

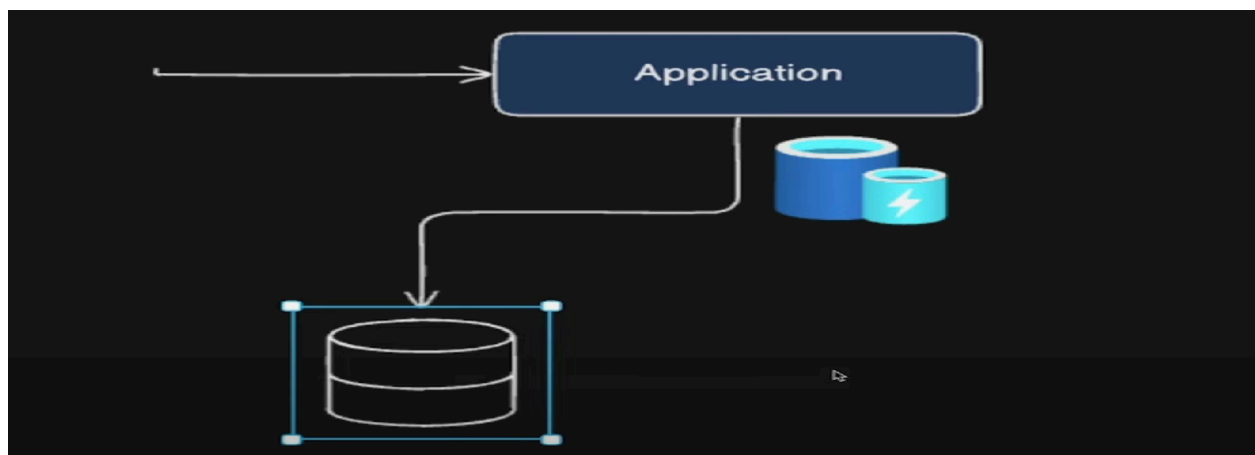
Spring Boot Caching and Hibernate Caching

Spring Boot Caching

Caching is a **mechanism to store frequently accessed data** (like database results, API responses, etc.) in memory (or other fast-access stores).

A **cache** is a **temporary storage** that sits between your **app** and the **database** or **service** (or another service).

It saves **recent or frequently used data** so that when the same data is needed again, it can be returned **faster** without asking the database again.



Advantages of Caching

- **Decreases database load** : Since frequently requested data comes from cache, it reduces the number of expensive database queries.
- **Improves Performance** : Cache provides data directly from memory, avoiding the need to contact the database or external service, making it much quicker.
- **Increases Application Scalability** : With reduced backend load, the system can handle more users and requests without slowing down.
- **Speeds Up Response Time** : Users get faster results because the app avoids recalculating or reloading data.

Types of Caching

1. In-Memory Caching

Cache is stored **locally inside the application's JVM memory**.

Works well for **single-instance** or **monolithic applications**. Fast access and easy to use.

How it works:

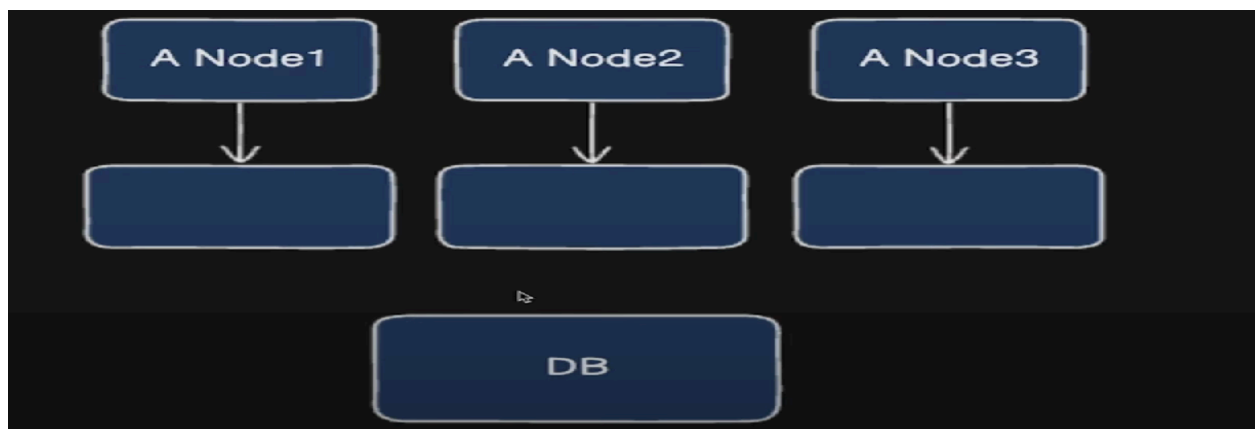
- Most frequently accessed data is stored in memory.
- When a method is called, it first checks if the result is already in the cache.
- If yes, it returns it instantly. If not, it executes the method and stores the result in cache.

