SpringBoot - Caching

In this article, We will learn about caching and how to implement a cache in Spring Boot Application and increase the performance of the application.

Spring Boot has lot of features to manage the cache and it is simple to use. So lets get started.

What is Caching ?

Cache is a part of temporary memory (RAM). It lies between the application and the persistent database.

Caching is a mechanism used to increase the performance of a system. It is a process to store and access data from the cache.

It stores the recently used data. This helps to reduces the number of database hits as much as possible.

Why should we use the cache ?

There are some main reasons of using cache are as follows :

1. It make data access faster and less expensive
2. It improves the performance of the application.
3. It gives responses quickly.
4. Data access from memory is always faster than fetching from database.
5. It reduces the costly backend requests.

What data should be cached ?

The data which is to be cached is based on different requirement and scenarios. So caching data will differ for each application.

Below are some of the examples for which data should be cached :

1. List of product should be cached for an e-Commerce store.
2. The data that do not change frequently.
3. The frequently used read query in which results does not change in each call at least for a period.

Types of Caching

There are four types of caching are as follows :

1. In-memory Caching.
2. Database Caching.
3. Web Server Caching.
4. CDN Caching

In-Memory Caching

In-memory caching is a technique which is widely used. In this type of caching, data is stored in RAM. **Memcached** and **Redis** are examples of in-memory caching.

Memcached is a simple in-memory cache while **Redis** is advanced.

Database Caching

Database caching includes cache in database. It improves the performance by distributing a query workload.

We can optimize the default configuration in database caching to further boosting the application performance.

Hibernate first level cache is an example of database caching.

Web Server Caching

Web server caching is a mechanism that stores data for reuse.

It is cached for the first time when a user visits the page. If the user requests the same next time, the cache serves a copy of the page.

CDN Caching

The CDN stands for Content Delivery Network. It is a component used in modern web application.

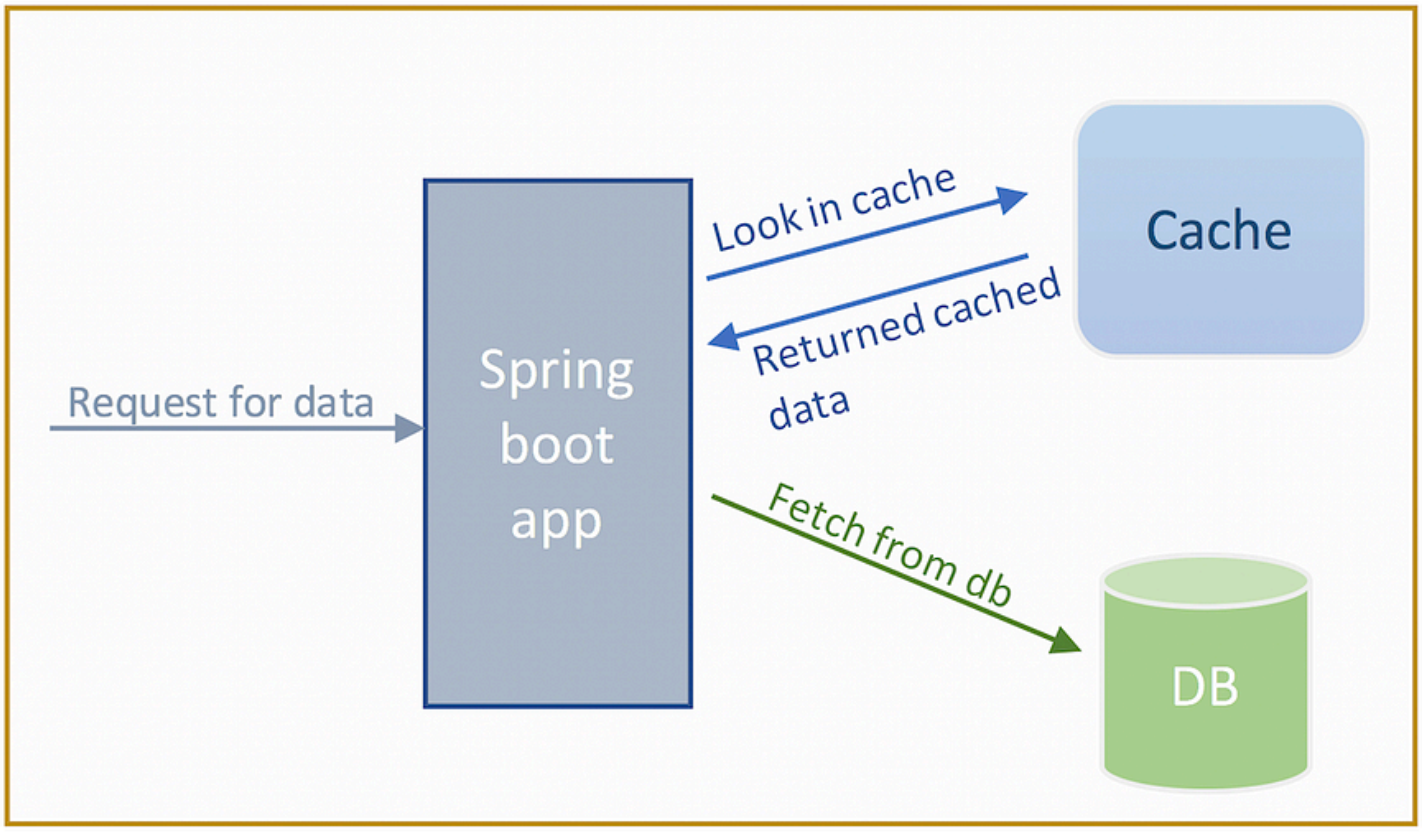
It improves delivery of the content by replicating common requested files such as Html Pages, images, videos, etc. across distributed set of caching servers.

