Singleton Design Pattern

What is Singleton?

The singleton pattern is one of the Gang of Four creational design patterns. In software engineering, this is a term that implies a class which can **only be instantiated once,**and a global point of access to that instance is provided. In the Java programming language, there are a few standard implementations of the singleton pattern. Here are some common ways of implementing a singleton.

public class SingletonClass {

private static SingletonClass instance = null;

private SingletonClass() {

}

public static SingletonClass getInstance() {

if (instance == null) {

instance = new SingletonClass();

}

return instance;

}

}

In the example above, we wrote a class with a method that creates a new instance of the class if one does not exist. If an instance has already been created, the constructor simply returns a reference to that object. This is the simplest implementation of the Singleton design pattern in Java.

**Note** that the constructor of the

SingletonClass

 class is made private, to make sure that there is no other way to instantiate the class. This example is known as **lazy initialization** – which means that it restricts the creation of the instance until it is requested for the first time.

There are times you may wish to eager initialization over lazy initialization. This will impact your startup time performance, but it may help improve the runtime performance of your application.  Here is an example of **eager initialization**:

public class SingletonClass {

private static final SingletonClass INSTANCE = new SingletonClass();

private SingletonClass() {}

public static SingletonClass getInstance() {

return INSTANCE;

}

}

This approach is similar to the lazy implementations, with the difference that this one **always creates an instance**, even if one is not currently necessary in the program. This might be considered a bad practice, but it’s suitable for situations where the creation of the instance is not too expensive or demanding. This is useful when the cost of creating the object is not significant or when you know your application will always instantiate the class.

**Note** that the

final

 keyword here enables the class to actually be a singleton, i.e. ensures that only one instance exists.

Advanced Examples

Thread Safe Singletons

The previous examples will both work just fine in a single-threaded environment. However, what happens if the program is multi-threaded? If we have two threads, and both of them call the

getInstance()

 method, it is possible that two individual objects of the same class get created, due to different times of accessing the

(instance==null)

 check. This is why we need to modify our implementation to make it thread-safe, and the simplest way of achieving this is by making the

getInstance()

 class **synchronized**.  
This will make it impossible for one thread to access the method if another thread hasn’t finished with its execution.

**Note**, while using synchronized will ensure thread safety, be aware there is a performance cost overhead.

Why and when to use the Singleton Design Pattern?

There are situations in which a singleton class is useful. Typical examples are classes that we could shortly describe as “managers”. These are classes which manage resources, read data from configuration files, instantiate threads etc. If you were making a role-playing game, you would probably have multiple “Item” objects or multiple “Level” objects, but it would make sense to make the “Score” class singleton – since you only need one object to take care of the player’s score.

Another example could be a factory with multiple manufacturing tracks. The tracks are the ones that actually do all the work, whether it is packing cans or making car parts. However, it is the “factory manager” that decides how many tracks will be working, for how long, and what they are going to do.

public class FactoryManager

{

private static FactoryManager instance = null;

private FactoryManager() {}

public static FactoryManager getInstance()

{

if (instance == null)

{

instance = new FactoryManager();

}

return instance;

}

private int numTracks = 5;

public static void startTracks()

{

for(int i=0;i<numTracks;++i)

{

Track t = new Track();

t.start();

}

}

}

Then, in some other part of the code, we can only “instantiate” an object of the

FactoryManager

 class by calling the

getInstance

 method.

public static void main(String[] args){

FactoryManager fm = FactoryManager.getInstance();

fm.startTracks();

}

The Singleton pattern is automatically implemented for Beans registered in the application context. By default, Beans in Spring Boot have a singleton scope, meaning that only one instance of the Bean is created and managed by the Spring container.

**Singleton Scope for Beans (per container per bean)**

Spring beans are often configured as singletons by default, ensuring that only one instance of a bean exists per container. This promotes efficient resource usage and consistent state management.

The latest version of the Spring framework defines six types of scopes - singleton, prototype, request, session, application, and websocket. The scope of bean defines its lifecycle and visibility.

@Configuration // A class marked @Configuration means it is the source of bean definitions.public class MyBeanConfig

{

@Beanpublic SingletonBean MyBean()

{

return new MyBean(); // the object returned here is singleton by default

}

}

*Spring’s concept of a singleton bean differs from the singleton pattern as defined in the Gang of Four (GoF) patterns book. The GoF singleton hard-codes the scope of an object such that one and only one instance of a particular class is created per ClassLoader. The scope of the Spring singleton is best described as being per-container and per-bean. This means that, if you define one bean for a particular class in a single Spring container, the Spring container creates one and only one instance of the class defined by that bean definition.*

In the world of software design patterns, the **Singleton** pattern stands out as one of the most widely used patterns. It’s particularly valuable when you want to ensure that only **one instance** of a class is created throughout the application. When it comes to building scalable and efficient enterprise applications with Spring Boot, understanding the Singleton pattern and how Spring handles singleton objects is crucial.

