

Spring Boot Caching with Redis



Aesha Shingala · Follow

Published in Simform Engineering · 8 min read · Aug 4, 2023



763



1



Implementing Caching in Spring Boot Application Using Redis



In today's fast-paced digital environment, where application speed is crucial in enhancing user experience, caching becomes a potent method to substantially boost application performance.

In this blog post, we will take a deep dive into the world of caching and explore how to implement Redis cache in a Spring Boot application,

unlocking its full potential for producing significant performance advantages.

. . .

Spring Boot Cache Providers

Cache providers allow us to transparently and clearly configure the cache in the application. Use the following steps to configure any given cache provider:

1. Add the `@EnableCaching` annotation to the configuration file.
2. Add the required cache library to the classpath.
3. Add the cache provider configuration file to the root classpath.

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

1. JCache
2. EhCache
3. Hazelcast
4. Infinispan
5. Couchbase
6. Redis
7. Caffeine
8. Simple

Redis is a NoSQL database, so it doesn't have any tables, rows, or columns. Also, it doesn't allow statements like select, insert, update, or delete. Instead, Redis uses data structures to store data. As a result, it can serve frequently requested items with sub-millisecond response times and allow easy scaling for larger workloads without increasing the cost of a more expensive back-end system.

Redis can also be used with streaming solutions to consume, process, and analyze real-time data with sub-millisecond latency. Hence, we advocate implementing caching in spring boot using Redis.

. . .

What is Redis?

Remote Dictionary Server, aka Redis, an in-memory data store, is one of the many options for implementing caching in Spring Boot applications due to its speed, versatility, and simplicity of use. It is a versatile key-value store that supports several data structures, such as Strings, Sorted Sets, Hashes, Lists, Streams, Bitmaps, Sets, etc., because it is a NoSQL database and doesn't need a predetermined schema.

Redis can be used in various ways, including:

1) In-Memory Database: In today's data-driven world, handling vast amounts of real-time data is a common challenge for businesses. A real-time database is a type of data repository designed to acquire, analyze, and/or augment an incoming stream of data points in real time, often immediately after the data is produced. Redis may be used to build data infrastructure for real-time applications that need high throughput and low latency.

2) Cache: Many applications struggle with the need to store and retrieve data quickly, especially in systems with high latency. Due to its speed, Redis is the ideal choice for caching API calls, session states, complex computations, and database queries.

3) Message Broker (MQ): It has always been difficult to stream data around the organization and make it accessible for various system components. Redis supports messaging, event sources, alerts, and high-speed data intake using its stream data type.

How Does Redis Caching Work?

Redis Cache effectively stores the results of database retrieval operations, allowing subsequent requests to retrieve the data directly from the cache. This significantly enhances application performance by reducing unnecessary database calls.



How caching works

When a request is made, the service initially looks in the Redis cache for the desired data. When a cache hit occurs, the data is swiftly retrieved from the cache and promptly provided back to the service, avoiding the need to interact with the database.

However, if the requested data is not found in the cache (cache miss), the service intelligently falls back to the database to retrieve the required information. Subsequently, the fetched data is stored in the Redis cache, enabling future requests for the same data to be served directly from the cache, thereby eliminating further database queries and speeding up overall response times.

Redis can also be used for deleting and updating tasks, guaranteeing consistent and latest data in the cache and further boosting overall efficiency.

. . .

Let's Set up Our Spring Boot Application!

Step 1: Add Redis dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
  <version>3.1.2</version>
</dependency>
```

Redis-related libraries are part of the Spring Data Package, which provides easy access to databases, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. The `spring-boot-starter-data-redis` will have all the necessary dependencies prepared.

Step 2: Configure Redis using application.yml

```
spring:
  cache:
    type: redis
    host: localhost
    port: 6379
    redis:
      time-to-live: 60000
  datasource:
    driver-class-name: org.postgresql.Driver
    url: jdbc:postgresql://localhost:5432/inventory
    username: root
    password: ****
  jpa:
    hibernate:
      ddl-auto: update
    properties:
      hibernate:
        dialect: org.hibernate.dialect.PostgreSQLDialect
```

The PostgreSQL database is used, and the necessary database connection properties are introduced. In addition, we define Redis as our cache provider, along with its hostname and port.

Time To Live(TTL): A good practice for caching is to ensure that excess and redundant data is not accumulated indefinitely, as this can result in stale or outdated data being served to users. To serve this, we can take advantage of the time-to-live property, which is an optional setting that allows us to set the expiration time for cached data. After the specified time has elapsed, the cached entry is automatically removed from the cache. This makes space for new data to be fetched and stored in the cache the next time it's requested. If no value is assigned to the property, it becomes -1 by default, which means the data will stay in the cache indefinitely.

We have set our time to live property to be 60000 ms in the example, which means that the data will be cleared from the cache after every minute.

Step 3: Creating entity and repository

Set up the entity and repository to perform CRUD operations like any other Spring Boot project. Since Redis is an in-memory database, we need to transform our object into a stream of bytes for storing as well as the other way around for retrieving data. Therefore, we need to serialize/deserialize our data by ensuring that the entity class implements the Serializable class.

