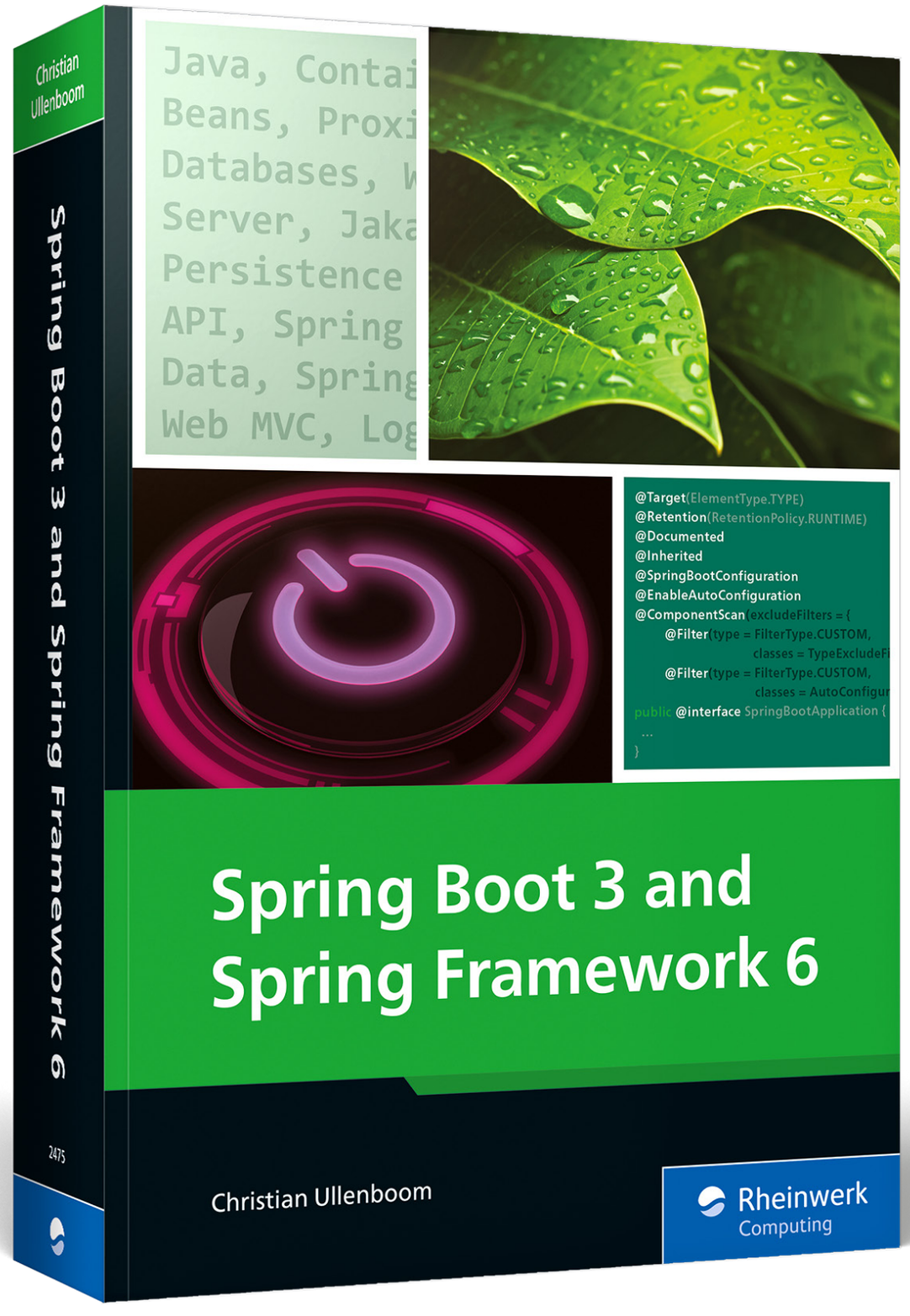


Reading Sample

This sample chapter delves into the importance of proxies in Spring, with a brief touch on Jakarta Bean Validation. Key topics covered include proxy patterns as well as caching, discussing how it can be implemented in Spring applications and the benefits it provides. The chapter also walks through using Spring proxies for concurrent programming (via asynchronous calls) and how to implement them in your projects.

Christian Ullenboom

Spring Boot 3 and Spring Framework 6

934 pages | 10/2023 | $59.95 | ISBN 978-1-4932-2475-3

www.rheinwerk-computing.com/5764

Chapter 4
Selected Proxies

Injection plays an important role in Spring applications. With the help of injection, objects are brought to the places where they are needed. However, the client may not get a reference to the desired object itself; instead, Spring sets up a proxy. A proxy can be thought of as a ring that wraps around an object, intercepts all method calls, and forwards them if necessary.

On the way to the target object in the core, there could be various checks, which could also go so far as to prevent access.

4.1 Proxy Pattern

What can be vividly explained using rings, we want to translate into the world of software engineering with a sequence diagram. The ring illustrates the *proxy pattern*, which is a well-known design pattern.

In the core shown in Figure 4.1 is the *target object*. This target object has an interface, let's call it `I`, with operations. The *proxy object* (aka substitute) wraps around this target object. It has the same interface `I` to the outside as the target object, which is also called

the *subject*. If a client has a reference to "something of type `I`," the client doesn't know whether it's talking to the proxy or to the target object.

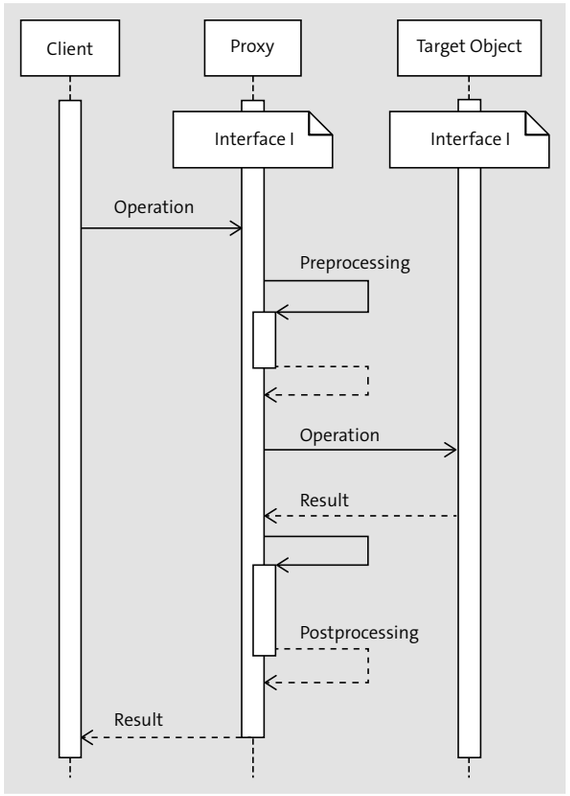


Figure 4.1 Sequence Diagram of the Proxy Pattern

A proxy doesn't sound exciting, but it can do two useful things:

- Before the proxy goes to the target object with the operation, it can perform preprocessing. That is, the proxy can call internal methods and modify the parameters. Then, the proxy can go to the target object or block the call.
- When the target object returns the result to the proxy, the proxy can perform postprocessing. For example, it can cache the data. Later, the proxy returns the result to the client.

Note

With the corresponding `Class` object, you can find out whether you have a proxy object in front of you or not. The `toString()` method also responds differently. Spring includes in `org.springframework.aop.support.AopUtils` the helper method `isAopProxy(…)` for Spring proxies:


```
public static boolean isAopProxy(@Nullable Object object) {
  return
    (object instanceof SpringProxy &&
        (Proxy.isProxyClass(object.getClass()) ||
    object.getClass().getName().contains(
    ClassUtils.CGLIB_CLASS_SEPARATOR)));
}
```

When implementing `equals(…)`, it must be taken into account in places that comparisons are programmed with `instanceof` instead of `getClass(…)`.

4.1.1 Proxy Deployment in Spring

In general, a proxy is limited in its abilities and is incapable of modifying the behavior of the target operation. However, the preprocessing and post-processing capabilities of a proxy are frequently sufficient for a variety of use cases. Consequently, Spring employs proxies in numerous contexts. On several occasions, the code is injected with a proxy, which often goes unnoticed because the proxy has the same interface as the target object.

Following are a few examples of how Spring uses proxies:

- **Transaction management**
  Methods can initiate new transactions, and everything the method does to the database takes place in a transactional context. If there are exceptions in the method, the proxy catches those exceptions and starts a rollback; if there are no errors, then a commit follows at the end of the method.
- **Caching**
  If a client calls a method with a certain argument, a proxy can determine that the method has seen the argument before and can return the result computed earlier.
- **Validation**
  The purpose of validation is to ensure that methods are called only with valid values. A proxy can check that the arguments are in the correct value ranges and pass them to the target object only if they are valid.
- **Asynchronous calls**
  Normally, a method is called synchronously (blocking). A special proxy can start a background thread—or fall back to a thread of a thread pool—and then asynchronously process this operation.
- **Spring Retry project**
  Aborts can occur—especially with remote accesses; the Spring Retry project can use a proxy that starts another attempt if a target object's operation fails, until the call finally succeeds or you give up.