In this blog, we’ll dive deep into what a singleton is, how Spring Boot implements it, and its practical uses in a real-world Spring application.

**What is the Singleton Pattern?**

The Singleton Design Pattern restricts the instantiation of a class to a single “global” instance. This means that whenever you request an object of this class, the system provides the same instance, ensuring consistency and saving resources.

Key characteristics:

* **One Instance**: A single instance for the entire application lifecycle.
* **Global Access**: The same instance can be accessed globally across the application.
* **Efficient Resource Usage**: Eliminates the overhead of creating multiple instances, especially useful for expensive-to-create objects.

**Singleton in Spring Boot**

In Spring Boot, beans are singleton by **default**. Whenever you define a Spring bean using annotations such as @Component, @Service, or @Repository, or register it in a @Configuration class, Spring ensures that only one instance of that bean is created in the application context. This makes Spring's singleton implementation aligned with the design pattern's principles.

**How to Create Singleton Beans in Spring Boot**

1. **Default Singleton Scope**: When you define a bean, Spring automatically treats it as a singleton unless otherwise specified.

Plain Text

import org.springframework.stereotype.Service;  
  
@Service  
public class MySingletonService {  
    public void performTask() {  
        System.out.println("Task performed by Singleton bean.");  
    }  
}

In this example, MySingletonService will have only **one instance** in the Spring container, shared across the entire application.

1. **Explicit Singleton Declaration**: If you want to explicitly define the scope of a bean as singleton, you can use the @Scope annotation.

Plain Text

import org.springframework.context.annotation.Scope;  
import org.springframework.stereotype.Component;  
  
@Component  
@Scope("singleton")  
public class ExplicitSingletonService {  
    public void executeTask() {  
        System.out.println("Task executed by explicitly defined Singleton bean.");  
    }  
}

The @Scope("singleton") annotation is redundant in most cases, as singleton is the default scope, but it can be useful for clarity in large applications.

**Key Benefits of Singleton Beans in Spring**

1. **Memory Efficiency**: Since Spring manages a single instance of the bean, it avoids the cost of repeatedly creating and destroying objects, making it memory efficient.
2. **Global Consistency**: All consumers of the singleton bean share the same instance, ensuring consistent behavior throughout the application.
3. **Thread Safety (With Care)**: Singleton beans are created once and shared, but you must be careful about shared mutable state. Spring itself does not make singleton beans inherently thread-safe.

**Common Use Cases for Singleton Beans in Spring Boot**

1. **Service Layer Beans**: Most service-level beans, such as those annotated with @Service, are often singleton. These beans handle business logic and do not typically maintain any state, making them perfect candidates for single-instance creation.

Plain Text

@Service  
public class UserService {  
    public User getUserById(Long id) {  
        // Business logic here  
        return userRepository.findById(id).orElse(null);  
    }  
}

1. The UserService bean will be a singleton, shared across all controllers or services that need to access user-related business logic.
2. **DAO/Repository Beans**: Data Access Objects (DAOs) and repository beans, like those annotated with @Repository, are singleton as well. Given that these beans typically interact with the database, having a single instance ensures optimal database connection usage.

Plain Text

@Repository  
public interface UserRepository extends JpaRepository<User, Long> {  
    // Data access methods  
}

1. **Configuration Beans**: Beans that store configuration details (e.g., property files, settings) are also common singleton candidates. Since application settings typically do not change after the application has started, there’s no need to create multiple instances.

Plain Text

@Configuration  
public class AppConfig {  
    @Bean  
    public MyConfigBean myConfigBean() {  
        return new MyConfigBean();  
    }  
}

1. **Utility Classes**: Utility classes such as email services, logging mechanisms, and caching layers are also well-suited for the singleton pattern. These services often have no mutable state and are safe to share across the application.

**Important Considerations for Singleton Beans**

* **Thread Safety**: Singleton beans are shared across multiple threads. If your bean holds any state (i.e., has instance variables), ensure that it is thread-safe. Spring does not enforce thread safety on singleton beans, so you need to implement mechanisms like synchronization if required.
* **Mutable State**: It’s generally advisable not to store mutable state in singleton beans, as changes to the state by one consumer will affect all other consumers. If state is necessary, consider using scopes like @Scope("prototype") or managing state within individual requests.