```
@Entity
public class Product implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;
    private String name;
    private String code;
    private int quantity;
    private double price;
}
```

We have an entity responsible for storing and fetching data, so we can create our repository by extending the JpaRepository interface. It will provide basic CRUD operations to work with, like save, find, and delete.

```
@Repository
public interface ProductRepository extends JpaRepository<Product, Long> {
}
```

Step 4: Enable caching in spring boot

Add the annotation `@EnableCaching` to the starter class. It will trigger a post-processor that inspects every Spring bean for the presence of caching annotations on public methods.

```
@SpringBootApplication
@EnableCaching
public class RedisDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(RedisDemoApplication.class, args);
    }
}
```

Step 5: Annotation-based caching on the controller layer

Use caching annotations in the controller layer to enable caching for specific methods:

1. `@Cacheable` is employed to fetch data from the database, storing it in the cache. Upon future invocations, the method retrieves the cached value directly, eliminating the need to execute the method again.

```
@GetMapping("/product/{id}")
@Cacheable(value = "product", key = "#id")
public Product getProductById(@PathVariable long id) {...}
```

The `value` attribute establishes a cache with a specific name, while the `key` attribute permits the use of Spring Expression Language to compute the key dynamically. Consequently, the method result is stored in the 'product'

cache, where respective 'product_id' serves as the unique key. This approach optimizes caching by associating each result with a distinct key.

2. **@CachePut** is used to update data in the cache when there is any update in the source database.

```
@PutMapping("/product/{id}")
@CachePut(cacheNames = "product", key = "#id")
public Product editProduct(@PathVariable long id, @RequestBody Product product)
```

The **cacheNames** attribute is an alias for **value**, and can be used in a similar manner.

3. **@CacheEvict** is used for removing stale or unused data from the cache.

```
@DeleteMapping("/product/{id}")
@CacheEvict(cacheNames = "product", key = "#id", beforeInvocation = true)
public String removeProductById(@PathVariable long id) {...}
```

We use **cacheName** and **key** to remove specific data from the cache. The **beforeInvocation** attribute allows us to control the eviction process, enabling us to choose whether the eviction should occur before or after the method execution.

```
@DeleteMapping("/product/{id}")
@CacheEvict(cacheNames = "product", allEntries = true)
```

```
public String removeProductById(@PathVariable long id) {...}
```

Alternatively, all the data can also be removed for a given cache by using the `allEntries` attribute as true. The annotation allows us to clear data for multiple caches as well by providing multiple values as `cacheName`.

4. `@Caching` is used for multiple nested caching on the same method.

```
@PutMapping("/{id}")
@Caching(
    evict = {@CacheEvict(value = "productList", allEntries = true)},
    put = {@CachePut(value = "product", key = "#id")}
)
public Product editProduct(@PathVariable long id, @RequestBody Product product)
```

As Java doesn't allow multiple annotations of the same type to be declared for a method, doing so will generate a compilation error. We can group different annotations into `Caching`, as shown above.

5. `@CacheConfig` is used for centralized configuration.

```
@CacheConfig(cacheNames = "product")
public class ProductController {...}
```

Used as a class-level annotation, `Cache config` allows centralizing some of the configurations to avoid specifying it at each step. For the example above,

we need not write `cacheName` for all methods, instead, specify the cache name at the class level.

Using `keyGenerator` attribute:

This is responsible for generating every key for each data item in the cache, which would be used to look up the data item on retrieval. The default implementation here is the `SimpleKeyGenerator` — which uses the method parameters provided to generate a key, as shown earlier. However, we may want to provide a more customized approach to generate the cache key based on specific criteria. To achieve this, we can use the `keyGenerator` attribute.

To give a different custom key generator, we need to implement the `org.springframework.cache.interceptor.KeyGenerator` interface. The class needs to implement a single method as shown:

```
public class CustomKeyGenerator implements KeyGenerator {  
    @Override  
    public Object generate(Object target, Method method, Object... params) {  
        return target.getClass().getSimpleName() + "_"  
            + method.getName() + "_"  
            + StringUtils.arrayToDelimitedString(params, "_");  
    }  
}
```

We can now use the `customeKeyGenerator` as a value to `keyGenerator` attribute and provide a customized key generator for the given method.

```
@GetMapping("/product/{id}")
@Cacheable(value = "product", keyGenerator = "customKeyGenerator")
public Product getProductById(@PathVariable long id) {...}
```

Conditional caching:

Conditional caching allows us to cache specific data using the `unless` and `conditional` attributes. The `unless` attribute allows us to filter data after the method has been called. By using the Spring Expression Language, we can choose not to cache the data if a certain condition evaluates to true. In the given example, we have allowed caching only for products priced less than 1000. This way, we cache selectively based on the defined condition.

```
@GetMapping("/product/{id}")
@Cacheable(value = "product", key = "#id" , unless = "#result.price > 1000")
public Product getProductById(@PathVariable long id) {...}
```

The `conditional` attribute, however, works quite the opposite. If the condition happens to be true, then the resulting data is cached in our cache. For the given example, only products named fridge will be stored in the cache.

