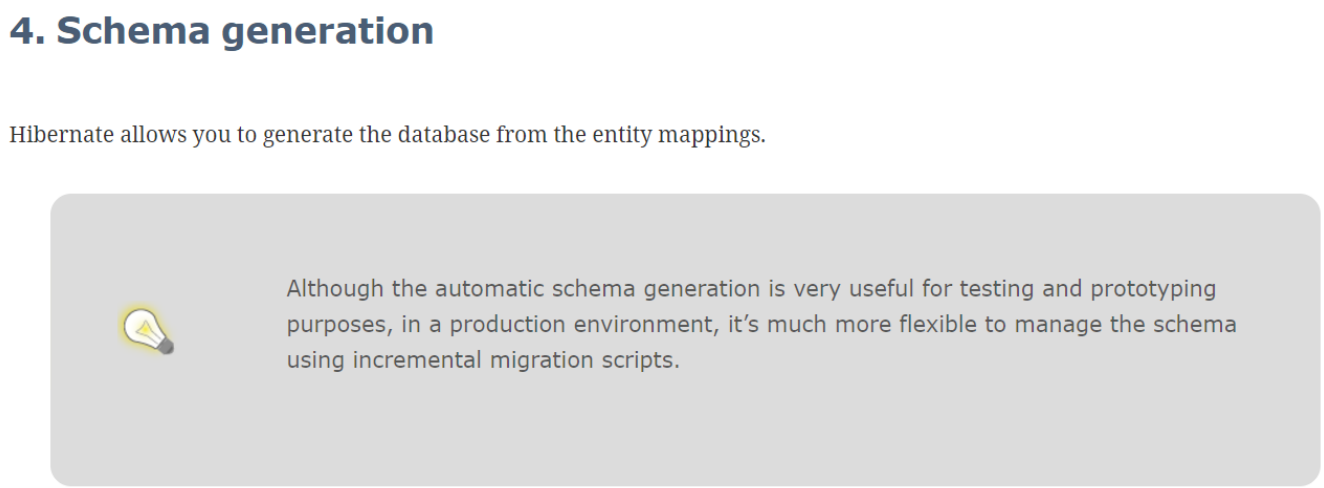
- Often used in combination with *DTO projections* for read-only datasets.
- You should always fetch just as much data you need to fulfill the requirements of a given business logic use case.
- Fetching too many columns than necessary has an impact, and that's why entities are not good candidates for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

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 tested on a QA server before being applied in production.
- Flexibility of writing your own migration logic.
- There is no need for manual intervention since the scripts can be run by Flyway/Liquibase, hence it reduces the possibility of human error associated with running scripts manually.

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: don’t use an embedded database like H2.
- 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.

`FETCH` directive in JPQL

- Solution for the N+1 problem.
- Avoid lazily navigating associations, or eagerly retrieving associations that you don’t need.
- When using `FETCH JOIN` we generate a new problem: *The cartesian product problem*.
- 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*).

Cartesian product problem

- Too many `JOIN`s lead to big datasets and duplicate results.
- Performance problems: Hibernate needs to read and de-duplicate all data.
- Solution: Fetch associations independently (be careful for the N+1 problem) or use `DISTINCT`.

DOCTOR_NAME	APPOINTMENT_ID	APPOINTMENT_TIME	PATIENT_NAME
Doctor John	1	2019-06-10 00:00:00	Patient 0
Doctor John	1	2019-06-10 00:00:00	Patient 1
Doctor John	1	2019-06-10 00:00:00	Patient 2
Doctor John	1	2019-06-10 00:00:00	Patient 3
Doctor John	2	2019-06-11 00:00:00	Patient 0
Doctor John	2	2019-06-11 00:00:00	Patient 1
Doctor John	2	2019-06-11 00:00:00	Patient 2
Doctor John	2	2019-06-11 00:00:00	Patient 3
Doctor John	3	2019-06-12 00:00:00	Patient 0
Doctor John	3	2019-06-12 00:00:00	Patient 1
Doctor John	3	2019-06-12 00:00:00	Patient 2
Doctor John	3	2019-06-12 00:00:00	Patient 3

(12 rows, 1 ms)

Cartesian product problem

- For more info see:
  - 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

Performance tips and common mistakes

Mistake 1: Using entities for read-only operations.

- Entities come with a lot of overhead.
- Use DTO projections if possible.

Mistake 2: Using `hibernate.show_sql`

- Statements are always logged to console.
- Use the logger framework instead:

```
logging:
  level:
    org.hibernate.SQL: debug
    org.hibernate.type.descriptor.sql: trace
```

Mistake 3: Using `@Transactional` in tests

- Why would you use `@Transactional` in a test?
  - To clean up data and ensure a deterministic test suite.
- 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`).

Mistake 4: Updating or removing entities one-by-one

- 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

Mistake 5: Using association fetching anti-patterns

- Open session in view
- `enable_lazy_load_no_trans`
- `FetchType.EAGER`
  - Including the default for all-to-one associations.
- Not using `JOIN FETCH` directive if necessary to avoid the N+1 problem.

Mistake 6: Not offloading enough to the database

- The database is here to work for you!
- Don’t be afraid to use views, procedures, etc. with Hibernate.