Scalability and Performance: Redis excels in handling large-scale data and can be easily distributed across multiple servers, making it an ideal choice for managing singletons in distributed systems. With Redis, I was able to handle an ever-increasing user base without compromising on performance.

// Node.js Redis client setup  
const redis = require('redis');  
const redisClient = redis.createClient();  
  
// Implementing a Singleton class using Redis  
class RedisSingleton {  
 constructor() {  
 if (!RedisSingleton.instance) {  
 RedisSingleton.instance = this;  
 }  
  
 return RedisSingleton.instance;  
 }  
  
 // Sample method to set data in Redis  
 setData(key, value) {  
 redisClient.set(key, value, (err) => {  
 if (err) {  
 console.error('Error setting data in Redis:', err);  
 }  
 });  
 }  
  
 // Sample method to get data from Redis  
 getData(key, callback) {  
 redisClient.get(key, (err, result) => {  
 if (err) {  
 console.error('Error getting data from Redis:', err);  
 return callback(err, null);  
 }  
  
 return callback(null, result);  
 });  
 }  
}  
  
// Creating an instance of the RedisSingleton class  
const redisSingletonInstance = new RedisSingleton();  
  
// Exporting the singleton instance  
module.exports = redisSingletonInstance;

Shared Resource Management: In many applications, there are resources that should be shared across different components. For instance, database connections, configurations, and caching mechanisms are prime examples. Redis Singleton enabled me to effectively manage these shared resources without creating multiple instances or facing resource contention issues.

// Sample code for using the RedisSingleton class to manage shared configurations  
  
// Importing the Redis Singleton instance  
const redisSingleton = require('./redisSingleton');  
  
// Method to fetch a configuration value from Redis  
function getConfiguration(key, callback) {  
 redisSingleton.getData(key, (err, value) => {  
 if (err) {  
 console.error('Error fetching configuration:', err);  
 return callback(err, null);  
 }  
  
 return callback(null, value);  
 });  
}  
  
// Usage example  
getConfiguration('api\_key', (err, apiKey) => {  
 if (err) {  
 console.error('Failed to fetch API key.');  
 return;  
 }  
  
 console.log('API Key:', apiKey);  
});

Synchronization and Concurrency: Implementing the Singleton pattern is crucial when dealing with concurrent access to shared resources. Redis provides atomic operations and supports various data structures, which allowed me to implement thread-safe Singleton classes seamlessly.

// Sample code for handling concurrency using Redis  
  
// Importing the Redis Singleton instance  
const redisSingleton = require('./redisSingleton');  
  
// Method to increment a counter in Redis atomically  
function incrementCounter(key) {  
 redisClient.incr(key, (err, newCounterValue) => {  
 if (err) {  
 console.error('Error incrementing counter:', err);  
 return;  
 }  
  
 console.log('New counter value:', newCounterValue);  
 });  
}  
  
// Usage example  
incrementCounter('page\_views');

**Bill Pugh Singleton (Initialization-on-demand holder idiom):**

**Double-Checked Locking (DCL):**

* This pattern attempts to reduce synchronization overhead by performing a check for the instance's existence before acquiring a lock. If the instance is null, a lock is acquired, and the check is performed again inside the synchronized block.
* **Requirement:** The instance variable must be declared volatile to ensure visibility of changes across threads and prevent reordering of instructions by the compiler/JIT.
* **Advantage:** Reduces synchronization overhead once the instance is initialized.

Synchronized getInstance() Method:

* Synchronizing the entire getInstance() method ensures that only one thread can access it at a time, preventing multiple instance creation.
* **Disadvantage:** This introduces performance overhead, as subsequent calls to getInstance() will still incur the synchronization cost even after the instance has been created.

**2.2. Lazy Initialization**

Singleton pattern implementations often use **lazy initialization to delay instance creation until the first time it is actually needed**. To ensure lazy instantiation, we can create an instance when the static getter is first invoked:

**public** **final** **class** **ThreadSafeSingleInstance** {

**private** **static** **volatile** **ThreadSafeSingleInstance** instance = null;

**private** **ThreadSafeSingleInstance**() {}

**public** **static** ThreadSafeSingleInstance **getInstance**() {

**if** (instance == null) {

**synchronized**(ThreadSafeSingleInstance.class) {

**if** (instance == null) {

instance = **new** **ThreadSafeSingleInstance**();

}

}

}

**return** instance;

}

//standard getters

}Copy

In multithreaded applications, lazy instantiation can cause [race conditions](https://www.baeldung.com/java-common-concurrency-pitfalls). Therefore, we’ve also applied [double-checked locking](https://www.baeldung.com/java-singleton-double-checked-locking) to prevent the creation of multiple instances by different threads.