To better comprehend how Spring operates with proxies, let's translate the concept into Java source code.

Proxy Pattern in Code

The proxy pattern, programmed in code, has the following basic form:

```
class Subject {
  public String operation( String input ) { return input; }
}

class Proxy extends Subject {
  private final Subject subject;

  Proxy( Subject subject ) { this.subject = subject; }

  @Override public String operation( String input ) {

    // Preprocess the input

    String result = subject.operation( input );

    // Postprocess the result

    return result;
  }
}
```

The proxy must be connected to the target object. This is how the constructor stores the reference.

What is symbolic here in Java code is automatically generated at runtime in the Spring Framework.

4.1.2 Dynamically Generate Proxies

The Spring Framework can generate proxies at runtime; this is why we refer to this form of proxy as *dynamic proxies*. The Spring Framework uses two techniques for this purpose.

- The Java Development Kit (JDK) can build dynamic proxies; these are called *JDK dynamic proxies*. In this case, there is a limitation because Java SE can only realize proxies for interfaces—this isn't possible for classes.
- When Spring generates a proxy for classes, the framework makes use of the library *cglib* (short for *Byte Code Generation Library*).[1] This Byte Code Generation Library is relatively old and no longer maintained, so the Spring team has heavily patched the version to make it fit for the current Java versions.

1 *https://github.com/cglib/cglib*

Rules for Target Objects

If proxies are to work well, then the classes must take some rules into account. If Spring builds proxy objects for classes, then a library creates bytecode for a subclass. It follows that the superclass (the subject) must not have a private constructor. This is clear because a subclass always calls the constructor of the superclass, and if the superclass doesn't have a visible constructor, the constructor can't be called. Therefore, the constructors in the subject must not be private.

If Spring implements a proxy through a subclass, as we did in our own proxy example with the types `Proxy` and `Subject` in the previous section, then the proxy must override the method from the subject. Therefore, the method must not be `final`; otherwise, it could not be overridden.

In principle, all methods for which a proxy is generated should be `public`. Spring relies on its own aspect-oriented programming (AOP) framework, which requires `public` methods. Although all of these restrictions can be circumvented, then the bytecode must be modified via AspectJ. We'll only use the "usual" way here.

Besides the preceding rules, there are a few limitations if the framework is creating proxy objects. There is one thing to watch out for: A proxy can only do its job if it sits between the client and the subject. If the subject calls methods among themselves in their own code, then it won't go through the proxy. This can quickly lead to a problem, for example, when refactoring.

To illustrate the issue at hand, let's consider the example of a proxy class named `TrimmingProxy`. When parameter variables are annotated with `@Trim`, the `TrimmingProxy` class removes any leading or trailing white spaces from these strings before passing them to the target object. Here is an example usage of the `TrimmingProxy`:

```
IntParser proxy = new TrimmingProxy( new IntParser() );
proxy.parseInt( " 12423\t " );
```

Now let's look at the code of the `IntParser` class with the method:

```
class IntParser {
  public int parseInt( String input ) {
    return parseInt( input, 10 );
  }
  public int parseInt( @Trim String input, int radix ) {
    return Integer.parseInt( input, radix );
  }
}
```

If the proxy pays attention to `@Trim`, it will only be activated when `parseInt(String, int)` is called *directly*; otherwise, it won't be activated. Our example calls `parseInt(String)`, and the parameter variable has no `@Trim`, so the proxy won't remove

white space either. If `parseInt(String)` later calls `parseInt(String, int)`, it happens inside the target object, and the proxy isn't involved.

Note

In practice, this is a mistake that is often made, so you must always remember that a proxy always goes into the core "from the outside" and only then does its job.

Build Proxies Yourself with ProxyFactory *

Spring provides the class `ProxyFactory`[2] with which we can also build proxy objects. Here's an example: A proxy object is to be created for a `java.util.List`. On this particular list, `add(null)` is to be ignored:

```
ProxyFactory factory = new ProxyFactory( new ArrayList<>() );
MethodInterceptor methodInterceptor = invocation ->
  (invocation.getMethod().getName() == "add"
  && isNull( invocation.getArguments()[ 0 ] )) ? false
                                               : invocation.proceed();
factory.addAdvice( methodInterceptor );
```

In the constructor of `ProxyFactory`, we pass the subject, an `ArrayList`. The goal is that `ProxyFactory` generates a proxy, and we pass certain methods to the list in the background.

The proxy intercepts every method call. It's therefore elementary to configure how the proxy should behave on which method and arguments. A `MethodInterceptor`[3] is helping us; the functional interface is declared as follows:

```
public interface MethodInterceptor extends Interceptor {
  @Nullable
  Object invoke(@Nonnull MethodInvocation invocation) throws Throwable;
}
```

If a method is called on the proxy from the outside, it passes the call to the configured `MethodInterceptor` and calls the `invoke(…)` method. In other words, no matter which method is accessed from the outside, it will always map to an `invoke(…)` method. The `MethodInvocation` parameter is so important because the object contains information about which method was called with which arguments.

Our implementation first tests with `invocation.getMethod().getName() == "add"` if the method `add` was really called. Because the strings of the method names are internal, it's

2 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/aop/framework/ProxyFactory.html*

3 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/aopalliance/intercept/MethodInterceptor.html*

not a problem to use `==` for the comparison; however, this is rare. The subsequent check with `isNull( invocation.getArguments()[ 0 ] )` checks if `null` was passed to the `add(…)` method. There are two outputs for the comparisons:

- `add(null)` has been called. In this case, we return a `boolean` value because this is also the return type of the `add(…)` method. Returning `false` expresses that no change was made to the data structure. We don't pass anything to the target objet. Although `invoke(…)` only returns `Object`, it's important to make sure that the return type matches that of the target object in the method.
- If `add(null)` was *not* called, so either another method or `add(…)` with another value than `null`, then the proxy goes to the target object with `invocation.proceed()`, and what the subject returns, we return at `invoke(…)`.

With this code, the `MethodInterceptor` is implemented. After it's passed into `addAdvice(…)`, we can obtain and use the proxy object from the `ProxyFactory`. Here's an example:

```
List list = (List) factory.getProxy();
list.add( "One" );
System.out.println( list );
list.add( null );
list.add( "Two" );
System.out.println( list );
```

The outputs are as follows:

```
[One]
[One, Two]
```

Having discussed home-coded proxies, let's now examine some standard proxies provided by the Spring Framework.

4.2 Caching

In this section, we'll delve into the concept of caching, how it can be implemented in Spring applications, and the benefits it provides. *Caching* is an essential technique used to speed up application performance, reduce database load, and enhance user experience. With Spring's robust caching support, you can easily incorporate caching into your applications and reap its benefits.

4.2.1 Optimization through Caching

Caching plays a crucial role in scenarios where data retrieval is resource-intensive, and a quick cache is available to temporarily store the object. Caching has various practical applications:

- The Domain Name System (DNS) cache is used by every PC and smartphone to recall the IP address of a specific host name.
- Companies make use of a web proxy and cache wherein the proxy caches the retrieved components when queries are sent out from the company network.
- When a web server is used, not all queries require file access because the server retains the file contents and only refreshes the cache entry when modifications are made to the file.
- Spring applications often store the contents of a database in the cache to minimize database access.

Caching Challenges

Caching is only worthwhile if the element being queried is relatively expensive to create or query.[4] Stored elements that are then invalidated are a particular challenge. For a purely idempotent operation, such as a sine function (something comes in and the same thing always comes out), caching is easy. But many entries eventually become invalid. For example, if a record is associated with an ID, then the record may change. Consequently, the cache content must be marked as *stale*.

A cache must also work like the human brain; that is, it must be able to forget. It's of little use if the cache occupies the entire main memory, and this leads to the `OutOfMemoryError` of the application. The cache must forget elements that haven't been used for a long time, just as probably all of us have forgotten the components of a flowering plant from the sixth grade.

4.2.2 Caching in Spring

In Spring, enabling caching is straightforward. The underlying mechanism involves a Spring proxy that associates method arguments with their respective return values. Additionally, it's easy to swap different caching implementations with minimal effort.

There are several approaches to implementing a caching mechanism in Spring. Usually, caching is realized declaratively, which means that the actual cache access isn't visible in your own code. There are two declarative approaches: *annotation-based* or *XML-based*. We'll omit the XML method because it's uncommon today, as is the whole container configuration.

There are two options for annotations:

4 Back in the early days of Java, developers were concerned about the cost of object creation and often tried to reuse objects to save memory. However, this approach is generally not recommended today as the cost of object creation and cleanup has become relatively cheap. In fact, artificially keeping objects alive for too long can actually cause issues with memory management. This is due to the generation-based garbage collector moving these objects to the "old generation" where memory release isn't as fast as with fresh objects. As a result, it's important to carefully consider object lifecycle management to avoid any unintended consequences.

- Spring annotations such as `@Cacheable`, `@CacheEvict`, `@CachePut`, `@Caching`, and `@CacheConfig`
- JCache annotations such as `@CacheResult`, `@CachePut`, `@CacheRemove`, `@CacheRemoveAll`, `@CacheDefaults`, `@CacheKey`, and `@CacheValue`

Note

JCache is the standard from JSR 107 (*www.jcp.org/en/jsr/detail?id=107*).

