Spring Boot Caching

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 :

- It make data access faster and less expensive
- It improves the performance of the application.
- It gives responses quickly.
- Data access from memory is always faster than fetching from database.
- 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 :

- In-memory Caching.
- Database Caching.
- Web Server Caching.

- 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

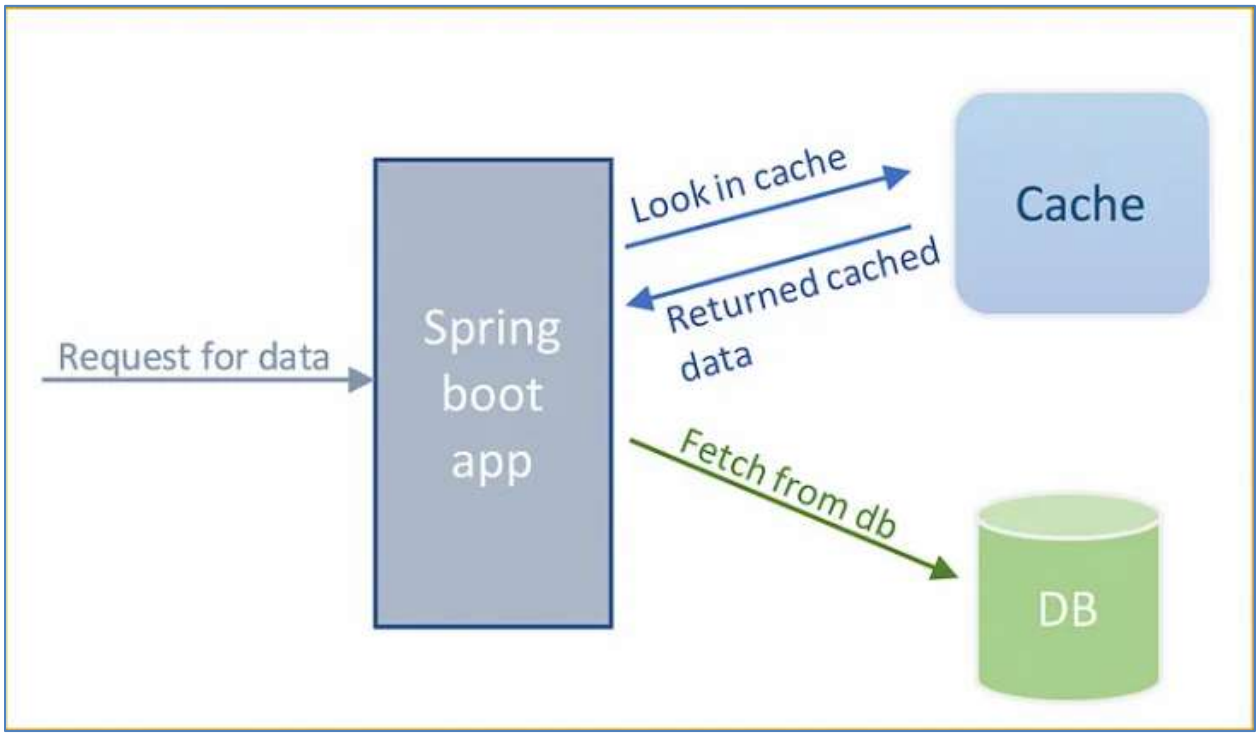
The CDN stands for Content Delivery Network. It is a component used in modern web application.

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.