Spring Boot Caching

Spring boot provides a **Cache Abstraction API** that allow us to use different cache providers to cache objects.

The below is the control flow of Spring boot caching.

Press enter or click to view image in full size



Spring Boot Caching

When the caching is enabled then the application first looks for required object in the cache instead of fetching from database. If it does not find that object in cache then only it access from database.

Caching makes the data access faster as data is fetched from database **only the first time** when it is required. Subsequently, it is fetched from the cache. Thus, Caching improves the performance of an application.

The cache abstraction works on two things :

1. **Cache Declaration :** It identifies the methods that need to be
2. **Cache Configuration :** The backing cache where data is stored and read from.

Spring Boot Caching Annotations

The following annotations are used to add caching support to Spring boot application.

@EnableCaching

It is a class level annotation. It is used to enable caching in spring boot application. By default it setup a **CacheManager** and creates in-memory cache using one concurrent **HashMap**.

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

It is also applied over a Spring configuration class as below :

```
@Configuration
@EnableCaching
public class CacheConfig {
 // some code
}
```

If you are using the default CacheManager and you do not want to customize it then there is no need to create separate class to apply **@EnableCaching**.

We can use external cache providers by registering them using **CacheManager**.

@Cacheable

It is a method level annotation. It is used in the method whose response is to be cached. The Spring boot manages the request and response of the method to the cache that is specified in the annotation attribute.

We can provide cache name to this annotation as follow :

```
@Cacheable(“employees”)
public Employee findById(int id) {
 // some code
}
```

This annotation has the following attributes :

1. cacheNames / value :

The **cacheNames** is used to specify the name of the cache while **value** specifies the alias for the cacheNames.

We can also provide a cache name by using the **value** or **cacheNames** attribute.

For example,

```
@Cacheable(value=”employees”)
public Employee findById(int id) {
  // some code
}@Cacheable(cachNames=”employees”)
public Employee findById(int id) {
  // some code
}
```

2. key :

This is the **key** with which object will be cached. It **uniquely identifies** each entry in the cache. If we do not specify the key then Spring uses the default mechanism to create the key.

For example,

```
@Cacheable(value=”employees”, key="#id")
public Employee findById(int id) {
  // some code
}
```

3. keyGenerator :

It is used to define your **own key generation mechanism**. We need to create custom key generator class.

For example,

```
@Cacheable(value=”employees”, keyGenerator=”customKeyGenerator”)
public Employee findById(int id) {
  // some code
}
```

4. cacheManager :

It specifies the name of cache manager. It is used define your own cache manager and do not want to use spring's default cache manager.

For example,

```
@Cacheable(value=”employees”, cacheManager=”customCacheManager”)
public Employee findByName(String name) {
  // some code
}
```

5. condition :

We can apply a condition in the attribute by using the **condition** attribute. We can call it as **conditional caching**.

For example, the following method will be cached if the argument name has length less than 20.

```
@Cacheable(value=”employees”, condition="#name.length < 20")
public Employee findByName(String name) {
  // some code
}
```

6. unless :

It specifies the object to to cached if it matches certain condition. SpEL provides a context variable **#result** which refers to the object that is fetched and we can apply condition on on its value.

For example,

```
@Cacheable(value=”employees”, unless=”#result.length < 20”)
public Employee findByName(String name) {
  // some code
}
```

@CachePut

It is a method level annotation. It is used to **update** the cache before invoking the method. By doing this, the result is put in the cache and the method is executed. It has same attributes of @Cacheable annotation.

```
@CachePut(value=”employee”)
public Employee updateEmployee(Employee employee) {
  // some code
}
```

Can we use @CachePut and @Cacheable into same method ?

There is difference between @Cacheable and @CachePut is that **@Cacheable** annotation skips the method execution while the **@CachePut** annotation runs the method and put its result in the cache.

If we use these annotations together then the application shows the unexpected behaviour. So two annotations cannot be used together.

@CacheEvict

It is a method level annotation. It is used to **remove** the data from the cache. When the method is annotated with this annotation then the method is executed and the cache will be removed / evicted.

We can remove single entry of cache based on **key** attribute. It provides parameter called **allEntries=true**. It evicts all entries rather one entry based on the key.

For example,

Evict an entry by key :

```
@CacheEvict(value=”employee”, key="#id")
public void deleteEmployee(int id) {
  // some code
}
```

Evict the whole cache :

```
@CacheEvict(value=”employee”, allEntries=true)
public void deleteEmployee(int id) {
  // some code
}
```

@Caching

It is allows multiple nested caching annotations on the same method. It is used when we want to use multiple annotations of the same type.

Java does not allow multiple annotations of same type to be declared for given method. To avoid this problem, we use **@Caching** annotation.

For example,

```
@Caching(evict = {
  @CacheEvict(“address”),
  @CacheEvict(value=“employee”, key=”#employee.id”)
})
public Employee getEmployee(Employee employee) {
  // some code
}
```

@CacheConfig

It is a class level annotation. It is used to share **common properties** such as cache name, cache manager to all methods annotated with cache annotations.

When a class is declared with this annotation then it provides default setting for any cache operation defined in that class. Using this annotation, we do need to declare things multiple times.