**3. Singleton Beans in Spring**

A [bean](https://www.baeldung.com/spring-bean) in the Spring framework is an **object created, managed, and destroyed in the**[**Spring IoC Container**](https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring).

**3.1. Bean Scope**

With Spring beans, we can inject an object into the Spring Container through metadata using [inversion of control](https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring) (IoC). In effect, an **object can define its dependencies without creating them and delegate that work to the IoC Container**.

The latest version of the Spring framework defines [six types of scopes:](https://www.baeldung.com/spring-bean-scopes)

* singleton
* prototype
* request
* session
* application
* websocket

The scope of a bean defines its life cycle and visibility. It also determines how actual instances of a bean will be created. For example, we may want to create a single global instance or a different instance every time a bean is requested.

**3.2. Singleton Beans**

We can declare beans in Spring using the [*@Bean* annotation](https://www.baeldung.com/spring-bean-annotations) located in a configuration class. The singleton scope in Spring creates **one bean per bean identifier in a container**:

@Configuration

**public** **class** **SingletonBeanConfig** {

@Bean

@Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)

**public** SingletonBean **singletonBean**() {

**return** **new** **SingletonBean**();

}

}Copy

Singleton is the default scope of all beans defined in Spring. So even if we didn’t specify a specific scope using the *@Scope* annotation, we’d still get a singleton bean. The scope is included here for illustration purposes only. It would normally be used for expressing the other scopes available.

**3.3. Bean Identifier**

Unlike the pure singleton design pattern, we can **create multiple singleton beans from the same class**:

@Bean

@Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)

**public** SingletonBean **singletonBean**() {

**return** **new** **SingletonBean**();

}

@Bean

@Scope(value = ConfigurableBeanFactory.SCOPE\_SINGLETON)

**public** SingletonBean **anotherSingletonBean**() {

**return** **new** **SingletonBean**();

}Copy

All requests for beans with a matching identifier will result in one specific bean instance being returned by the framework. When we use the *@Bean* annotation on a method, Spring uses the method name as a [bean identifier](https://www.baeldung.com/spring-bean-names).

When injecting beans, if more than one bean of the same type is available in the container, the framework throws  *NoUniqueBeanDefinitionException*:

@Autowired

**private** SingletonBeanConfig.SingletonBean bean; //throws exceptionCopy

In that case, we can make use of the [*@Qualifier* annotation](https://www.baeldung.com/spring-qualifier-annotation) to specify the correct bean identifier to inject:

@Autowired

@Qualifier("singletonBean")

**private** SingletonBeanConfig.SingletonBean beanOne;

@Autowired

@Qualifier("anotherSingletonBean")

**private** SingletonBeanConfig.SingletonBean beanThree;Copy

Alternatively, another annotation – [*@Primary*](https://www.baeldung.com/spring-primary) – can be used to define a primary bean when multiple beans of the same type are present.

**4. Comparison**

Let’s now compare the two approaches and identify a best practice to follow in Spring.

**4.1. Singleton Anti-Pattern**

Some consider singleton to be an **anti-pattern because it introduces an application-level global state.** Any other object using the singleton has a direct dependency on it. That results in unnecessary inter-dependencies between classes and modules.

The Singleton pattern also violates the single-responsibility principle. As singleton objects are responsible for at least two things:

* Ensuring only one instance is created
* Performing their normal operations

In addition, singletons require special treatment in multi-threaded environments to ensure that separate threads don’t create multiple instances. They might also make unit testing and mocking harder. As many mocking frameworks rely on inheritance, the private constructor makes singleton objects hard to mock.

**4.2. Recommended Approach**

Using Spring’s singleton beans instead of implementing the singleton design pattern eliminates many of the above disadvantages.

**Spring framework** **injects a bean in all classes that use it, but retains the flexibility to replace or extend it**. The framework achieves that by maintaining control over the bean’s lifecycle. Therefore, it can later be replaced by another approach without any code having to change.

In addition, Spring beans make unit testing much simpler. Spring beans are easy to mock and the framework can inject them into test classes. We may choose to inject actual bean implementations or their mocks.

We should note that singleton beans will not create only one instance of a class, but one bean per bean identifier in a container.