Performance Optimization

Spring Boot Applications with Hibernate

Building Faster, More Efficient Applications

Agenda

1. **Common Performance Issues**

- N+1 Query Problem
- Row explosion & Excessive Data Transfer
- Open Session in View
- Connection Pool Exhaustion
- External API calls

2. **Diagnostic Tools**

- Monitoring
- Query Logging
- Distributed tracing & Observability

3. **Solutions & Exercices**

- Entity graph
- Disable Open Session in View
- BatchSize
- Strategic caching

Common Performance Issues

Understanding what slows your application down

The N+1 Query Problem

Row Explosion & Data Transfer

Poll time !

Scan to join the poll -->

ahaslides.com/WL15A AhaSlides

Spring Open in View

1. Who remembers the time when we had to deal with LazyInitializeException?

2. Have you ever noticed this warning?

```
2025-11-12T14:59:07.077+01:00  WARN 19656 --- [company-directory] 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
```

This seemingly innocent warning is actually alerting you to a major performance anti-pattern!

Open Session In View: The Silent Killer

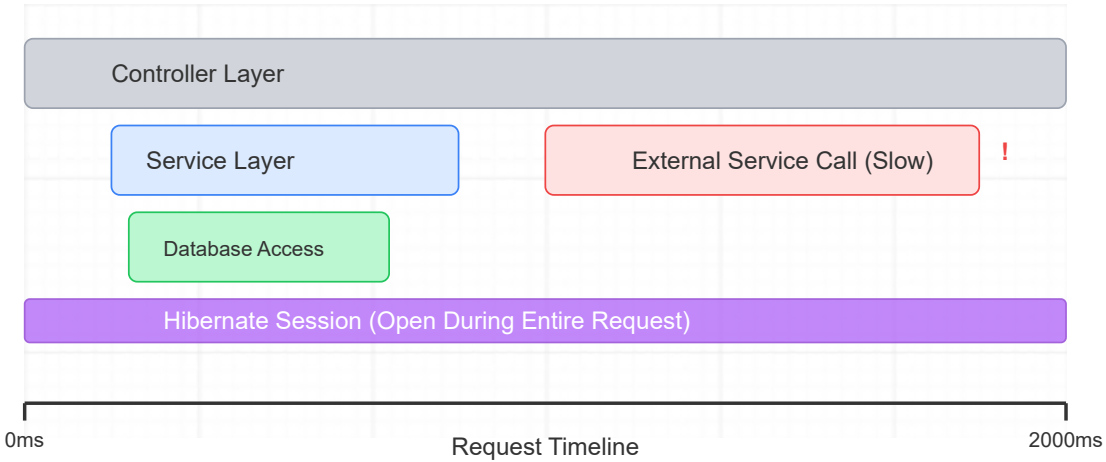
What is it?

Spring Boot's default behavior that keeps the Hibernate session open throughout the entire HTTP request, including view rendering.

Default Configuration:

```
spring.jpa.open-in-view=true
```

Open Session In View: Real-World Impact



In this example, a database connection is held open during a **slow external API call**, even though the database is not being used during this time.

With many concurrent requests, this can rapidly exhaust your connection pool.

Open Session In View: The Silent Killer

Benefits of Having It Enabled

- Convenient lazy loading in controllers/templates
- Simpler code - no need for explicit fetching
- Easier prototyping and development
- Allows loading entities in the mapper after the service layer has completed
- Avoid the infamous LazyInitializeException

Why it's a Problem

- Database connections held for the entire request duration
- Unpredictable N+1 queries in controllers/views
- Hidden performance costs
- Makes connection pool exhaustion more likely

Connection Pool Exhaustion

The Problem:

- Limited number of DB connections in the pool
- When all connections are in use, new requests wait
- Can lead to cascading failures
- Common causes:
  - Connections held too long (OSIV)
  - Inefficient queries
  - Connection leaks
  - Undersized pool

Symptoms:

- Increasing request latency
- HikariCP connection timeout exceptions
- Requests queuing up
- Application becomes unresponsive

Common Indicators:

```
HikariPool-1 - Connection is not available,
request timed out after 30000ms.HikariPool-1 - Thread starvation or clock leap
detected (housekeeper delta=XXms).
```