For example,

```
@Service
@CacheConfig(cacheNames=”employees”)
public class EmployeeService {  @Cacheable
  public Employee findById(int id) {
    // some code
  }
}
```

Spring Boot Cache Providers

The cache providers allow us to configure cache transparently and explicitly in an application.

The following steps are needed in order to configure any of these cache providers :

1. Add the annotation **@EnableCaching** in the configuration file.
2. Add the required **caching library** in the classpath.

3. Add the **configuration file** for the cache provider in the root classpath.

   The following are the cache provider supported by Spring Boot framework :

1. JCache (JSR-107)
2. EhCache
3. Hazelcast
4. Infinispan
5. Couchbase
6. Redis
7. Caffeine
8. Simple

JCache

JCache is the standard caching API for Java. It is provided by **javax.cache.spi.CachingProvider**.

It is present on the classpath. The spring-boot-starter-cache provides the **JCacheCacheManager**.

EhCache

The EhCache is an open source Java based cache used to boost performance. It stores the cache in memory and disk (SSD).

EhCache used a file called **ehcache.xml**. The **EhCacheCacheManager** is automatically configured if the application found the file on the classpath.

If we want to use EhCache then we need to add the following dependency :

```
<dependency>
 <groupId>org.ehcache</groupId>
 <artifactId>ehcache</artifactId>
</dependency>
```

HazelCast

The Hazelcast is a **distributed** in-memory data grid structure. It distributes the data equally among all the nodes. We can configure Hazelcast by using following property :

```
spring.hazelcast.config=classpath:config/demo-config.xml
```

If we want to use Hazelcast then we need to add the following dependency :

```
<dependency>
 <groupId>com.hazelcast</groupId>
 <artifactId>hazelcast</artifactId>
</dependency>
```

Infinispan

Infinispan is embedded java library. It is used as a **cache** or a **data grid**. It stores data in key-value form. It can be easily integrated with JCache, Spring, etc.

It does not have default file location so we need to configure it by using following property :

```
spring.cache.infinispan.config=infinispan.xml
```

If we want to use Infinispan then we need to add the following dependency :

```
<dependency>
 <groupId>org.infinispan</groupId>
 <artifactId>infinispan-core</artifactId>
</dependency>
```

Couchbase

Couchbase is a **NoSQL** database that can act as cache provider on top of the spring boot cache abstraction layer.

The **CouchbaseCacheManager** is automatically configured when we implement couchbase-spring-cache and configure couchbase.

If we want to use Couchbase then we need to add the following dependency :

```
<dependency>
 <groupId>com.couchbase.client</groupId>
 <artifactId>couchbase-spring-cache</artifactId>
</dependency>
```

Redis

Redis is a popular in-memory data structure. It is a keystore-based data structure which is used to persist data.

The **RedisCacheManager** is automatically configured when we configure **Redis**. The default configuration is set by using property **spring.cache.redis.***.

If we want to use Redis then we need to add the following dependency :

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

Caffeine

Caffeine is a high performance Java based caching library. It also provides an in-memory cache.

The spring boot automatically configures the **CaffeineCacheManager** if Caffeine is found in the classpath.

If want to use Caffeine then we need to add the following dependency :

```
<dependency>
 <groupId>com.github.ben-manes.caffeine</groupId>
 <artifactId>caffeine</artifactId>
</dependency>
```

Simple

It is the default implementation. It configures a **ConcurrentHashMap** as a cache store if spring boot does not find any cache provider in the classpath.

Spring Boot Cache Example

Now its time for some practical implementation. We will create simple spring boot application and implement cache mechanism into it.

Below are the steps to create spring boot application using Spring Initializr API :

1. Go to url : **https://start.spring.io/**
2. Fill out the Project name and package as per your application.
3. Add the dependencies **Spring Web** and **Spring Cache Abstraction**
4. Click on **Generate** the Project. When we click the Generate button then it will download the project in .zip file.
5. Extract the .zip file and import it into your Intellij or any IDE.

Now our project is created. Lets create all neccessary files.

pom.xml

Lets open pom.xml file and see which dependencies we have added to it.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.4</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>   <groupId>com.example</groupId>
  <artifactId>spring-boot-cache-example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>spring-boot-cache-example</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>11</java.version>
    <lombok.version>1.18.10</lombok.version>
  </properties>
```

```
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-cache</artifactId>
    </dependency>    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
    </dependency>    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
      <version>${lombok.version}</version>
      <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>

</project>
```

Main Class

Open the **SpringBootCacheExampleApplication.java** and add **@EnableCaching** annotation to enable cache in the application.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;

@SpringBootApplication
@EnableCaching
public class SpringBootCacheExampleApplication {

  public static void main(String[] args) {

    SpringApplication.run(SpringBootCacheExampleApplication.class, args);
  }

}
```

Model Class

Create model class **Employee.java.**

```
package com.example.model;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import java.io.Serializable;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
```

```
@Entity
public class Employee implements Serializable {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(nullable = false)
   private int id;

   @Column(nullable = false)
   private String name;

   private String address;
}
```

Repository Class

Create repository class **EmployeeRepository.java**.

```
package com.example.repository;

import com.example.model.Employee;
import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {

}
```

Service Class

Create service class **EmployeeService.java.**

```
package com.example.service;

import com.example.model.Employee;
import com.example.repository.EmployeeRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class EmployeeService {