```
@PutMapping("/{id}")
@Caching( value = "product", condition = "#product.name == 'Fridge'"))
public Product editProduct(@PathVariable long id, @RequestBody Product product)
```

Wrapping Up

In this post, we discussed the basics of using Spring caching in your application. Remember to fine-tune your caching configuration based on your application's specific needs.

Integrating Redis cache into your Spring Boot applications can significantly enhance their performance and scalability. Redis's in-memory storage and blazing-fast read/write operations make it an excellent choice for caching frequently accessed data. By intelligently using caching strategies and leveraging Redis' distributed architecture, you can create responsive and efficient applications that deliver an exceptional user experience.

For more updates on the latest tools and technologies, follow the [Simform Engineering](#) blog.

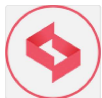
Follow Us: [Twitter](#) | [LinkedIn](#)

Redis

Spring Boot

Caching

Cache

**Published in Simform Engineering**

1.1K Followers · Last published 2 days ago

Our Engineering blog gives an inside look at our technologies from the perspective of our engineers.

[Follow](#)**Written by Aesha Shingala**

70 Followers · 9 Following

[Follow](#)

Responses (1)



What are your thoughts?

Respond



S M Junayed Shanto

7 months ago



I don't know why. But this is not working :|



50



1 reply

Reply

More from Aesha Shingala and Simform Engineering



Unlocking Graph Power: A Journey with Spring Data Neo4j

In Simform Engineering by Aesha Shingala

Unlocking Graph Power: A Journey with Spring Data Neo4j

Building scalable solutions with Spring Boot and Neo4j

May 28 497 2



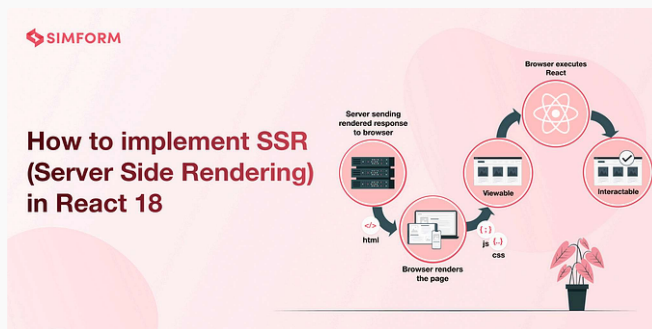
The Ultimate Guide to Writing Clean Jetpack Compose

In Simform Engineering by Meetmistry

The Ultimate Guide to Writing Clean Jetpack Compose

A Step-by-Step Approach to Crafting Elegant and Maintainable Compose

Nov 21 1.4K 3



How to implement SSR (Server Side Rendering) in React 18

In Simform Engineering by Jay Sheth

How to implement SSR(Server Side Rendering) in React 18

Learn how to implement the “renderToPipeableStream” server API to...

Sep 15, 2023 1.1K 6



Monitoring Made Simple: Empowering Spring Boot Applications with Prometheus and Grafana

In Simform Engineering by Arshit Moradiya

Monitoring Made Simple: Empowering Spring Boot...

Learn how to configure and use Prometheus and Grafana to collect metrics from your...

Jul 6, 2023 409 6




See all from Aesha Shingala

See all from Simform Engineering

Recommended from Medium




 Jobin J

Redis Caching with Java using Jedis

Redis caching is a technique that uses Redis, an open-source, in-memory data structure...

Aug 27



 In Stackademic by Mpavani

Microservices tricky interview questions must know

1.You have two microservices, Order Service and Payment Service. When an order is...



Oct 30



478



8



Lists



Staff picks

780 stories · 1492 saves



Stories to Help You Level-Up at Work

19 stories · 891 saves



Self-Improvement 101

20 stories · 3126 saves



Productivity 101

20 stories · 2641 saves



 Priya Upmaka

Implementing a Two-Level Cache with Spring Boot

Introduction

★ Jun 14 🖱 122 💬 2

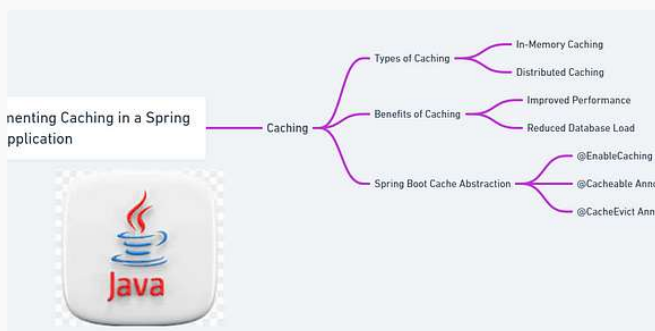



 Technical Life

Solving Redis Connection Issues in a Spring Boot Application

Troubleshooted Redis connection issues in Spring Boot

★ Jun 24 🖱 2



 Sanjay Singh

Implementing Caching in a Spring Boot Application: A Complete...

Overview

★ Oct 11 🖱 42



 Tharindu Dulshan

Optimizing Spring Boot Applications with Redis Caching

In my previous blog, I explained what is Caching and different methods of caching...

Sep 20 🖱 52 💬 1



See more recommendations