

Hibernate Session & Cache - Conversation Notes

User: So In Hibernate I have learnt that Session is not thread safe..

Assistant: Exactly ■ You got that right.

Here's why Session is not thread-safe:

- A Session represents a single unit of work with the database (one conversation between your app and DB).
- Internally, it holds things like:
 - First-level cache (objects you've loaded in this session).
 - Transaction context.
 - Pending SQL statements.
- If two threads use the same Session, they might interfere with each other's:
 - cache (one thread might overwrite what another stored),
 - flushing (queries might get mixed up),
 - transaction handling.

That's why each thread/request should get its own Session.

Instead, Hibernate gives us SessionFactory, which is thread-safe and designed to be shared. Each thread asks it for a fresh Session.

Think of it like this:

- SessionFactory = Coffee machine (safe to share)
- Session = Your cup of coffee (not sharable, it's yours only)

User: yes please

Assistant: Cool ■ let's connect the dots with Spring + Hibernate (or JPA).

Problem:

- In plain Hibernate, you'd need to do this manually:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
// work...
tx.commit();
session.close();
```

- If you had multiple threads/requests, you'd have to ensure each one gets its own Session → messy, error-prone.

Solution in Spring:

Spring handles this automatically using Transaction Management + ThreadLocal binding.

1. Spring Boot + JPA/Hibernate integration

- You don't manually create/close sessions.
- Instead, Spring manages an EntityManager (JPA wrapper over Hibernate Session).

2. @Transactional annotation

- When you mark a method with @Transactional, Spring:
 - Opens a new Hibernate Session (or reuses one bound to the current thread).
 - Begins a transaction.
 - Executes your method.
 - Commits/rolls back transaction.

- Closes the Session.

3. Thread safety

- Spring uses a ThreadLocal to bind one Session (actually EntityManager) to the current thread/request.
- So each thread/request gets its own independent session.
- You don't risk two threads messing with the same session.

Example:

```
@Service
public class StudentService {
    @Autowired
    private StudentRepository studentRepository;

    @Transactional
    public void saveStudent(Student student) {
        studentRepository.save(student);
        // No need to manually open/close Session
    }
}
```

User: Okay now explain with an example how a session shared across multiple threads can create problem in Hibernate ??

Assistant: Great question ■ This will make it crystal clear why Session is not thread safe.

Example:

```
Session session = sessionFactory.openSession();
```

```
Runnable task1 = () -> {
    Transaction tx = session.beginTransaction();
    Student s1 = session.get(Student.class, 1);
    s1.setName("Alice");
    tx.commit();
};
```

```
Runnable task2 = () -> {
    Transaction tx = session.beginTransaction();
    Student s2 = session.get(Student.class, 1);
    s2.setName("Bob");
    tx.commit();
};
```

```
new Thread(task1).start();
new Thread(task2).start();
```

Problems:

1. Shared First-Level Cache:

- Both threads work on the same cached object → last commit wins.

2. Transaction Conflicts:

- Only one transaction can be active per session → errors like Transaction already active.

3. Data Corruption / Dirty Reads:

- One thread may see another's uncommitted changes.

Correct Way:

Each thread/request must use its own Session.

User: So is it not a good idea ever to share a cache across multiple threads ?

Assistant: Nice catch ■ Let's clear this up.

1. First-Level Cache (Session Cache)

- Tied to a single Session.
- Not thread-safe → never share across threads.

2. Second-Level Cache

- Shared across Sessions (and threads).
- Providers like EhCache, Caffeine handle thread-safety internally.
- Safe to share across threads.

3. Query Cache

- Built on second-level cache.
- Safe to share across threads.

Rule of Thumb:

- First-level cache: never share.
- Second-level cache: safe to share.
- Query cache: safe to share.