Mistake 7: Using a different database system in your tests

- No need for H2, use *Testcontainers* for integration tests.
- Ensures that your tests are representative for production.

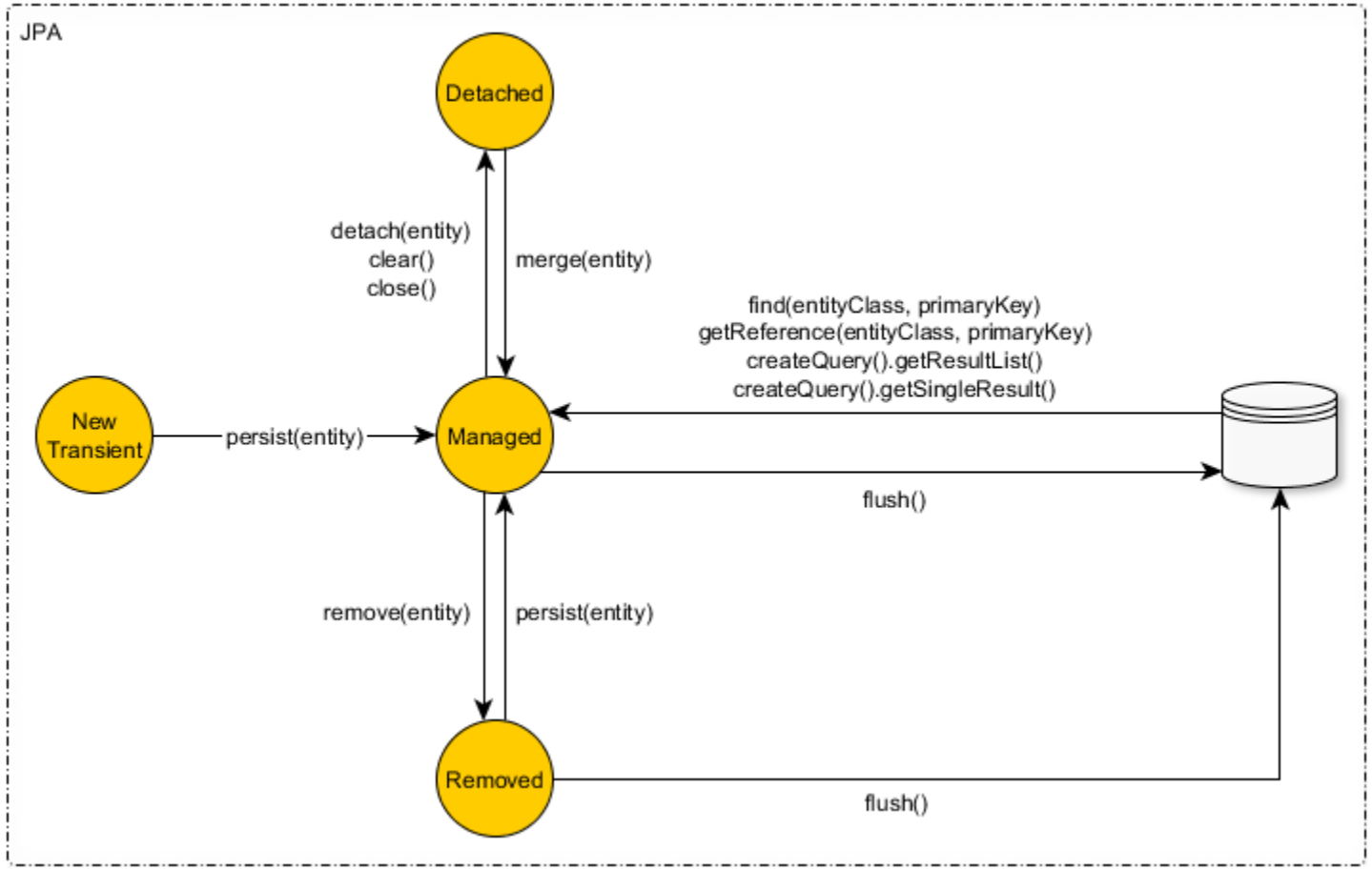
Mistake 8: Using `flush` without reason

- It forces Hibernate to perform dirty checking and writing updates to the database early.
- Let Hibernate combine updates into 1 SQL update, bundle multiple identical statements into a batch call, etc.
- Rare exception: batch updates.

Mistake 9: Not understanding `@Transactional` semantics

- Spot the problems:
  - Demo: retrieve all comments
  - Demo: give managers a raise

Mistake 9: Not understanding `@Transactional` semantics



Mistake 10: Leaking entities to web layer

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

Mistake 11: Including lazy associations in `toString()` of an entity

- It forces Hibernate to retrieve (an otherwise lazy collection) whenever `toString()` is called.
- With *Lombok* it's easy to run into this with `@ToString`.
  - Use `@ToString.Exclude` on the lazy associations.
  - https://projectlombok.org/features/ToString

Mistake 12: Trusting Stack Overflow blindly

Further resources

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
- How does Spring Transactional work?
  - https://dzone.com/articles/how-does-spring-transactional

Questions?