```

```
    @Autowired
    EmployeeRepository employeeRepository;

    public Employee saveEmployee(Employee employee) {
        System.out.println("Save the record");
        return employeeRepository.save(employee);
    }

    @Cacheable(value = "employee", key = "#id")
    public Employee getEmployeeById(int id){
        System.out.println("Get the record with id : " + id);
        return employeeRepository.findById(id).orElse(null);
    }

    @CachePut(value = "employee", key = "#employee.id")
    public Employee updateEmployee(Employee employee) {
        System.out.println("Update the record with id : " + employee.getId());
        return employeeRepository.save(employee);
    }

    @CacheEvict(value = "employee", key = "#id")
    public void deleteEmployee(int id) {
        System.out.println("Delete the record with id : " + id);
        employeeRepository.deleteById(id);
    }
}
```

Controller Class

Create controller class **EmployeeController.java**

```
package com.example.controller;

import com.example.model.Employee;
import com.example.service.EmployeeService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
```

```
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/employees")
public class EmployeeController {

  @Autowired
  EmployeeService employeeService;

  @PostMapping("/save")
  public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee) {
    return new ResponseEntity<>(employeeService.saveEmployee(employee),
HttpStatus.CREATED);
  }

  @PutMapping("/update")
  public ResponseEntity<Employee> updateEmployee(@RequestBody Employee employee) {
    return new ResponseEntity<>(employeeService.updateEmployee(employee),
HttpStatus.OK);
  }

  @GetMapping("/{id}")
  public ResponseEntity<Employee> getEmployee(@PathVariable("id") int id) {
    return new ResponseEntity<>(employeeService.getEmployeeById(id), HttpStatus.OK);
  }

  @DeleteMapping("/{id}")
  public ResponseEntity<Void> deleteEmployee(@PathVariable("id") int id) {
    employeeService.deleteEmployee(id);
    return new ResponseEntity<>(HttpStatus.ACCEPTED);
  }
}
```

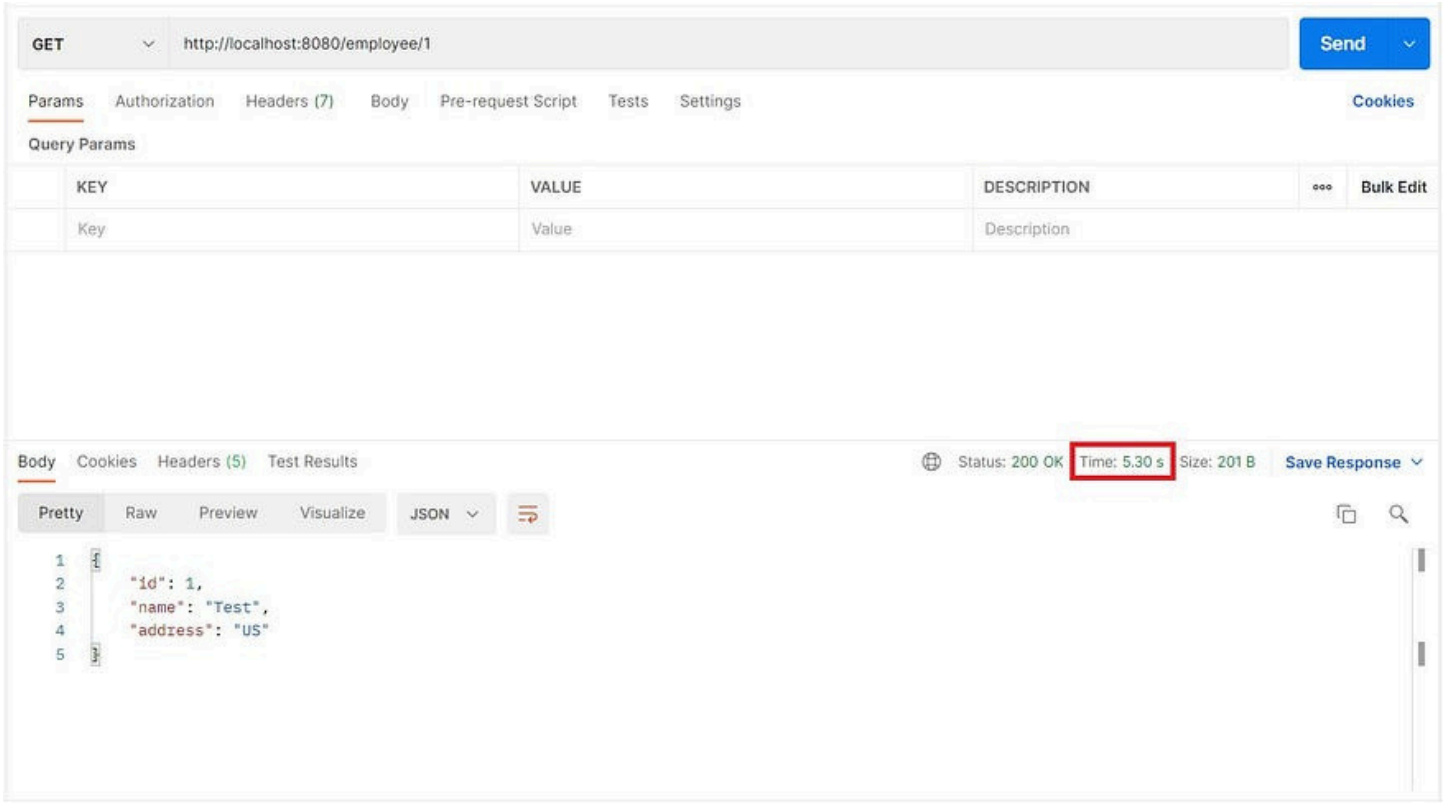
Running and Testing the Application

Now, we are going to run the application. Our application is running on port **8080**.