Use case: Best for small-to-medium-sized data that changes infrequently and needs fast access.

Spring Boot's default cache type is **SimpleCache**, which uses **ConcurrentHashMap**

⚠ Problem with In-Memory Cache:

| Problem | Why it matters |
|-------------------------------|--|
| ✗ Not shared across instances | Each instance of your app has its own separate memory cache. |
| ✗ Cache lost on restart | If the app restarts, the cache is gone. |
| ✗ Not scalable | Doesn't work well in multi-server or cloud environments. |



- You have **3 application nodes**: A Node1, A Node2, and A Node3
- Each node has **its own local in-memory cache**
- All nodes are connected to the **same DB**

✗ Problem:

- Cache is **not shared** across nodes.
- If Node1 caches a student, and the next request goes to Node2, it **won't find it in its local cache**, causing a **DB hit** again.
- This results in **redundant DB queries**, **higher latency**, and **wasted memory**.

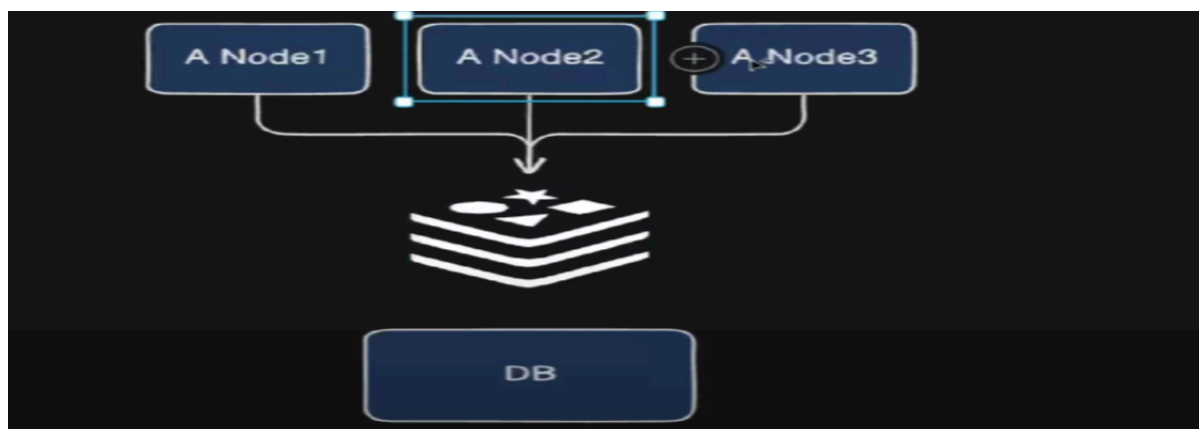
2. Distributed Caching

Cache is stored outside the application in a distributed system (like a separate server or cluster), accessible by multiple instances of the app.

Examples:

- Redis
- Memcached

Use case: Best for **large-scale applications** with multiple servers or microservices.



All 3 nodes: A Node1, A Node2, A Node3 are connected to a single shared Redis cache

Redis is used as the central distributed cac

Advantages:

- Cached data is shared between all app nodes
- If Node1 caches a student, Node2 and Node3 can also read that same data
- Improved performance
- Less load on the DB
- Scales well in cloud and microservices environments

How to Enable Caching in Spring Boot (In-Memory Caching)

1. Add Dependency

If you're using Spring Boot Starter, it's often already included.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Otherwise:

```
<!-- Add in pom.xml -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

2. Enable Caching in Main Application

EnableCaching is a Spring annotation used to enable caching support across the entire application.

Without it, annotations like **@Cacheable**, **@CachePut**, and **@CacheEvict** won't have any effect.

```
@SpringBootApplication
@EnableCaching
public class CollegeManagementApplication {
    public static void main(String[] args) {
        SpringApplication.run(CollegeManagementApplication.class, args);
    }
}
```

3. Use **@Cacheable** to Cache Method Results

@Cacheable is a Spring annotation that stores the result of a method (like a database call) in cache, so the next time the method is called with the same input, the cached result is returned instead of fetching from the database again.