The External API Call

The Performance Impact

Network Latency

- External calls: 100ms - 2000ms
- Database queries: 1-10ms
- Application logic: microseconds

Blocking Operations

- Thread blocked waiting for response
- Resources tied up (connection pools)
- Cascading timeouts across services

Key Issues:

- Slow API = slow application response
- With OSIV, DB connections held during calls
- Low throughput, poor scalability
- Timeout errors under load

Agenda

1. **Common Performance Issues**

- N+1 Query Problem
- Row explosion & Excessive Data Transfer
- Open Session in View
- Connection Pool Exhaustion
- External API calls

2. **Diagnostic Tools**

- Monitoring
- Query Logging
- Distributed tracing & Observability

3. **Solutions & Exercices**

- Entity graph
- Disable Open Session in View
- BatchSize
- Strategic caching

Diagnostic Tools & Observability

Identifying performance bottlenecks

Performance Troubleshooting: First Steps

Observing Slow Response Time?

- High latency is usually the first symptom
- Users complain about "slowness"
- Timeouts start occurring under load
- Connection pool alerts triggered

SQL Logging in Action

During Development:

Always keep SQL logging enabled to catch N+1 queries and other issues early in the development cycle

Enable SQL Logging:

```
# application-local.properties or application-local.yml

spring.jpa.show-sql=true
```

What to Look For:

- Repeated Similar Queries: Indicator of N+1 problems
- Unexpected Query Volume: Too many queries for simple operations

Distributed Tracing & Observability

Agenda

1. **Common Performance Issues**

- N+1 Query Problem
- Row explosion & Excessive Data Transfer
- Open Session in View
- Connection Pool Exhaustion
- External API calls

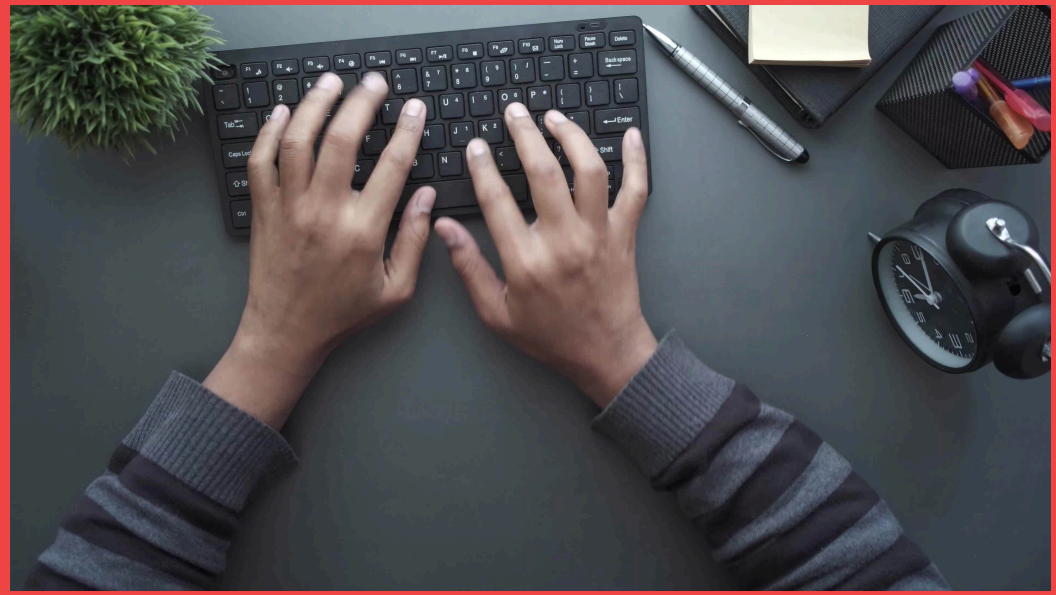
2. **Diagnostic Tools**

- Monitoring
- Query Logging
- Distributed tracing & Observability

3. **Solutions & Exercices**

- Entity graph
- Disable Open Session in View
- BatchSize
- Strategic caching

Solutions & Best practices



Hands on !

EntityGraph: Solving N+1 Elegantly

What is EntityGraph?