Press enter or click to view image in full size

Now, hit the url : **http://localhost:8080/employee/1**

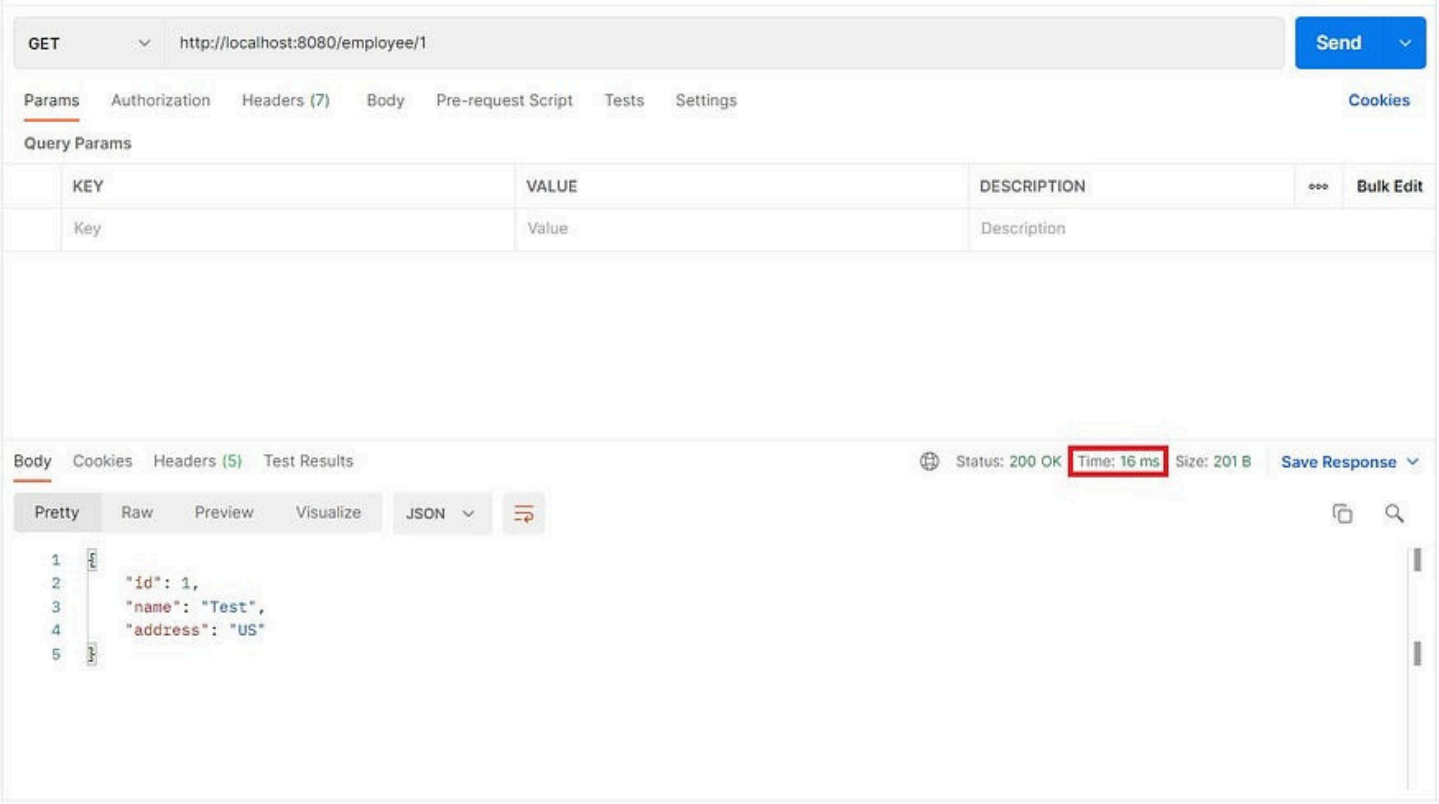
Press enter or click to view image in full size



When you hit the url for the first then it executes the service method and data is fetched from the database.

But if you again hit the same url then the results will be displayed very fast as this time the data is fetched from the cache.

Press enter or click to view image in full size



In this post, we have discussed all about Caching, Types of Cache, Caching annotations, different Caching providers and its configuration and how to add caching in Spring Boot application with example.

Now you have learned how to implement caching in Spring Boot application. With the help of caching we can increase the performance of application and reduces the application host.

More about Caching

Spring Boot caching — pragmatic explainer

Think of caching as a high-performance shortcut: keep expensive read results in a fast store so the application serves repeat requests with lower latency and lower backend cost. Spring Boot gives you a clean caching **abstraction** so you can instrument methods for caching with a few annotations and swap the underlying cache implementation (Caffeine, Ehcache, Redis, etc.) without changing business logic.

How it works (at-a-glance)

- Enable caching in the app (@EnableCaching).
- Annotate methods where results should be cached. Spring intercepts calls and consults the configured CacheManager.
- If a cached value exists for the cache key, return it; otherwise execute the method, store the result, and return it.
- You can evict/update cache entries when writes occur so cache and data stay consistent.