```
@Cacheable(value = "studentByName", key = "#name")
public List<Student> findByName(String name) {
    List<Student> std =
studentRepository.findByNameContainingIgnoreCase(name);
    if (std.isEmpty()) {
        throw new StudentsNotFoundException("No Students found in the
database.");
    }
    return std;
}
```

```
@Cacheable(value = "students")
public List<Student> getAllStudents() {
    List<Student> std = studentRepository.findAll();
    if (std.isEmpty()) {
        throw new StudentsNotFoundException("No Students found in the
database.");
    }
    return std;
}
```

```

@Cacheable(value = "studentById", key = "#id")
public Student getStudentById(Long id) {
    Student std = studentRepository.findById(id)
        .orElseThrow(() -> new StudentNotFoundException("Student with
ID " + id + " not found.));
    return std;
}

```

4. Use @CacheEvict to Remove Cache When Updating

@CacheEvict tells Spring to **remove entries from the cache**.

This annotation tells Spring "**remove entries from the cache**" when this method runs.

Case-1 : It removes only the entry for that **id**.

```

@CacheEvict(value = "students", key = "#id")

```

This **evicts (removes) only one entry** from the **students** cache based on the key.

If you're caching **students by ID**, like this:

```

@Cacheable(value = "students", key = "#id")
public Student getStudentById(Long id) { ... }

```

Then you can use:

```

@CacheEvict(value = "students", key = "#id")
public void updateStudent(Long id, Student student) { ... }

```

Case-2: Remove all entries from these caches.

```

@CacheEvict(value = { "studentById", "students", "studentByName" }, allEntries = true)
public String deleteStudent(Long id) {
    Student std = studentRepository.findById(id)
        .orElseThrow(() -> new StudentNotFoundException("Student with
ID " + id + " not found.));
    studentRepository.deleteById(std.getId());
    return "Student with ID " + id + " deleted";
}

```

These are the **names of the caches** that we want to evict from.

You're using **three separate caches** elsewhere in your service:

| Cache Name | Where Used |
|-----------------|---|
| "studentById" | Caches result of <code>getStudentById(Long id)</code> |
| "students" | Caches result of <code>getAllStudents()</code> |
| "studentByName" | Caches result of <code>findByName(String name)</code> |

So here, you're saying:

```
"Clear all the entries in the above three caches."
```

allEntries = true

By default, Spring evicts a **single key** from the cache. But with **allEntries = true**, you're telling it:

"Remove all entries from these caches."

5. Use @CachePut to Update Cache Along with DB

@CachePut tells Spring to **execute the method and update the cache** with the return value.

To **update the cache with new values**, typically used in **update operations**.

Case-1 : It removes **only** the entry for that **id**.

```
@CachePut(value = "products", key = "#product.id")
public Product updateProduct(Product product) {

    return product;
}
```

case-2

```

@CachePut(value = "studentById", key = "#id")
@CacheEvict(value = {"students", "studentByName"}, allEntries = true)
public Student updateStudent(Long id, Student updatedStudent) {
    Student std = studentRepository.findById(id)
        .orElseThrow(() -> new StudentNotFoundException("Student with
ID " + id + " not found."));
    std.setName(updatedStudent.getName());
    std.setAge(updatedStudent.getAge());
    std.setAddress(updatedStudent.getAddress());
    return studentRepository.save(std);
}

```

You update the cache for the specific student.

You clear list and search caches, which are now invalid due to the update.

Simple Steps to Check Cache Behavior

Step 1: Add a Print Statement in Your Cached Method

```

@Cacheable(value = "studentById", key = "#id")
public Student getStudentById(Long id) {
    System.out.println("Fetching from DB for ID: " + id);
    Student std = studentRepository.findById(id)
        .orElseThrow(() -> new StudentNotFoundException("Student with
ID " + id + " not found."));
    return std;
}

```

Step 2: Call the API Endpoint Twice

For example:

First call (cache miss or DB hit): <http://localhost:8091/CollegeManagement/api/students/1>


```

35 @Cacheable(value = "studentById", key = "#id")
36 public Student getStudentById(Long id) {
37     System.out.println("Fetching from DB for ID: " + id);
38     Student std = studentRepository.findById(id)
39         .orElseThrow(() -> new StudentNotFoundException("Student with ID " + id + " not found"));
40     return std;
41 }
42
43 @CachePut(value = "studentById", key = "#id")

```

Problems Javadoc Declaration Console ×

CollegeManagementApplication [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (19-Jun-2025, 9:20:28 pm) [pid: 23480]