Not only are the Spring and JCache annotations named differently, but the semantics are different in places; JCache also caches exceptions. Spring Cache can also be configured with a JCache implementation.

In the following example, the Spring annotations are used.

@EnableCaching

The actual work in the background is done by proxy objects. A proxy will later wrap itself around the objects and cache the result of the annotated method. The proxies aren't built automatically, but for this purpose, the annotation `@EnableCaching`[5] has to be present on any configuration class:

```
@EnableCaching
@Configuration
class CacheConfig { }
```

Our main class that was annotated with `SpringBootApplication` is a special `@Configuration` class, so it would work there too.

If methods are later called that are annotated with `@Cacheable`, `@CacheEvict`, and so on, a proxy will receive the calls.

4.2.3 Component with the @Cacheable Method

Consider the following brief example of an operation that should store its output in the cache:

```
@Component
class HotProfileToJsonConverter {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Cacheable( "date4u.jsonhotprofiles" )
  // @Cacheable( cacheNames = "date4u.jsonhotprofiles" )
```

5 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/EnableCaching.html*

```
  public String hotAsJson( List<Long> ids ) {
    log.info( "Generating JSON for list {}", ids );
    return ids.stream().map( String::valueOf )
              .collect( Collectors.joining( ",", "[", "]" ) );
  }
}
```

The `hotAsJson(…)` method gets a list of IDs featuring the hottest unicorns and responds with a JavaScript Object Notation (JSON) string.

There are three key aspects to consider when dealing with caching:

- The annotation `@Cacheable`[6] is used on specific methods to associate the method's arguments with its corresponding return value. Essentially, a cache is an associative data store that maps a key to a value, where the key is the argument passed to the method. With each unique argument passed, the proxy retains the resulting return value in memory.
- A cache always needs a name; a hierarchical structure is useful to avoid conflicts with other applications. The name can be set via annotation attribute `cacheNames` or `value`. The name is important for two reasons:
  - Different caches can be used within one class.
  - Different classes can use the same cache.
- The methods that the proxy calls must be `public`. We discussed this in Section 4.1.2 under "Rules for Target Objects" subsection. In principle, other visibilities would be possible, but the proxy must fall back on these methods, and this works best if the method is `public`. Other visibilities are possible, but then the bytecode must be wrapped with AspectJ.

Now let's see how we can use the `HotProfileToJsonConverter`.

4.2.4 Use @Cacheable Proxy

The `HotProfileToJsonConverter` is a Spring-managed bean and can be injected:

```
@Autowired HotProfileToJsonConverter converter;
```

What Spring injects is *not* our own object! We get something that is of type `HotProfileToJsonConverter`, but it's the proxy. In the debugger (or via the `Class` object), this is easy to see:

```
log.info( converter.getClass().getName() );
// c.t.d.HotProfileToJsonConverter$$EnhancerBySpringCGLIB$$8d40339b
```

6 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html*

The name of the `Class` object contains the string `EnhancerBySpringCGLIB`, and this is the typical identification of a proxy generated at runtime.

This proxy has the `hotAsJson(…)` method that we can use:

```
String json1 = converter.hotAsJson( Arrays.asList( 1L, 2L, 3L ) );
String json2 = converter.hotAsJson( Arrays.asList( 1L, 2L, 3L ) );
// assertThat( json1 ).isSameAs( json2 );
```

The resulting strings aren't just equivalent—they are also identical. This is a crucial point, which is attributed to the working of the proxy.

Even the first call to `hotAsJson(…)` is received by the proxy. The proxy maps the argument to a key (how to do this in a customized way is shown in Section 4.2.9) and internally requests the cache with the question, "Is there already an associated result for the list with values 1, 2, 3?" The answer is "no" in the first query. Therefore, the proxy calls the core, gets the string, and puts it in the cache. With the second call of `hotAsJson(…)`, it already looks different: even then, the proxy goes to the cache first, but it has an entry for the list of 1, 2, 3. In other words, the second call doesn't go to the core, but the proxy returns the cached value. Therefore, it appears in the log output only once:

```
Generating JSON for list [1, 2, 3]
```

If you were to work with a second list, the same thing would start all over again:

```
String json3 = converter.hotAsJson( Arrays.asList( 1L ) );
// assertThat( jspn1 ).isNotSameAs( json3 );
```

The cache hasn't seen the list with 1 yet, so it goes to the core and caches the result. Of course, the JSON strings of lists 1, 2, 3 and 1 are entirely different.

Cache Images

We'll examine caching using a concrete example in our Date4u application.

In `PhotoService`, there is a `download(…)` method with the following implementation:

```
public Optional<byte[]> download( String imageName ) {
  try {
    return Optional.of( fs.load( imageName + ".jpg" ) );
  }
  catch ( UncheckedIOException e ) {
    return Optional.empty();
  }
}
```

Listing 4.1 PhotoService.java Excerpt

An image name is passed to the `download(…)` method, and the return is `Optional` with a `byte` array. Internally, the method calls the file system.

If the `download(…)` method is called twice, the image could easily come out of the cache as a `byte` array. It can be implemented as follows:

1. In any configuration, `@EnableCaching` must be present. Because `@SpringBootApplication` is a special kind of `@Configuration`, it is valid to apply the `@EnableCaching` to the class with the `main` method:

```
@SpringBootApplication
@EnableCaching
 public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
 }
```

Listing 4.2 Date4uApplication.java Extension

2. For the `PhotoService`, a name must be added to the `@Cacheable` annotation.

```
@Cacheable( "date4u.filesystem.file" )
 public Optional<byte[]> download( String imageName )
```

Listing 4.3 PhotoService.java Extension

The cache name is arbitrary, and `date4u.filesystem.file` works and is unique.

4.2.5 @Cacheable + Condition

In some situations, specific items ought not to be cached. This could be due to their size or because they aren't frequently accessed. Thus, caching can be associated with certain criteria, such as if an object possesses particular attributes, it should not be cached.

In Spring, the `condition` annotation attribute[7] controls whether an object should be cached. If the `condition` is true, the object is cached. In the code, it looks like this:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            condition  = "true" )
```

This condition is written as a Spring Expression Language (SpEL) expression (refer to Chapter 2, Section 2.11). SpEL is powerful and allows access to the object graph and to all parameters. `condition = true` is a simple SpEL expression and exemplifies the notation. By the way, the assignment `condition = true` is the default.

7 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#condition()*

Suppose that caching with `hotAsJson(…)` is only beneficial for extensive lists, and small lists should not be cached. This idea can be conveyed as follows:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            condition  = "#ids.size() > 10" )
public String hotAsJson( List<Long> ids )
```

The condition `#ids.size() > 10` contains an expression regarding whether the list contains more than 10 elements. The variable `ids` is the parameter name that can be accessed by the SpEL expression. The parameter type is `java.util.List`, and it has a `size()` method. If the number of list items exceeds 10, then the cache should be asked and also filled. This is obvious: cache large lists, not small lists. If the condition is false—that is, the list is small—then the call goes directly to the core and also again from the target object directly to the client without caching the object.

For the condition, the parameters are usually used, and further identifiers are predefined in SpEL:

- `#root.method`, `#root.target`, `#root.caches`
- `#root.methodName`, `#root.targetClass`
- `#root.args[0]`, `#p0`, `#a0`, `root.args[1]`, and so on, or simply `#parameter-name`

You can fall back on the reflection `Method` object or, alternatively, on the method name, `Class` object, or class name.

There are different ways of writing to access the parameters. The most readable is the variant that our example uses: with the number sign and the parameter name.

4.2.6 @Cacheable + Unless

In the Spring caching framework, a veto is a mechanism that allows the programmer to prevent certain method invocations from being cached. More specifically, it ensures that a previously uncached result won't be cached even if the method is invoked multiple times with the same arguments.

This can be particularly useful in scenarios where the method may return `null` or empty results, as caching such outcomes could result in wasted memory and reduced performance. By setting a veto, the programmer can instruct the caching system to skip the caching process for specific method outcomes.