Core annotations & semantics (Java examples)

```java
// Enable caching
@SpringBootApplication
@EnableCaching
public class App { ... }
```

```java
@Service
public class ProductService {
  @Cacheable(value = "product", key = "#id", unless = "#result == null", sync = true)
  public ProductDto getProduct(Long id) {
    // expensive DB or remote call
  }
}
```

- value (or cacheNames) = cache name.
- key can be SpEL (#id, #root.methodName + ':' + #id, etc.).
- unless prevents caching for certain results (e.g., null).
- sync = true reduces cache stampede by synchronizing cache population for the same key.

@CachePut — update cache when method runs (useful for writes that also return the new value):

```java
@CachePut(value = "product", key = "#product.id")
public ProductDto saveProduct(ProductDto product) { ... }
```

@CacheEvict — remove entries (single or many):

```
// evict single key
@CacheEvict(value = "product", key = "#id")
public void deleteProduct(Long id) { ... }

// evict all entries
@CacheEvict(value = "product", allEntries = true, beforeInvocation = false)
public void deleteAllProducts() { ... }
```

@Caching — compose multiple cache annotations:

```
@Caching(
  put = {@CachePut(value="product", key="#p.id")},
  evict = {@CacheEvict(value="productList", allEntries=true)}
)
public ProductDto update(ProductDto p) { ... }
```

CacheManager & providers

Spring Boot auto-configures a CacheManager if you include a supported provider and set spring.cache.type (or rely on auto-detection). Two common choices:

- **Caffeine** — in-memory, very fast, JVM local. Great for single-instance apps or read-heavy local caches.
- **Redis** — distributed, durable (if configured), TTL-friendly. Ideal for multi-instance clustered apps.
- **Ehcache**, **Hazelcast**, **Infinispan**, etc. — choose per requirements.

Example: application.yml (auto-config for Caffeine)

```
spring:
  cache:
    type: caffeine
  caffeine:
    spec: maximumSize=500,expireAfterAccess=10m
```

Example: application.yml for Redis cache manager

```yaml
spring:
  cache:
    type: redis
  redis:
    host: redis.example.local
    port: 6379
```

Programmatic bean for Caffeine CacheManager:

```java
@Bean
public CacheManager cacheManager() {
  CaffeineCacheManager cm = new CaffeineCacheManager("product", "users");
  cm.setCaffeine(Caffeine.newBuilder()
            .maximumSize(1000)
            .expireAfterWrite(Duration.ofMinutes(10)));
  return cm;
}
```

Programmatic bean for RedisCacheManager with TTL per cache:

```java
@Bean
public RedisCacheManager redisCacheManager(RedisConnectionFactory cf) {
  RedisCacheConfiguration defaultConfig = RedisCacheConfiguration.defaultCacheConfig()
      .disableCachingNullValues()
      .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new GenericJackson2JsonRedisSerializer()));

  Map<String, RedisCacheConfiguration> perCacheConfigs = Map.of(
    "product", defaultConfig.entryTtl(Duration.ofMinutes(10)),
    "users", defaultConfig.entryTtl(Duration.ofMinutes(5))
  );

  return RedisCacheManager.builder(cf)
      .cacheDefaults(defaultConfig)
      .withInitialCacheConfigurations(perCacheConfigs)
      .build();
}
```

Key generation & collisions

- Default key generation uses method parameters. For complex keys use SpEL (#root.methodName + ':' + #user.id) or provide a custom KeyGenerator.
- Example custom KeyGenerator bean:

```
@Bean("customKeyGen")
public KeyGenerator keyGenerator() {
  return (target, method, params) -> {
    StringJoiner sj = new StringJoiner(":");
    sj.add(target.getClass().getSimpleName()).add(method.getName());
    for (Object p : params) sj.add(String.valueOf(p));
    return sj.toString();
  };
}
```

Then use @Cacheable(value="x", keyGenerator="customKeyGen").

Patterns & strategies

- **Cache-aside (explicit)** — most common: application checks cache, if miss load from DB and populate cache. (@Cacheable implements this.)
- **Read-through / Write-through** — cache provider coordinates read/write to the store (less common in Spring-managed setups).
- **Write-behind** — cache accepts writes asynchronously (be careful with consistency).
- Use TTL, maximum size and eviction policies (LRU, etc.) to bound memory.

Best practices (operational & design)

- Cache **only** expensive, read-heavy, mostly-idempotent operations. Avoid caching low-cost or highly dynamic data.
- Prefer **immutable DTOs** as cached values or defensively copy when returning to avoid shared-state bugs.
- Use unless="#result==null" or configure disableCachingNullValues() to avoid nulls filling cache.
- Set sensible TTLs and size limits. Without TTL, stale data risk increases.
- For distributed apps prefer Redis (or a distributed cache) for consistency across instances.
- Use @CachePut on write operations returning the new state, and @CacheEvict on deletes.
- Monitor cache hit/miss rates (Micrometer + Prometheus) and tune accordingly.
- Protect against **cache stampede** using sync=true or application-level locks / request coalescing.

- Remember serialization costs for remote caches (Redis): choose compact serializers (JSON vs MsgPack vs binary).

Common pitfalls

- Caching inside the same bean method call won't be intercepted by Spring proxy — calls must go through proxy (external call or use @Autowired self-injection).
- Mutable objects in cache lead to surprising bugs. Serialize/clone or use immutable DTOs.
- Forgetting to evict or update cache on data changes causes stale responses.
- Over-caching everything wastes memory and adds overhead — be selective.
- Ignoring cold-start: caches are empty after restart; consider warming strategies for critical keys.