- JPA feature for defining graph of entities to fetch
- Declares which associations should be loaded eagerly
- More flexible than fixed fetch strategies
- Can be defined at entity or query level
- Supported natively by Spring Data JPA

EntityGraph: How to use

EntityGraph: Benefits & Pitfalls

Benefits

- Precise control over what's fetched
- Solves N+1 problem efficiently
- Can be applied conditionally
- Works with pagination

Common Pitfalls

- Over-fetching with too many associations
- Cascading entity graphs (nested associations)
- Using with very large collections

First exercice

- Check out the code: https://github.com/kevinrigot/spring-boot-perf-training
- `mvn install`
- Run the app ("Run Spring boot" in the IntelliJ launch configuration)
- in api/request , run "Search companies - N+1 test case"
- Use the tools to find out the problem (spoiler alert; it’s the name of the request)
- Fix it

Second exercice: Open Session in View

- Disable Open session in view `spring.jpa.open-in-view: false` in `application.yaml`
- Run the app ("Run Spring boot" in the IntelliJ launch configuration)
- in api/request , run "Search companies with full list of employees - Open In View test case"
- You should get a 500.
- Fix it.

@BatchSize: Optimizing Collection Fetching

What is @BatchSize?

- Annotation to configure batch fetching of collections
- Reduces N+1 queries by loading multiple collections in a single query

How does it work?

1. When you first access the collection of any parent. *eg: The list of employees of a department.*
2. Hibernate identifies up to x other element (*employees*) from the collection that are in the current session
3. Loads elements for all of them in a single query

```
Select * FROM employees WHERE deptId IN (id1, id2, id3, ..., id20)
```

4. When you access other elements not in the first batch, it triggers another batch

Third exercice: BatchSize

Instead of fetching all employees of all department of all companies, which would result in a cartesian of c * d * e.
Optimize it with batchsize.
Request is "Search companies with full list of employees - Open In View test case"

Strategic Caching External services

In modern applications, the most expensive operations are often the external api calls.

In some situation, we can avoid it.

eg: Get the employee name and firstname from HrServices

Solution: Get the full list of employees, and refresh it regularly.

```
@Cacheable(value = "allEmployees", unless = "T(org.springframework.util.ObjectUtils).isEmpty(#result)")
```

Fourth exercice: Caching

1. Run "Search employees - Caching"
2. It is slow...
3. Use `@Caching` and `ExternalEmployeeService.getAllEmployees`

Search and pagination

You might have notice the following warning during the exercices (kudos to whom did! And triple kudos to whom investigate why):

```
2025-11-13T15:16:19.419+01:00  WARN 16924 --- [company-directory] [nio-8080-exec-1] org.hibernate.orm.query: HHH90003004:
   firstResult/maxResults specified with collection fetch; applying in memory
```

What this means?

1. You're using pagination parameters and fetching collections

2. Hibernate is not applying the pagination at the db level as you might expect, but instead:

- Fetching all the data from the database

- Applying the pagination in memory after all data is loaded

Why is it a problem?

This approach can lead to serious performance issues and unexpected behavior:

1. Memory Usage: Your application loads the entire result set into memory, even if you only want a small page
2. Performance: The database transfers much more data than needed
3. Unexpected Results: The pagination may not work as expected because the collection joins can create multiple rows for a single entity

Search and pagination - How to deal with it?

1. Use Two Separate Queries Split your query into two steps:

```
// First query: Get paginated parent entities
Page<Company> findAll(@Nullable Specification<Company> spec, Pageable pageable);

// Second query: Fetch collections for the specific IDs
@EntityGraph(attributePaths = {"departments", "departments.chief"})
List<Company> findAllByIdIn(List<Long> ids);
```

Then in your service:

```
Page<Company> companiesIds = repo.findAll(spec, PageRequest.of(page, size, Sort.by("id")));
List<Company> companies = repo.findAllByIdIn(companiesIds.getContent().stream().map(Company::getId).toList());
return new PageImpl<>(companies, companiesIds.getPageable(), companiesIds.getTotalElements());
```

Fifth exercice: Solve this warning

1. First Search ids with pagination and no joins
2. Then fetch with all the entities by ids
3. Merge both response into a PageImpl


That's all Folks!