In Spring caching, the veto is expressed with the annotation attribute `unless`:[8]

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            unless     = "false" )
```

8 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html#unless()*

The SpEL expression here is `false`, which means there is *no* veto, so the result goes into the cache.

`unless` set to `false` is only an example for clarification. In practice, a SpEL expression is used, and this can fall back on the known information, the method name, the parameter list, and so on. An additional variable, `result`, provides access to the result from the core in the SpEL expression. This can be used to decide whether the object goes into the cache or not.

Let's move on to something more practical with a couple of examples:

Example 1: A JSON result under 100 characters should not be cached:

```
@Cacheable( cacheNames = "date4u.jsonhotprofiles",
            unless     = "#result.length() < 100" )
```

If the `result`—it's a `String` here—has less than 100 characters, then it's a veto, and the string should not be cached. `unless` controls in this way that small strings aren't cached and that caching is only worthwhile if the strings become larger.

Example 2: If a method returns a collection, but the result is `null` or the collection is empty, this should be a veto. The SpEL expression could look like this:

```
#result == null || #result.size() == 0
```

When the veto is true, the collection is prevented from being stored in the cache. On the other hand, if there is no veto, and the collection isn't `null` or empty, it will be stored in the cache.

Remarks

If an exception occurs in the core that the cache proxy wraps, those exceptions aren't cached. That is, there is no association between a value and the exception that it doesn't work with the value.

If a method returns an `Optional`, the result doesn't contain the `Optional`, but contains either `null` (if the `Optional isEmpty()`) or the value from the `Optional`.

4.2.7 @CachePut

If a method is annotated with `@CachePut`,[9] the cache can be written directly. This is convenient when the cache is to be filled initially because, if the elements aren't present, the cache doesn't have to be queried first. `@CachePut` is also useful when elements in the cache are to be updated and overwritten, such as for outdated values. The underlying method in the core is thus always called, and the value is determined and stored.

9 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CachePut.html*

`@CachePut` and `@Cacheable` have similar annotation attributes. But using both annotations together doesn't make sense, and we use either `@CachePut` or `@Cacheable`.

4.2.8 @CacheEvict

Effective cache management is essential to prevent it from becoming too large and impacting system performance. There are two common strategies for managing a cache's elements:

- **Automatic expiration**
  This strategy involves setting conditions that trigger the cache to remove elements on its own. For example, a cache may have a *time-to-live* (TTL) value that specifies the maximum amount of time an element can remain in the cache before it's expired and removed automatically. Another example is setting a maximum cache size, where the oldest elements are removed when the cache reaches its limit.
- **Manual eviction**
  In this strategy, the programmer manually deletes specific elements or clears the entire cache. This can be useful in scenarios where the cached elements are no longer valid or when memory needs to be reclaimed.

Choosing the appropriate strategy will depend on the specific use case and requirements of the application. Automatic expiration may be suitable for caches that have a fixed lifetime or are used for infrequently accessed data. Manual eviction, on the other hand, may be more appropriate for caches that store sensitive or frequently changing data.

The first point pertains to a configuration matter, and we'll examine the process in Section 4.2.12 on a local cache.

A method annotated with `@CacheEvict`[10] can delete entries from the cache:

```
@CacheEvict( cacheNames = "date4u.jsonhotprofiles" )
public void removeHotAsJson( List<Long> ids )
```

Again, the cache name is given in the annotation, and the parameter list is as known.

If we call the method from outside, the cache will remove the element. With the assignment `allEntries=true`, all elements for the named cache can be deleted.

```
@CacheEvict( cacheNames = "date4u.jsonhotprofiles",
             allEntries = true )
public void removeAllHotAsJson()
```

The parameter list is empty because a parameter isn't necessary. `condition` is also allowed with this annotation.

10 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheEvict.html*

Methods annotated with `@CacheEvict` are usually responsible for deleting data from an underlying data store. As a result, these methods should also remove the corresponding data from the cache.

4.2.9 Specify Your Own Key Generators

In a cache, a key is always associated with a value. For methods annotated with `@Cacheable`, `@CachePut`, or `@CacheEvict`, the cache proxy must generate a key from the provided arguments. If a method has a single parameter with a "friendly" key type such as `long` or `String`, the key formation is straightforward. However, key formation can become more complicated when dealing with methods with multiple parameters or custom key types.

The problem starts when a method declares no parameter or more than one parameter. In addition, there may be parameter types that are not good keys.

To generate a key from passed arguments of the method, Spring uses a `KeyGenerator`[11] (see Figure 4.2). The interface condenses the passed arguments of a method into an `Object`, which is the key for the cache.

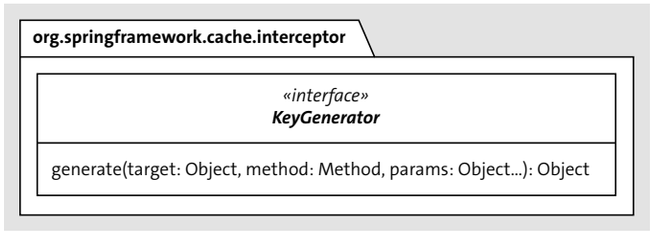


Figure 4.2 The "KeyGenerator" Functional Interface

SimpleKeyGenerator

By default, the Spring Framework uses the `KeyGenerator` implementation `SimpleKeyGenerator` for mapping arguments to a cache key.[12] The code[13] is simple:

```
public class SimpleKeyGenerator implements KeyGenerator {

    @Override
    public Object generate(Object target, Method method, Object... params) {
        return generateKey(params);
    }
```

11 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html*

12 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKeyGenerator.html*

13 *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKeyGenerator.java*


```
    public static Object generateKey(Object... params) {
        if (params.length == 0) {
            return SimpleKey.EMPTY;
        }
        if (params.length == 1) {
            Object param = params[0];
            if (param != null && !param.getClass().isArray()) {
                return param;
            }
        }
        return new SimpleKey(params);
    }
}
```

The overridden method `generate(…)` from the interface calls its own public static method `generateKey(…)`, which does the actual mapping. `target` and the `Method` object aren't used by the `SimpleKeyGenerator`—the implementation only builds the cache key from the arguments of a method.

The method `generateKey(…)` considers four cases for the handover to the method intercepted by the cache proxy:

- The method has no parameters. This is rare, but valid, and, in this case, the return is `SimpleKey.EMPTY`. In this way, expensive factory methods can be declared whose results can be temporarily cached.
- The method has exactly one parameter, which is not `null` and also not an array. Then the passed argument really forms the key directly. This was the case with our list of IDs, and it's performant and memory-saving because no additional container objects are needed.
- The method has exactly one parameter, and it's `null`, or an array was passed as a data structure. Then a `SimpleKey` object is built with `null` or the array.
- If multiple parameters exist, a `SimpleKey` is generated for the arguments.

The SimpleKey Object

Often, the cache will reference a `SimpleKey`[14] object as the key. The code[15] looks like this (`readObject(…)` for serialization has been omitted):

```
package org.springframework.cache.interceptor;

import …
```

14 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/interceptor/SimpleKey.html*

15 *https://github.com/spring-projects/spring-framework/blob/main/spring-context/src/main/java/org/springframework/cache/interceptor/SimpleKey.java*

```
public class SimpleKey implements Serializable {

    public static final SimpleKey EMPTY = new SimpleKey();


    private final Object[] params;

    private transient int hashCode;


    public SimpleKey(Object... elements) {
        Assert.notNull(elements, "Elements must not be null");
        this.params = elements.clone();
        this.hashCode = Arrays.deepHashCode(this.params);
    }


    @Override
    public boolean equals(@Nullable Object other) {
        return (this == other ||
                (other instanceof SimpleKey &&
                        Arrays.deepEquals(this.params,
                          ((SimpleKey) other).params)));
    }

    @Override
    public final int hashCode() {
        return this.hashCode;
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + " ["
        + StringUtils.arrayToCommaDelimitedString(this.params) + "]";
    }
    …
}
```

Listing 4.4 SimpleKey.java

The constructor accepts the method arguments passed to the proxy, copies the values, and calculates the hash value in advance so that the later query is cheap. The implementations of the `equals(…)` and `hashCode()` methods are important because the key must be found in the associative store. A `toString()` implementation is useful but unnecessary for the cache.

Set Your Own Key Generator via Key or keyGenerator

The `SimpleKeyGenerator` is a good standard, but it has disadvantages if, for example, `equals(…)`/`hashCode()` aren't overridden, aren't well implemented, or the object isn't desired as a key in the cache. Therefore, you can specify a key generator, and Spring provides two ways to do this.

The annotation types `@Cacheable`, `@CachePut`, and `@CacheEvict` provide two different ways to set a key generator via an annotation attribute:

```
public @interface Cacheable / CachePut / CacheEvict {
   …
  // Option 1:
  String key() default "";           (1)

  // Option 2:
  String keyGenerator() default "";  (2)
   …
}
```

Option 1 is a SpEL expression to extract a key; for option 2, we set the bean name of a Spring-managed bean, which implements the interface type `KeyGenerator`. We'll look at the details of option 2 in the next section.

Option 1 uses a SpEL expression to retrieve the key from the provided object. However, suppose a complicated object such as a book is involved. In that case, using the entire book as the cache key may be unnecessary and even unfeasible if there are no suitable `equals(…)`/`hashCode()` methods implemented. A more reasonable approach is to resort to the ISBN, which can serve as an excellent key for the object if it's a string.

Set the Key Generator via Key

Consider the type `Profil`, which represents a dating profile. While `Profile` objects should not be used as keys, a profile ID of type `long` serves as an excellent cache key.