Quick end-to-end example (service + repository pattern)

```
@Repository
public class ProductRepository {
  public ProductEntity findById(Long id) { /* DB call */ }
}

@Service
public class ProductService {
  private final ProductRepository repo;

  public ProductService(ProductRepository repo) { this.repo = repo; }

  @Cacheable(value = "product", key = "#id", unless = "#result == null", sync = true)
  public ProductDto getProduct(Long id) {
    ProductEntity e = repo.findById(id);
    return e == null ? null : mapToDto(e);
  }

  @CachePut(value = "product", key = "#dto.id")
  public ProductDto updateProduct(ProductDto dto) {
    ProductEntity updated = repo.save(mapToEntity(dto));
    return mapToDto(updated);
  }

  @CacheEvict(value = "product", key = "#id")
  public void deleteProduct(Long id) {
    repo.deleteById(id);
  }
}

```

Operational checklist before you ship

- Choose provider: Caffeine for single node, Redis for distributed.
- Configure TTL, max size, serializers, and null-value handling.
- Add metrics (cache hit/miss) and alerts for anomalous behavior.
- Add tests verifying caching behavior (mock repo, assert repo called once for repeated reads).
- Document which APIs are cached and their TTLs for cross-functional teams.

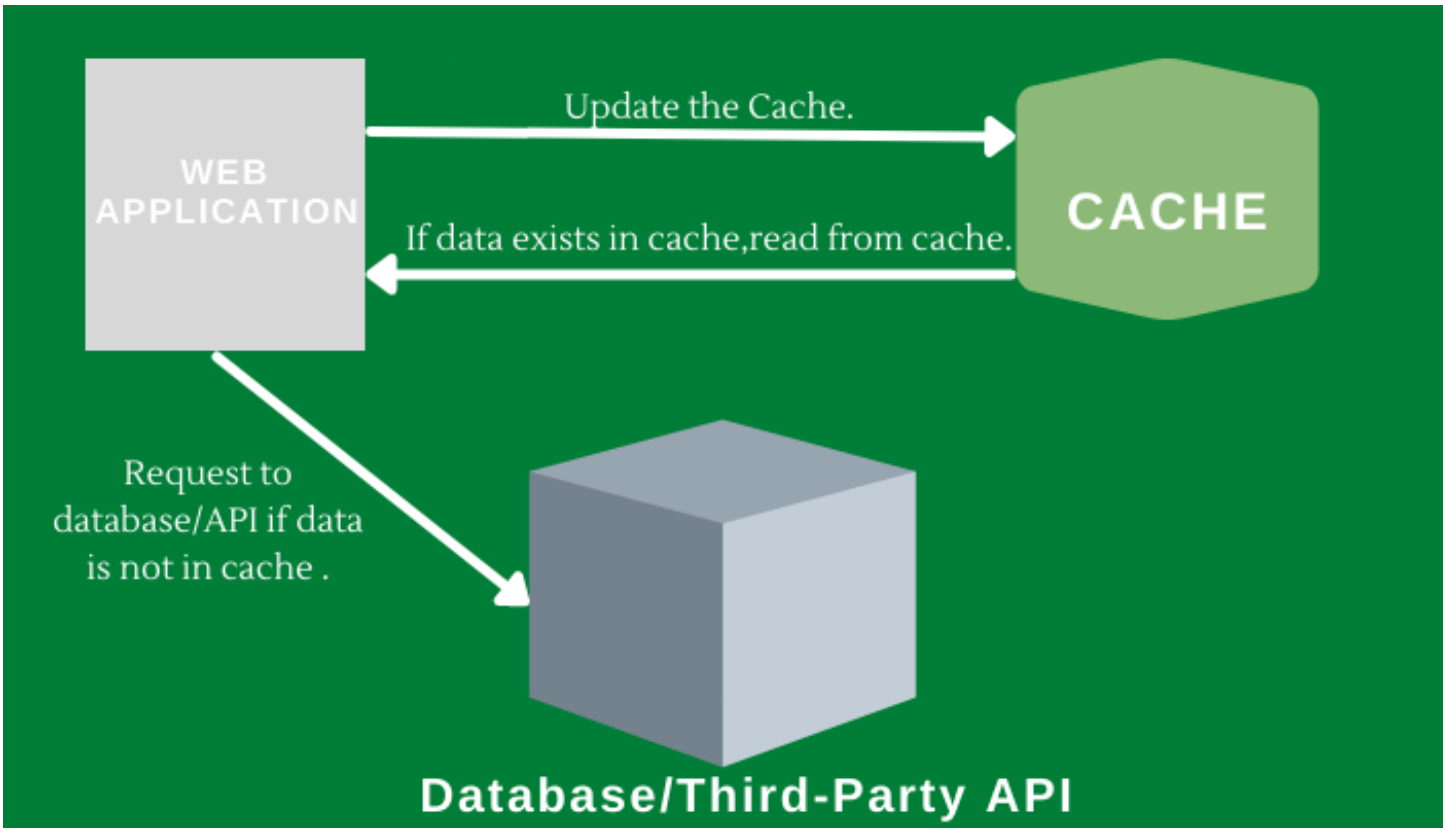
Caching is a leverage play: it accelerates user experience and reduces backend load but introduces complexity around consistency and operations. Treat it as a cross-team concern (apps, infra, SRE) and bake observability and eviction rules into the design.

Spring Boot - Caching

Spring Boot is a project that is built on top of the Spring Framework that provides an easier and faster way to set up, configure, and run both simple and web-based applications. It is one of the popular frameworks among developers these days because of its rapid production-ready environment which enables the developers to directly focus on the logic instead of struggling with the configuration and setup. It is a microservice-based framework used to create a stand-alone Spring-based application that we can run with minimal Spring configurations.