```

2025-06-19T21:20:36.829+05:30 INFO 23480 --- [ restartedMain] o.s.b.w.e.o.OptionalHeaderLoader
2025-06-19T21:20:36.884+05:30 INFO 23480 --- [ restartedMain] o.s.b.w.e.tomcat.TomcatWebServer
2025-06-19T21:20:36.895+05:30 INFO 23480 --- [ restartedMain] c.s.p.CollegeManagementApplication
2025-06-19T21:21:00.365+05:30 INFO 23480 --- [nio-8091-exec-2] o.a.c.c.C.[.][./CollegeManagement]
2025-06-19T21:21:00.365+05:30 INFO 23480 --- [nio-8091-exec-2] o.s.web.servlet.DispatcherServlet
2025-06-19T21:21:00.370+05:30 INFO 23480 --- [nio-8091-exec-2] o.s.web.servlet.DispatcherServlet
2025-06-19T21:21:00.981+05:30 INFO 23480 --- [nio-8091-exec-9] o.springdoc.api.AbstractOpenApiResource
Fetching from DB for ID: 1

```

Second call (cache hit): <http://localhost:8091/CollegeManagement/api/students/1>

```

34
35 @Cacheable(value = "studentById", key = "#id")
36 public Student getStudentById(Long id) {
37     System.out.println("Fetching from DB for ID: " + id);
38     Student std = studentRepository.findById(id)
39         .orElseThrow(() -> new StudentNotFoundException("Student with ID " + id + " not found."));
40     return std;
41 }
42
43 @CachePut(value = "studentById", key = "#id")

```

Problems Javadoc Declaration Console ×

CollegeManagementApplication [Java Application] C:\Program Files\Java\jdk-17.0.5\bin\javaw.exe (19-Jun-2025, 9:20:28 pm) [pid: 23480]

```

2025-06-19T21:20:36.895+05:30 INFO 23480 --- [ restartedMain] c.s.p.CollegeManagementApplication : Started
2025-06-19T21:21:00.365+05:30 INFO 23480 --- [nio-8091-exec-2] o.a.c.c.C.[.][./CollegeManagement] : Initiali:
2025-06-19T21:21:00.365+05:30 INFO 23480 --- [nio-8091-exec-2] o.s.web.servlet.DispatcherServlet : Initiali:
2025-06-19T21:21:00.370+05:30 INFO 23480 --- [nio-8091-exec-2] o.s.web.servlet.DispatcherServlet : Complete:
2025-06-19T21:21:00.981+05:30 INFO 23480 --- [nio-8091-exec-9] o.springdoc.api.AbstractOpenApiResource : Init dur:
Fetching from DB for ID: 1
Fetching from DB for ID: 2

```

You **will NOT** see the print statement.

This means the method was skipped → result returned from **cache**.

Distributed Caching

Spring Boot uses **in-memory** caching by default unless specified otherwise.

You can use **Redis as a distributed cache** with the same annotations: `@Cacheable`, `@CachePut`, and `@CacheEvict`.

To switch to Redis, simply:

1. Add the Redis dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

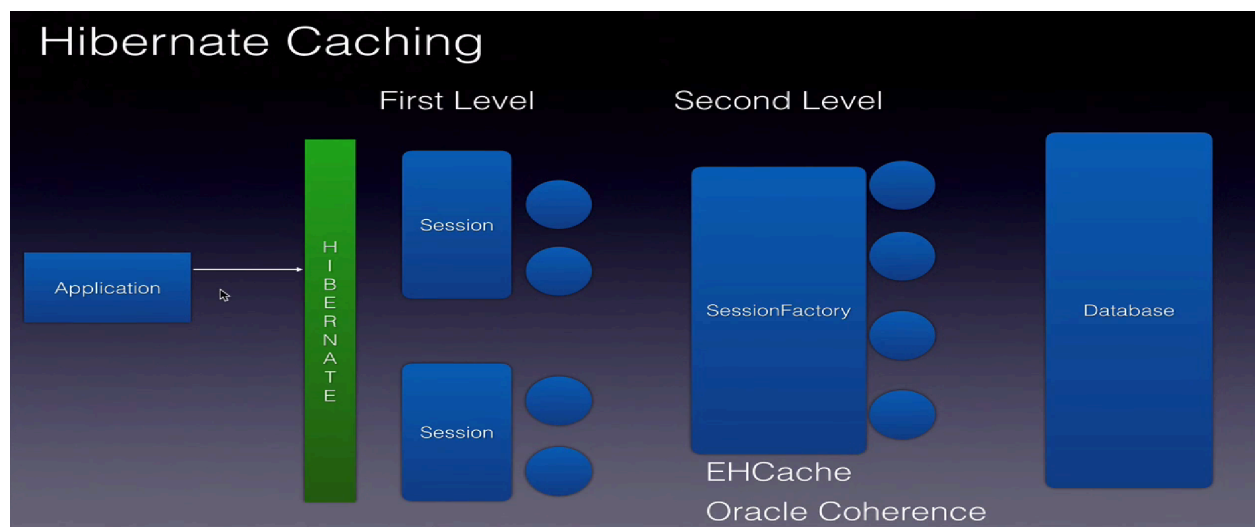
2. Configure the following in `application.properties`:

```
spring.cache.type=redis
spring.redis.host=localhost
spring.redis.port=6379
```

Hibernate Caching

Hibernate provides **built-in caching** to improve performance by **reducing database hits** for entity data.

Hibernate Caching is a mechanism used to **reduce the number of database queries by storing frequently accessed data in memory**. This improves application performance by minimizing costly database calls.



Hibernate offers **two levels of caching**:

1. First-Level Cache (L1 Cache)

- **Enabled by default.**
- Associated with the **Hibernate Session**.
- Data is cached **per session**.
- Cleared when the session is closed.

Example:

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

```
Student student1 = session.get(Student.class, 1); // Query goes to DB
```

```
Student student2 = session.get(Student.class, 1); // Data comes from cache
```

```
session.close();
```

The second `get()` call doesn't hit the database—it retrieves from the **session's cache**.

If I want to use first level cache in spring boot application

```
//first level cache example code in service layer
```

```
@Transactional
```

```
public void demonstrateFirstLevelCache(Long id) {
```

```
    System.out.println("1st call to findById:");
```

```
    Student student1 = studentRepository.findById(id).orElse(null); // Hits DB
```

```
    System.out.println("2nd call to findById:");
```

```
    Student student2 = studentRepository.findById(id).orElse(null); // Served from L1
```

```
Cache
```

```
    System.out.println("Are both objects same? " + (student1 == student2));
```

```
}
```

```
//first level cache example code in controller
```

```
@GetMapping("/students/cache/test/{id}")
```

```
public void testFirstLevelCache(@PathVariable Long id) {
```

```
    studentService.demonstrateFirstLevelCache(id);
```

```
}
```

2. Second-Level Cache (L2 Cache)

- **Optional:** Not enabled by default.
- Works **across sessions**.
- Requires **explicit configuration** and **external caching provider** (like Ehcache, Infinispan, Redis, etc.).
- Caches entities, collections, queries, etc.

Configuration Steps:

1. **Enable second-level cache** in `hibernate.cfg.xml` or `application.properties`.
2. Choose a cache provider (e.g., Ehcache).
3. Annotate or configure your entities to be cacheable.

Example using Ehcache:

a. Maven dependencies:

```
<dependency>
<groupId>org.hibernate</groupId>
<artifactId>hibernate-ehcache</artifactId>
<version>5.6.15.Final</version> <!-- Use the version compatible with your Hibernate version -->
</dependency>
```

b. Configuration in `hibernate.cfg.xml`:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">
  org.hibernate.cache.ehcache.EhCacheRegionFactory
</property>
```

c. Entity Setup:

```
@Entity
@Cacheable
@org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Student {
    @Id
    private int id;
    private String name;
    // other fields
}
```

✓ CacheConcurrencyStrategy options:

- **READ_ONLY** – good for reference data; no updates allowed.
- **READ_WRITE** – allows updates; uses a strategy to prevent stale data.
- **NONSTRICT_READ_WRITE** – allows updates, no strict locking.
- **TRANSACTIONAL** – for JTA and transactional caching.

If I want to use second level cache in spring boot application

1. Add Ehcache (or another provider) to **pom.xml**

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>5.6.15.Final</version> <!-- Use the version compatible with your Hibernate version
-->
</dependency>
```

2. Enable Hibernate L2 Cache in **application.properties**

```

# Enable Hibernate second-level cache
spring.jpa.properties.hibernate.cache.use_second_level_cache=true
spring.jpa.properties.hibernate.cache.use_query_cache=true
# Set Ehcache as the cache provider
spring.jpa.properties.hibernate.cache.region.factory_class=org.hibernate.cache.ehcache.EhCache
RegionFactory
# Optional: show SQL logs
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

```

3. Add **ehcache.xml** to **resources/** folder

```

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">
  <defaultCache
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="300"
    overflowToDisk="false" />
  <cache name="com.spring.project.entity.Student"
    maxEntriesLocalHeap="500"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="300"
    overflowToDisk="false" />
</ehcache>

```

4. Make **Student** Entity Cacheable

```

@Entity
@Table(name = "student_table")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Student {
    @Id

```

```
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id;  
private String name;  
private int age;  
private String address;  
}
```