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**

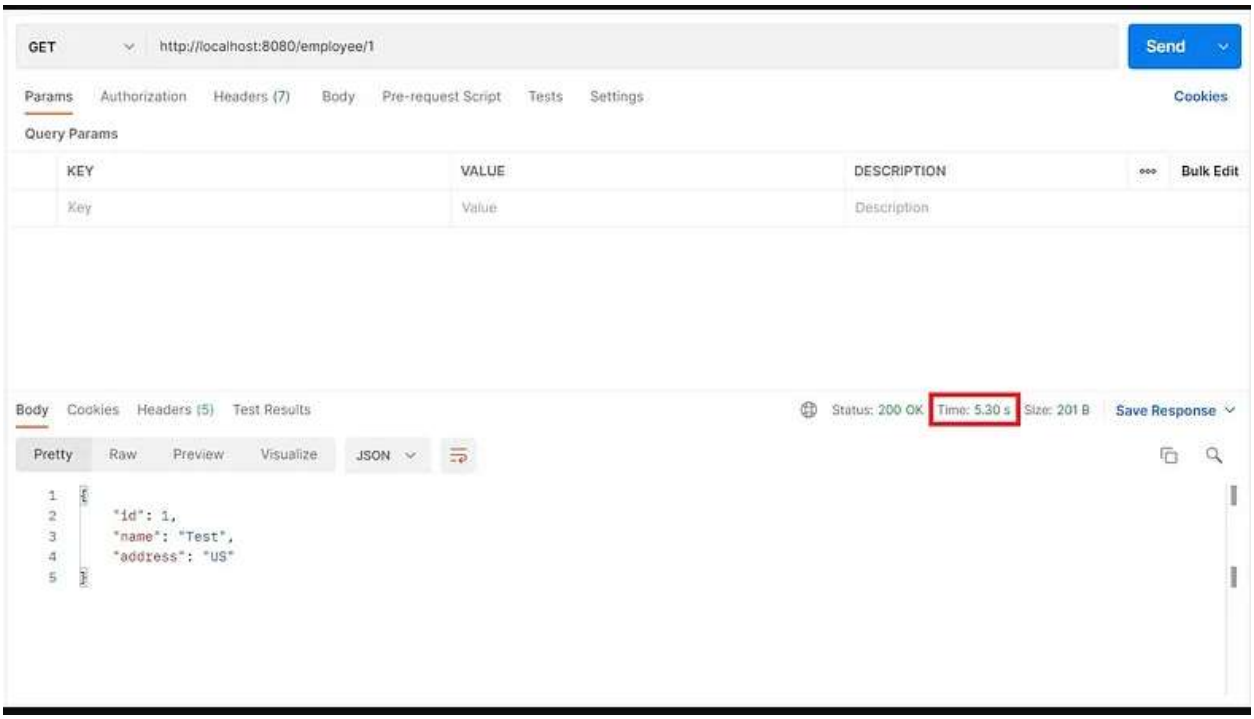
```
package com.example.controller;

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

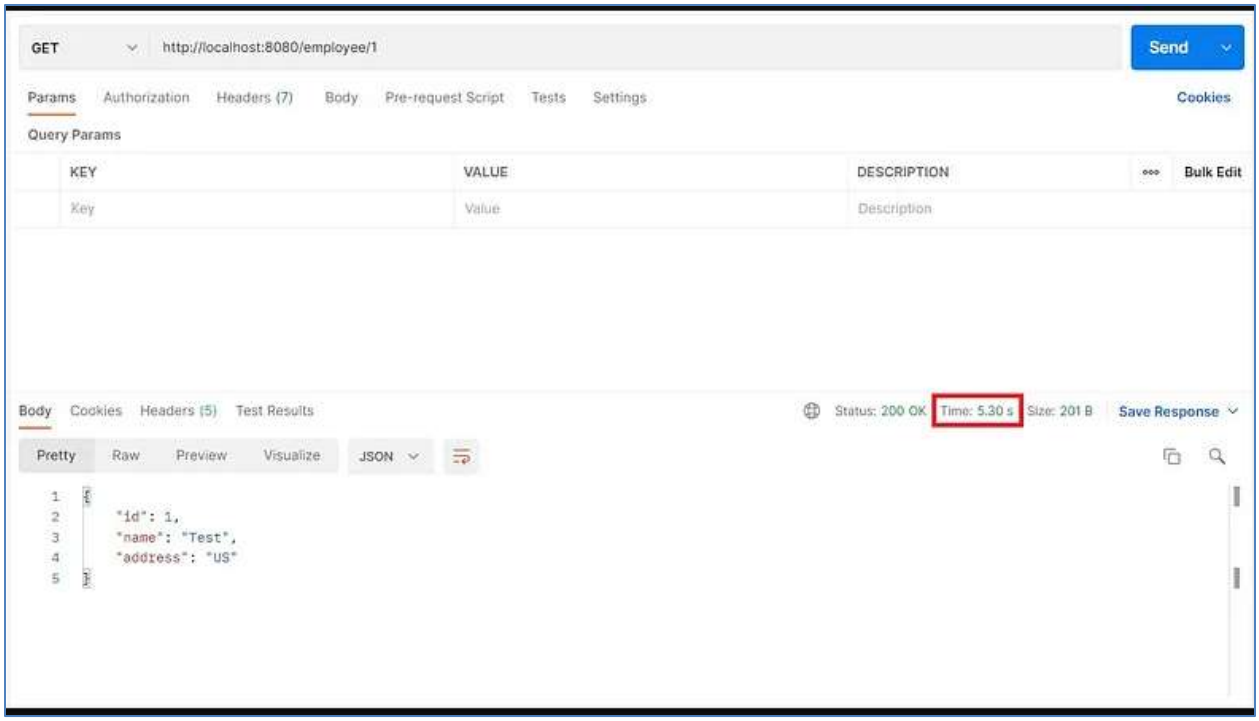
Now, we are going to run the application. Our application is running on port **8080**.

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



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.


GET
http://localhost:8080/employee/1
Send
Params
Authorization
Headers (7)
Body
Pre-request Script
Tests
Settings
Cookies
Query Params
KEY
VALUE
DESCRIPTION
Bulk Edit
Key
Value
Description
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 5.30 s
Size: 201 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"id": 1,
"name": "Test",
"address": "US"
}