Salient Features

- There is no requirement for heavy **XML** configuration.
- Developing and testing the Spring Boot application is easy as it offers a **CLI** based tool and it also has embedded HTTP servers such as Tomcat, Jetty, etc.
- Since it uses convention over configuration, it increases productivity and reduces development time.



Clarification on Database Caching.

A **Cache** is any temporary storage location that lies between the application and persistence database or a third-party application that stores the most frequently or recently accessed data so that future requests for that data can be served faster. It increases data retrieval performance by reducing the need to access the underlying slower storage layer. Data access from memory is always faster in comparison to fetching data from the database. Caching keeps frequently accessed objects, images, and data closer to where you need them, speeding up access by not hitting the database or any third-party application multiple times for the same data and saving monetary costs. Data that does not change frequently can be cached.

Types of Caching

There are mainly 4 types of Caching :

1. CDN Caching
2. Database Caching
3. In-Memory Caching
4. Web server Caching

1. CDN Caching

A **content delivery network (CDN)** is a group of distributed servers that speed up the delivery of web content by bringing it closer to where users are. Data centers

across the globe use caching, to deliver internet content to a web-enabled device or browser more quickly through a server near you, reducing the load on an application origin and improving the user experience. CDNs cache content like web pages, images, and video in proxy servers near your physical location.

2. Database Caching

Database caching improves scalability by distributing query workload from the backend to multiple front-end systems. It allows flexibility in the processing of data. It can significantly reduce latency and increase throughput for read-heavy application workloads by avoiding, querying a database too much.

3. In-Memory Caching

In-Memory Caching increases the performance of the application. An in-memory cache is a common query store, therefore, relieves databases of reading workloads. Redis cache is one of the examples of an in-memory cache. Redis is distributed, and advanced caching tool that allows backup and restores facilities. In-memory Cache provides query functionality on top of caching.

4. Web server Caching

Web server caching stores data, such as a copy of a web page served by a web server. It is cached or stored the first time a user visits the page and when the next time a user requests the same page, the content will be delivered from the cache, which helps keep the origin server from getting overloaded. It enhances page delivery speed significantly and reduces the work needed to be done by the backend server.

Cache Annotations of Spring Boot

1. @Cacheable

The simplest way to enable caching behavior for a method is to mark it with **@Cacheable** and parameterize it with the name of the cache where the results would be stored.

```
@Cacheable("name")
public String getName(Customer customer) {...}
```

The **getName()** call will first check the cache **name** before actually invoking the method and then caching the result. We can also apply a condition in the annotation by using the condition attribute:

```
@Cacheable(value="Customer", condition="#name.length<10")
public Customer findCustomer(String name)  {...}
```

2. @Cache Evict

Since the cache is small in size. We don't want to populate the cache with values that we don't need often. Caches can grow quite large, quite fast. We can use the **@CacheEvict** annotation to remove values so that fresh values can be loaded into the cache again:

```
@CacheEvict(value="name", allEntries=true)
public String getName(Customer customer) {...}
```

It provides a parameter called **allEntries** that evicts all entries rather than one entry based on the key.

3. @CachePut

@CachePut annotation can update the content of the cache without interfering with the method execution.

```
@CachePut(value="name")
```

```
public String getName(Customer customer) {...}
```

@CachePut selectively updates the entries whenever we alter them to avoid the removal of too much data out of the cache. One of the key differences between **@Cacheable** and **@CachePut** annotation is that the **@Cacheable** skips the method execution while the **@CachePut** runs the method and puts the result into the cache.

4. @Caching

@Caching is used in the case we want to use multiple annotations of the same type on the same method.

```
@CacheEvict("name")
@CacheEvict(value="directory", key="#customer.id")
public String getName(Customer customer) {...}
```

The code written above would fail to compile since **Spring-Boot** does not allow multiple annotations of the same type to be declared for a given method.

```
@Caching(evict = {
 @CacheEvict("name"),
 @CacheEvict(value="directory", key="#customer.id") })
public String getName(Customer customer) {...}
```

As shown in the code above, we can use multiple caching annotations with **@Caching** and avoid compilation errors.

5. @CacheConfig

With @CacheConfig annotation, we can simplify some of the cache configurations into a single place at the class level, so that we don't have to declare things multiple times.

```
@CacheConfig(cacheNames={"name"})
public class CustomerData {
 @Cacheable
  public String getName(Customer customer) {...}
```

Conditional Caching

Sometimes, a method might not be suitable for caching all the time. The cache annotations support such functionality through the **conditional parameter** which takes a **SpEL** expression that is evaluated to either true or false. If true, the method is cached else it behaves as if the method is not cached, which is executed every time no matter what values are in the cache or what arguments are used.

```
@Cacheable(value="name", condition="#customer.name ==  'Megan' ")
public String getName(Customer customer) {...}
```

The above method will be cached, only if the argument name equals Megan.

Unless Parameter

We can also control caching based on the output of the method rather than the input via the **unless** parameter:

```
@Cacheable(value="name", unless="#result.length() < 20")
public String getName(Customer customer) {....}
```

The above annotation would cache addresses unless they were shorter than 20 characters. It's important to know the condition and unless parameters can be used in conjunction with all the caching annotations.

Cache Dependency

If we want to enable a cache mechanism in a **Spring Boot** application, we need to add cache dependency in the **pom.xml** file. It enables caching and configures a **CacheManager**.

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