```
@Cacheable( cacheNames = "…",
            key        = "#prof.id" )
Object method( Profile prof )
```

If a SpEL expression is specified for `key`, Spring will no longer use the `SimpleKeyGenerator` for key generation, but will fall back on the SpEL expression for key generation.

SpEL can make use of context objects, as we've already seen with `condition` and `unless`:

- `#root.args[0]`, `#p0`, `#a0`, or simply `#parameter-name`
- `#root.method`, `#root.methodName`, `#root.target`, `#root.targetClass`, and so on

Thus, `"#prof.id"` accesses the parameter `prof` and then the `id`. In the cache, the keys are of type `Long` (wrapper types) and not of type `Profile`.

Task: Set the Key Generator via Key

You can practice caching with the next task. As a reminder, `PhotoService` had a method `download(String)`:

```
public class PhotoService {
  …
  public Optional<byte[]> download( String name ) {
    try { return Optional.of( fs.load( name + ".jpg" ) ); }
    catch ( UncheckedIOException e ) { return Optional.empty(); }
  }
  …
}
```

Listing 4.5 PhotoService.java

In `PhotoService`, another overloaded method `download(Photo)` is to be added, so that callers can decide whether they want to get the image via the file name or via the `photo` (`Photo` contains the file name). `Photo` is a complex data type, and a key generator is needed:

```
public Optional<byte[]> download( Photo photo ) { implementMe }
```

The `Photo` parameter type is new and should look like this:

```
public class Photo {
  public Long          id;
  public Long          profile;
  public String        name;
  public boolean       isProfilePhoto;
  public LocalDateTime created;
}
```

Listing 4.6 Photo.java

Later, we'll extend the class so that photos can also be stored in the database.

A SpEL key generator is to be implemented that extracts `name` from `Photo` as the cache key.

The proposed solution follows:

```
@Cacheable( cacheNames = "date4u.filesystem.file",
            key        = "#photo.name" )
public Optional<byte[]> download( Photo photo ) {
  return download( photo.getName() );
}
```

Listing 4.7 PhotoService.java Extension

The parameter variable is called `photo`, and the SpEL expression `#photo.name` thus references the name, which is a string. Internally, the cache thus forms associations between the file name and the `byte` array.

Set the Key Generator via KeyGenerator

Returning to option 2, the custom `KeyGenerator` implementation, it's worth noting that although SpEL expressions are convenient and compact, they are less efficient and not as well-typed as direct Java code.

The `KeyGenerator` interface prescribes this method:

```
Object generate(Object target, Method method, Object... params)
```

The following example returns a `KeyGenerator` implementation via a `@Bean` method inside a `@Configuration` class:

```
@Bean
public KeyGenerator photoNameKeyGenerator() {
  return ( Object __, Method ___, Object... params ) -> {
    if ( params.length == 1 && params[ 0 ] instanceof Photo photo )
      return photo.name;
    throw new UnsupportedOperationException(
        "Can't apply this KeyGenerator here" );
  };
}
```

Because `KeyGenerator` is a functional interface, it can be implemented concisely using a lambda expression. From the parameter list, `Object` target and `Method` method aren't needed, only `params`, for the method arguments passed to the cache proxy.

If the `KeyGenerator` is used incorrectly, a consistency check follows: `params.length` must be `1`, and the type must be `Photo`. If this is true, the name is extracted and returned; otherwise, an exception follows.

The `KeyGenerator` implementation isn't set via a `Class` object, but via the bean name; the method is called `photoNameKeyGenerator`, and this is also the name of the component. It's set in the annotation attribute `keyGenerator`:

```
@Cacheable( cacheNames   = "date4u.filesystem.file",
            keyGenerator = "photoNameKeyGenerator" )
public Optional<byte[]> download( Photo photo )
```

Because `key` and `keyGenerator` are both strings, there is a danger of confusion.

4.2.10 @CacheConfig

If specifications like the cache name apply to all methods of a class, code duplication at each annotated caching method isn't optimal. Another solution is provided by the `@CacheConfig`[16] annotation, which is valid on one type and determines the name for `cacheNames`.

Here's an example:

```
@Component
@CacheConfig( cacheNames = "date4u.jsonhotprofiles" )
class HotProfileToJsonConverter {
  @Cacheable
  public String hotAsJson( List<Long> ids ) { … }
}
```

The cache name can be omitted for the individual methods. However, methods could also reassign the name.

In total, the `@org.springframework.cache.annotation.CacheConfig` annotation type declares four annotation attributes:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CacheConfig {
    String[] cacheNames() default {};
    String keyGenerator() default "";
    String cacheManager() default "";
    String cacheResolver() default "";
}
```

4.2.11 Cache Implementations

Up to this point, we've enabled caching declaratively using annotations, but we haven't yet discussed the actual implementation of the cache. By default, Spring uses the `java.util.concurrent.ConcurrentHashMap` data structure, which is a specialized associative store based on hash tables that allows concurrent modifications. However, this data structure isn't a true cache because it lacks an important property: a good cache should forget elements that haven't been used for a long time. In contrast, a `ConcurrentHashMap` would never forget values.

16 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/annotation/CacheConfig.html*

Distributed versus Local Caches

There are numerous cache implementations on the market, which can be categorized as follows:

- **Distributed cache**
  Data is stored on a cache server and not locally. All applications share the data on the cache server. Typical products in use are *Redis*, *Ehcache*, *Hazelcast*, *Infinispan*, *Couchbase*.
- **Local cache**
  Each application has its own cache in main memory. Typical Java libraries are *Caffeine* and *cache2k*.

Spring recognizes what is present based on the entry in the classpath via auto-configuration, and the proxy forwards the cache operations to the implementations. Often the lifetime, size, and so on can be configured externally via configuration properties, so that your own code doesn't notice any change of the cache implementation.

4.2.12 Caching with Caffeine

To illustrate a cache implementation, consider the open-source library Caffeine just mentioned (*https://github.com/ben-manes/caffeine*), which evolved from the *Google Guava cache*.

Two dependencies are necessary in the project object model (POM):

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
<dependency>
  <groupId>com.github.ben-manes.caffeine</groupId>
  <artifactId>caffeine</artifactId>
  <version>3.1.8</version>
</dependency>
```

Listing 4.8 pom.xml Extension

`spring-boot-starter-cache` combines two dependencies, `spring-boot-starter` and `spring-contextsupport`, and has no code itself.

Next, the cache must be configured; this is possible, for example, via *application.properties*:

```
spring.cache.caffeine.spec=maximumSize=1,expireAfterWrite=10s
```

The key `spring.cache.caffeine.spec` contains a string with multiple segments separated by commas. One of these segments is the `maximumSize`, which can be set to `1` for testing purposes, meaning that only one element can be stored in the cache at a time. Additionally, the `expireAfterWrite` segment is set to `10` seconds, which causes any cached entry to automatically expire and be removed from the cache after 10 seconds have passed. It's worth noting that setting the `maximumSize` to `0` would disable caching altogether.

Note

To explore the configurations in detail, go to *www.javadoc.io/doc/com.github.benmanes.caffeine/caffeine/latest/com.github.benmanes.caffeine/com/github/benmanes/caffeine/cache/CaffeineSpec.html*.

In short, the following can be set:

- `initialCapacity=[integer]`
- `maximumSize=[long]`
- `maximumWeight=[long]`
- `expireAfterAccess=[duration]`
- `expireAfterWrite=[duration]`
- `refreshAfterWrite=[duration]`
- `weakKeys`
- `weakValues`
- `softValues`
- `recordStats`

If we make no configuration, a default configuration applies.

4.2.13 Disable the Caching in the Test

In the test case, caching is usually switched off. There are various solution approaches that can also be transferred to other technologies:

- **Option 1**
  Mostly there is a switch that can turn something off or on. `spring.shell.interactive.enabled=false` will turn off the shell, and `spring.main.banner-mode=off` can turn off the banner. For caching, this configuration property is `spring.cache.type`:

  ```
  spring.cache.type=NONE
  ```

  The assignment with `NONE` switches off the caching.

- **Option 2**
  Spring Boot enables a `CacheManager`[17] via auto-configuration. If there is no cache manager, Spring builds an instance. If we now build a cache manager ourselves, auto-configuration has nothing to do, and there is no cache:

  ```
  @Bean
  CacheManager cacheManager() {
    return new NoOpCacheManager();
  }
  ```

  The factory method is as usual in a `@Configuration` class and returns the `NoOpCacheManager`. As the name says—`NoOp`(eration)—nothing happens.
- **Option 3**
  Caching needs a proxy generator, and this is enabled by `@EnableCaching`. If the cache is to be disabled, this can be controlled by enabling `@EnableCaching`. For example, a profile could activate `@Configuration`. For example, if the application isn't in development mode, that is, it's in productive mode, this configuration should first be activated with `@EnableCaching`:

  ```
  @Profile( "!dev" )
  @EnableCaching
  @Configuration
  public class CachingConfig { }
  ```

  If one is in development mode, the profile isn't activated, so there is also no configuration. Without configuration, there is also no `@EnableCaching`—so no cache.

4.3 Asynchronous Calls

Concurrency is an essential aspect of modern programming that enables high-performance and efficient applications. The Spring Framework provides a powerful mechanism to use this capability through its proxy functionality, which allows for the execution of operations in a separate background thread. This feature enables the main program to continue its work while the background thread processes the task, leading to faster and more responsive applications.

To implement Spring proxies in your projects, you need to define the beans and configure the proxy settings in your application context. Once configured, Spring will automatically generate the proxy instances that you can use in your code. It's also possible to customize the proxy behavior by specifying additional properties, such as concurrency level, thread pool size, and synchronization mode.

In this section, we'll explore the benefits of using Spring proxies for concurrent programming and how to implement them in your projects.

17 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/cache/CacheManager.html*

4.3.1 @EnableAsync and @Async Methods

As usual, the async proxy must be enabled; for this, a `@Configuration` is annotated with `@EnableAsync`.[18] In the next step, methods are annotated with `@Async`.[19] Here's an example:

```
@Async
public void heavyWork()
```

If we get a reference to the proxy wrapping the `heavyWork(…)` method, a call will by default cause the proxy to use a background thread and the `heavyWork(…)` method to work concurrently.

Asynchronous methods can have two different returns:

- In the simple case, you don't deliver anything back. This is also called the *fire-and-forget strategy*. That is, the return type is `void`.
- A method executed in the background can return a result, but a special return type must be used for this—a kind of container—because the receiver doesn't know when it will get the return. The answer may come 10 milliseconds later or a day later. Therefore, the `@Async` methods usually have the `Future` return type or the `CompletableFuture` subtype. Via the `Future` objects, the client receives the result later when it's available.

Note: Language Comparison

While JavaScript has an `async` keyword for handling concurrency, Java lacks native language support for this feature. Although the Java SE library provides some concurrency tools, Spring's proxy functionality allows for the use of `@Async` methods that are almost as concise and user-friendly as their JavaScript counterparts. Additionally, implementing proxies through a library offers more flexibility for configuration compared to building the functionality directly into the language.

4.3.2 Example with @Async

Consider an example where both scenarios are combined: an `@Async` method with and without a return value.

```
@Component
class SleepAndDream {

  private final Logger log = LoggerFactory.getLogger( getClass() );
```

18 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/EnableAsync.html*

19 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html*


```
  @Async
  public void sleep1Seconds() throws Exception {
    log.info( "Going to sleep: SNNNNOOORRRREEEEE" );
    TimeUnit.SECONDS.sleep( 1 );
    log.info( "Woke up" );
  }

  @Async
  public CompletableFuture<String> sleep1SecondWithDream()
      throws Exception {
    log.info( "Starting to dream" );
    TimeUnit.SECONDS.sleep( 1 );
    log.info( "Finished the dream" );
    return CompletableFuture.completedFuture( "about unicorns" );
  }
}
```

The two methods are annotated with `@Async` and are `public`; this is important for the proxy generator. Something else is important too: if the methods call each other, there is no proxy in between.

Following are some details about each methods:

- **First method: `sleep1Seconds(…)`**
  This method outputs a logger message, sleeps for 1 second, so to speak, pauses the executing thread for 1 second, and writes a log message again. The `sleep(…)` method from the `TimeUnit` is a viable alternative to the `Thread.sleep(…)` method. Both `sleep(…)` methods can throw an `InterruptedException`, which the code passes up to the framework.
- **Second method: `sleep1SecondWithDream()`**
  This method has a return and uses the `Future` types. The code is similar to the first method, only at the end, the method builds, fills, and returns a container of type `CompletableFuture`. A `Future` object is somewhat similar to `Optional`, except it can take time to fill with data.

The `SleepAndDream` component can be injected; strictly speaking, it's the async proxy we get:

```
@Autowired SleepAndDream dreamer;
```

The next example calls the methods and therefore sets log messages. The comment at the beginning of the line indicates the threads that are active before the line is executed. `M` stands for the main thread, and `1` and `2` exemplarily for the threads used by the framework.

```
/*M  */ log.info( "Before sleep1Seconds()" );
/*M  */ dreamer.sleep1Seconds();
/*M1 */ log.info( "After sleep1Seconds()" );

/*M1 */ log.info( "Before sleep1SecondWithDream()" );
/*M1 */ Future<String> future = dreamer.sleep1SecondWithDream();
/*M12*/ log.info( "After sleep1SecondWithDream()" );

/*M12*/ TimeUnit.SECONDS.sleep( 2 );

/*M  */ log.info( "Dream: " + future.get() );
```

In simple Spring boot applications, the main thread executes the code; we see an `M` at the beginning. After the call to `sleep1Seconds(…)`, a short time later, there are two threads of action: one from the main thread and one from the (presumably newly built) thread that processes `sleep1Seconds()`. The main thread continues to run, unimpressed, the whole time. The thread reserved for `sleep1Seconds(…)` hangs in the method for 1 second and logs.

If the client on the proxy calls the second method, `sleep1SecondWithDream()`, then after a few milliseconds, another thread is used/created. This one is also condemned to sleep for 1 second. That is, two threads are in the waiting state, and the main thread also runs into the `SECONDS.sleep(2)`. Because the other two threads each take 1 second to complete, after 2 seconds, the result is definitely in the `Future` object and can be fetched. If the `SECONDS.sleep(2)` didn't exist, the output would be the same because the `get()` method is blocking and waiting. That is, if the result is in the container, it can be consumed directly, but `get()` blocks until the result is available. Take a second to think about what the program output might look like.

Note

In Spring, when a task needs to be executed, it uses threads from a thread pool instead of creating a new thread every time. This allows for more efficient usage of system resources and can reduce the overhead associated with creating new threads. The thread pool contains a predetermined number of threads that are available for processing tasks. Because the threads are being recycled, it can be difficult to determine whether a new thread or a previously used thread is being used for a specific task. The decision of which thread to assign a task to depends heavily on the current state of the thread pool. If all threads are currently busy, the task will have to wait until a thread becomes available. Once a thread is assigned a task, it will execute the task until completion. After the task has been executed, the thread becomes free again and is put back into the thread pool. At this point, the thread goes into a sleep state, waiting for the next instruction block to be assigned to it. This way, threads are reused efficiently, and the overhead associated with creating new threads is minimized.

Possible Output

The main thread will briskly do all the log output and bump the other threads:

```
… [  main] com.tutego.boot.Sleepyhead    : Before sleep1Seconds()
… [  main] com.tutego.boot.Sleepyhead    : After sleep1Seconds()
… [  main] com.tutego.boot.Sleepyhead    : Before sleep1SecondWithDream()
… [  main] com.tutego.boot.Sleepyhead    : After sleep1SecondWithDream()
… [task-2] com.tutego.boot.SleepAndDream : Starting to dream
… [task-1] com.tutego.boot.SleepAndDream : Going to sleep: SNNNNOOORRRREEEEE
… [task-1] com.tutego.boot.SleepAndDream : Woke up
… [task-2] com.tutego.boot.SleepAndDream : Finished the dream
… [  main] com.tutego.boot.Sleepyhead    : Dream: about unicorns
```

Until the processing and logging in the threads starts, it takes some time. The output could also look different because threads are highly nondeterministic. In the square brackets, you can easily read the thread names: `main`, `task-1`, and `task-2`.

Even if the threads were requested in a fixed order, they need exclusive access to the logger, and then `sleep1Seconds()` or else `sleep1SecondWithDream()` could win. After 1 second, almost both are finished at the same time, so there is again a race to the logger, so the order of the outputs could be different in principle.

4.3.3 The CompletableFuture Return Type

An asynchronous method in Spring returns a `Future` object, and here the `CompletableFuture` implementation is ideal.[20] The `Future` type is boring and has few methods: to fetch the result, to cancel, or to query its status. To obtain the result, we must use the `get(…)` method, as there are no other alternatives available.

An issue with the `Future` data type is that it doesn't support cascading, which means that if the `Future` contains a result, it can't trigger another operation in sequence. Developers familiar with programming in JavaScript may be familiar with the Promise application programming interface (API), which supports cascading through the use of the `then(…)` method, allowing for subsequent operations to be triggered.

The Java library has something similar, namely the extensive data type `CompletableFuture` as an implementation of `Future`. This can also be used to build cascades of operations, trigger background operations, return the results to the series, and so on. `CompletableFuture` is a "better" `Future`, and that is why we use it in the sample program.

The receiver can cascade operations and/or wait for the result with `get()`/`join()`. Schematically, it looks like this:

```
XXX xxx = completableFuture.thenApply(function).join();
```

20 Before Spring Framework 6.0, people used `AsyncResult<V>`, but the type is now *deprecated*.

The `thenApply(…)` method takes the value from the `CompletableFuture`, calls a function with the value, and produces a new element in the pipeline. The `join(…)` method waits until the operation is complete.

Transmit Exceptions

During the execution of a background operation, unforeseen issues may arise, resulting in the throwing of an exception. Because the background operation is executed in a separate thread from the caller thread, any exceptions that occur may go unnoticed, and the background thread will terminate abruptly without returning any results. To prevent this scenario, exceptions must also be reported to the caller thread, allowing for proper handling of the error and ensuring that the client doesn't wait indefinitely for a result.

This is why there is a second static method besides `CompletableFuture.completedFuture(…)`, which is `failedFuture(Throwable ex)`. This way, the exception can be stored in the `CompletableFuture`, and it will show up again later when calling `get(…)` because if an exception is stored, `get(…)` will also throw an exception.

Suppose the thread has reported an exception with `CompletableFuture.failedFuture(…)`. This is how a client can react to the exception:

```
try {
  log.info( "{}", future.get() );
}
catch ( ExecutionException e ) {
  log.error( "{}", e.getClass() );
  // class java.util.concurrent.ExecutionException

  log.error( "{}", e.getCause() );
  // class of the original exception
}
```

On the `Future` object, `get(…)` is called, but a value isn't returned. Instead, the `get(…)` method will throw an `ExecutionException`. This signals that there was an error in the background thread. The error is queried via `getCause(…)`: the `ExecutionException` piggybacks the actual exception, so to speak. Wrapped exceptions occur in many places, for example, when a checked exception is wrapped in a `RuntimeException`.

[+]

Tip

There are two drawbacks to using an asynchronous method that returns `void`. First, the client may not be aware that the operation is asynchronous, which could be intentional or unintentional. Second, and a more significant issue, is that there is no feedback on whether the operation succeeded or failed in the background, as `void` signifies a fire-

and-forget approach where any failures aren't detected. To mitigate these problems, it's advisable to use `Future<Void>` or `CompletableFuture<Void>` when implementing asynchronous methods. This way, the caller can receive feedback on whether there were any exceptions in the background thread and take appropriate action based on the result.

Task: Calculate Thumbnail Concurrently

After so much theory, now follows a practical task for the further development of the Date4u application. In the class `PhotoService`, the `upload(…)` method is to be rewritten in such a way that the preview image is built up in a background thread.

[+]

Tip

If you want to execute something concurrently, the concurrent operation is triggered as early as possible. The program is then busy with something else first, and if the result is needed, the concurrent program is waited for if necessary.

Suggested solution: If thumbnail images are built in the background, the first thing to do is to adjust the `Thumbnail` interface:

```
public interface Thumbnail {
  @Async
  Future<byte[]> thumbnail( byte[] imageBytes );
}
```

Listing 4.9 Thumbnail.java Extension

So, we added `@Async`. There is one peculiarity: if the interface carries `@Async`, this also applies to the implementation. This isn't true for all annotations; some don't carry over to the implementations.

The annotation `@Async` alone isn't enough. The asynchronous method entails further change:

- The return value of the method needs to be wrapped in a `Future` or `CompletableFuture`.
- The client must retrieve the result from the `Future` object.
- The `@Async` annotation must be enabled with a corresponding enabler in the configuration.

To begin, let's modify the implementation of the interface, specifically in `AwtBicubicThumbnail`.

```
@Service
public class AwtBicubicThumbnail implements Thumbnail {
  …
```

```
  @Override
  public Future<byte[]> thumbnail( byte[] imageBytes ) {
    try ( InputStream is = new ByteArrayInputStream( imageBytes );
          ByteArrayOutputStream baos = new ByteArrayOutputStream() ) {
      ImageIO.write( create( ImageIO.read( is ), 200, 200 ), "jpg", baos );
      log.info( "thumbnail" );
      return CompletableFuture.completedFuture( baos.toByteArray() );
    }
    catch ( IOException e ) {
      throw new UncheckedIOException( e );
    }
  }
}
```

Listing 4.10 AwtBicubicThumbnail.java Extension

The byte array is placed into a `Future` object at the end.

Next, let's look at the call site at `PhotoService`. We've said that the background operation should be triggered as early as possible.

```
public String upload( byte[] imageBytes ) {
  Future<byte[]> thumbnailBytes = thumbnail.thumbnail( imageBytes );

  String imageName = UUID.randomUUID().toString();

  // First: store original image
  fs.store( imageName + ".jpg", imageBytes );

  // Second: store thumbnail
  try {
    log.info( "upload" );
    fs.store( imageName + "-thumb.jpg", thumbnailBytes.get() );
  }
  catch ( InterruptedException | ExecutionException e ) {
    throw new IllegalStateException( e );
  }

  return imageName;
}
```

Listing 4.11 PhotoService.java Extension

The generation of the preview image is initiated first in a background thread, and the Universally Unique Identifier (UUID) for the image name can be determined later.

However, it's possible for the background thread to throw an exception, requiring the use of a `try-catch` block. Passing the exception up to the caller would be unfair as it would create a problem for them. In our case, an `InterruptedException` can't occur, but an `ExecutionException` is always thrown when an error occurs in the background thread.

However, the program isn't yet fully functional as we haven't accounted for what happens if the preview image can't be generated. One solution could be to delete the original image as well, within a transaction. In the proposed solution, the exception is passed to the framework without any special handling.

To complete the program, we still need to configure the enabler.

```
@SpringBootApplication
@EnableCaching
@EnableAsync
public class Date4uApplication {
  public static void main( String[] args ) {
    SpringApplication.run( Date4uApplication.class, args );
  }
}
```

Listing 4.12 Date4uApplication.java Extension

Upon program start, the following threads are involved: the Spring Shell calls the method of the shell component, which then calls the `upload(…)` method of the service. This is executed on the main thread. On the other hand, the `thumbnail(…)` implementation is executed by a separate thread, as indicated by the logger. It should be noted that scaling the image is a relatively expensive operation, whereas saving it's generally cheaper because the program usually spends some time in the `get()` method waiting for the thumbnail to be ready.

4.4 TaskExecutor *

The asynchronous operations in the background must be executed by one entity. This is the job of an executor. To abstract this executor, Spring has declared the data type `TaskExecutor`.[21] A `TaskExecutor` inherits from `Executor` in the Java SE library (see Figure 4.3).

From today's perspective, a `TaskExecutor` would not be necessary, but the Java standard library only introduced the `Executor` type in Java 5. The `TaskExecutor` from Spring is older.

21 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/core/task/TaskExecutor.html*

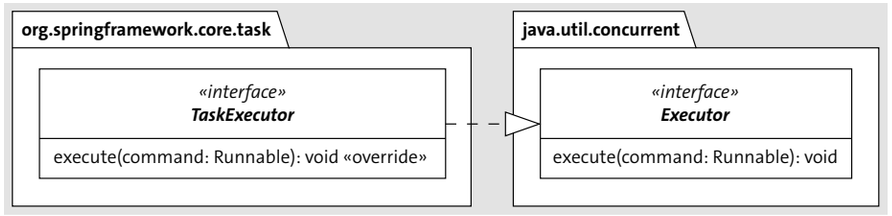


Figure 4.3 "TaskExecutor": The "Runnable" Executor from Spring

Both interfaces declare method `execute(Runnable)`, so they can execute a code block of type `Runnable`. Which strategy is used to execute the `Runnable` is up to the implementation. Often a thread pool is used, but the `Runnable` could also be executed by a chosen thread. This is often used in the GUI environment, for example, where a block of code is to be executed in the GUI thread. It's all a matter of the `TaskExecutor` implementation.

4.4.1 TaskExecutor Implementations

The Spring Framework provides a number of implementations of the `TaskExecutor` interface (see Figure 4.4).

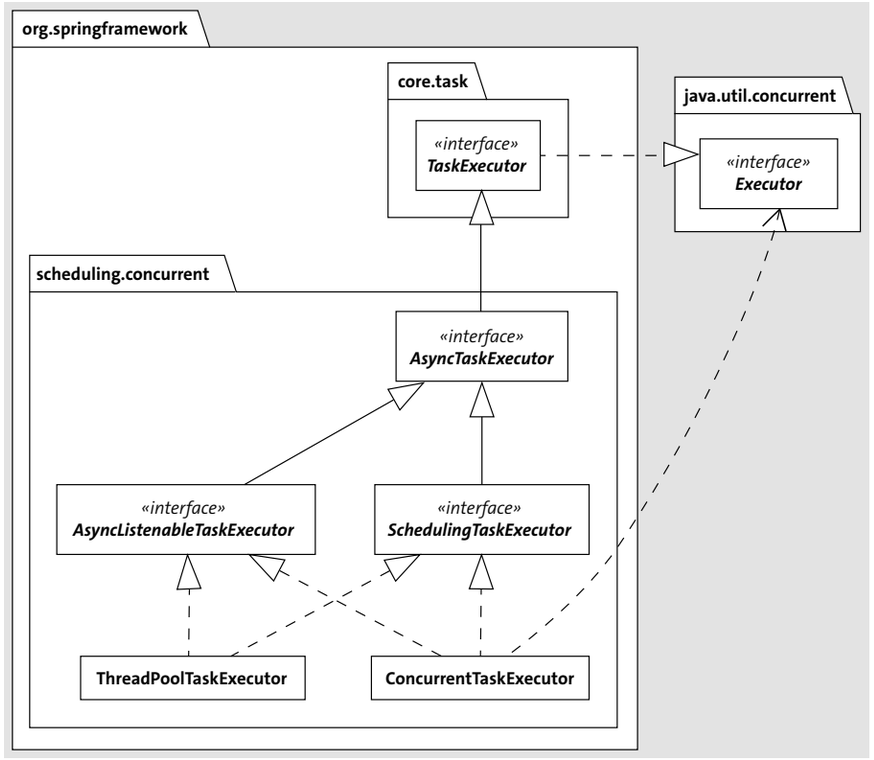


Figure 4.4 "TaskExecutor" Implementations

Let's turn our attention to two implementations: the `ThreadPoolTaskExecutor` and the `ConcurrentTaskExecutor`. The `ThreadPoolTaskExecutor`[22] is a kind of predecessor of the current `java.util.concurrent.ThreadPoolExecutor`,[23] which works internally with a pool of threads. The `ThreadPoolTaskExecutor` is more powerful than the `ThreadPoolExecutor` of Java SE because the Spring thread pool can be reconfigured at runtime. Often, a thread pool is set up early at application startup and runs as long as the application. However, usage might deviate from the planned usage pattern and initial configuration, and then it's useful if, for example, the number of concurrent threads can be changed at runtime. The Java SE thread pool can't do this.

Another useful data type is the `ConcurrentTaskExecutor`, which brings an existing `Executor` implementation from Java SE into the Spring universe. The `ConcurrentTaskExecutor` is an application of the *adapter pattern*, which adapts two incompatible interfaces to each other, with the core functionality being the same. Spring 6.1 introduces support for a `TaskExecutor` featuring virtual threads through the `VirtualThreadTaskExecutor`.

Declare TaskExecutor Beans and Use Them for @Async

When using asynchronous operations with the `@Async` annotation, a `TaskExecutor` can be specified for execution. To accomplish this, a Spring-managed bean must be created and named accordingly:

```
@Bean( "threadPoolTaskExecutor" )
public TaskExecutor myThreadPoolTaskExecutor() {
  ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
  Executor.…
  executor.initialize();
  return executor;
}

@Bean( "concurrentTaskExecutor" )
public TaskExecutor myConcurrentTaskExecutor () {
  return new ConcurrentTaskExecutor( Executors.newFixedThreadPool(3) );
}
```

The name is explicitly set, but, of course, the method name would be fine too.

The first `TaskExecutor` uses the Spring-specific type `ThreadPoolTaskExecutor`, and the second uses the Java SE `ThreadPoolExecutor`, which is adapted into a Spring data type `TaskExecutor` via `ConcurrentTaskExecutor`.

In the second step, the `@Async` annotation specifies the name of the `TaskExecutor` bean:

22 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/concurrent/ThreadPoolTaskExecutor.html*

23 *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadPoolExecutor.html*

```
@Async( "threadPoolTaskExecutor" ) public void abc() { }
@Async( "concurrentTaskExecutor" ) public void xyz() { }
```

This allows the asynchronous calls to be processed with differently configured executors.

4.4.2 Set Executor and Handle Exceptions

When Spring uses a `TaskExecutor` for execution, an exception may occur in the `Runnable`. The question then becomes: What happens to the exception? We've looked at our dream method in Section 4.3.2, which showed that with `@Async` methods, even checked exceptions can be forwarded to the framework.

What should happen with the exceptions that arrive at the framework can be configured via an `AsyncConfigurer`.[24] The interface looks like this.

```
package org.springframework.scheduling.annotation;

import …

public interface AsyncConfigurer {

  @Nullable
  default Executor getAsyncExecutor() { return null; }

  @Nullable
  default AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
    return null;
  }
}
```

Listing 4.13 AsyncConfigurer.java

The `AsyncConfigurer` provides two objects: an `Executor` that executes the code, and an `AsyncUnchaughtExceptionHandler` for the uncaught exceptions. The `AsyncUncaughtExceptionHandler` data type also comes from the Spring environment.

```
package org.springframework.aop.interceptor;

import java.lang.reflect.Method;

@FunctionalInterface
public interface AsyncUncaughtExceptionHandler {
```

24 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/scheduling/annotation/AsyncConfigurer.html*


```
 void handleUncaughtException(Throwable ex, Method method,
                              Object... params);
}
```

Listing 4.14 AsyncUncaughtExceptionHandler.java

In Java SE, there is something similar called `UncaughtExceptionHandler`.[25] The difference between the two is that in the Java SE environment, an `UncaughtExceptionHandler` is used only for unchecked exceptions because everything else has already been caught with `Runnable` according to the API contract (the `run()` method of `Runnable` has no `throws`). Conversely, in the Spring environment, the `AsyncUncaughtExceptionHandler` can handle all exceptions. The default implementation of Spring is the `SimpleAsyncUncaughtExceptionHandler`.[26]

```
public class SimpleAsyncUncaughtExceptionHandler implements AsyncUncaughtExceptionHandler {

    private static final Log logger =
       LogFactory.getLog(SimpleAsyncUncaughtExceptionHandler.class);


    @Override
    public void handleUncaughtException(Throwable ex, Method method,
                                        Object... params) {
        if (logger.isErrorEnabled()) {
            logger.error(
               "Unexpected exception occurred invoking async method: "
              + method, ex);
        }
    }
}
```

Listing 4.15 SimpleAsyncUncaughtExceptionHandler.java

From the implementation, you can see that the exception is logged and nothing else happens.

A Custom AsyncConfigurer Implementation

If a custom executor and an `AsyncUncaughtExceptionHandler` need to be specified, an `AsyncConfigurer` implementation can be defined as a `@Configuration`, which might look like this:

25 *https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.UncaughtExceptionHandler.html*

26 *https://github.com/spring-projects/spring-framework/blob/main/spring-aop/src/main/java/org/springframework/aop/interceptor/SimpleAsyncUncaughtExceptionHandler.java*

```
@Configuration
class AsyncConfig implements AsyncConfigurer {
  private final Logger log = LoggerFactory.getLogger( getClass() );

  @Override
  public Executor getAsyncExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    // …
    return executor;
  }

  @Override
  public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {
    return ( throwable, method, params ) -> {
      log.info( "Exception: {}", throwable );
      log.info( "Method: {}", method );
      IntStream.range( 0, params.length ).forEach(
        index -> log.info( "Parameter {}: {}", index, params[ index ] )
      );
    };
  }
}
```

The `AsyncConfigurer` implementation overrides both `default` methods. For the `Executor`, the `getAsyncExecutor()` method builds a `ThreadPoolTaskExecutor`. `getAsyncUncaughtExceptionHandler()` returns an implementation of the `AsyncUncaughtExceptionHandler` functional interface via a lambda expression. The data received from the parameter list `(Throwable ex, Method method, Object... params)` are all logged.

4.5 Spring and Bean Validation

Validating the state of objects is an essential aspect of software development, as it helps ensure that data is consistent and accurate throughout the application. In this section, we'll explore how to use Spring and Jakarta Bean Validation to implement robust validation rules and improve the reliability of your application.

4.5.1 Parameter Checks

Typical methods always have the same structure: first, they check the validity of parameters, and then only continue if everything is okay. Normally, incorrect parameters are reported by an `IllegalArgumentException`, and sometimes also by a `NullPointerException`. In the following, we see two examples from Java SE:

- **Example 1:** `ArrayList/get(…)`

```
public E get(int index) {
  Objects.checkIndex(index, size);
  return elementData(index);
 }
```

The index for the access must not be negative and not too large. `elementData(…)` is only called when the parameter assignment is okay, and the result is returned.

- **Example 2:** `LocalDate.of(…)`

```
public static LocalDate of(int year, Month month, int dayOfMonth) {
  YEAR.checkValidValue(year);
  Objects.requireNonNull(month, "month");
  DAY_OF_MONTH.checkValidValue(dayOfMonth);
  return create(year, month.getValue(), dayOfMonth);
 }
```

Parameter checking can also be found in constructors and factory methods, for example, in the `of(…)` method of `LocalDate`.

Often, methods are divided into two sections: validation of parameter assignments at the top, followed by the actual logic that can assume that all parameter variables are correct. However, if all validations are placed at the beginning of the method, it may be worth considering whether certain checks can be deferred to the framework. This is because anything that occurs at the beginning or end of a method can be handled by a proxy.

4.5.2 Jakarta Bean Validation (JSR 303)

Jakarta Bean Validation is a standard with two parts at its core: annotations used to describe valid states, and a validator used to check the rules. Jakarta Bean Validation provides a number of basic annotation types, for example:

- `@Size`
- `@NotNull`, `@Null`
- `@Min`, `@Max`
- `@Pattern`
- `@Past`, `@Future`

New annotation types can be declared, which in turn can validate their own rules. The annotations are valid on instance variables, setters/getters, or method parameters.

4.5.3 Dependency on Spring Boot Starter Validation

Although the Spring Framework relies on validation in many places, the core starter doesn't include an implementation. If we want to use validation, an additional dependency is required.

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

Listing 4.16 pom.xml Extension

The dependency contains the core starter and, as a reference implementation of the Jakarta Bean Validation, the *Hibernate Validator* (*https://github.com/hibernate/hibernate-validator*). Spring Boot 3 uses the latest Hibernate Validator 8 (Jakarta Bean Validation 3), whereas Spring Boot 2.7 uses only Hibernate Validator 6 (Jakarta Bean Validation 2).

The Hibernate Validator serves as the default implementation in Jakarta Enterprise Edition, which comes with its own set of benefits. First, it provides additional validation capabilities through proprietary annotation types, which allows developers to create more robust validation rules. Second, the use of the Hibernate Validator isn't limited to just Spring applications, as it's compatible with other enterprise frameworks as well. Therefore, developers can use the same validation framework across multiple projects and easily port Spring applications to other frameworks without any issues.

4.5.4 Photo with Jakarta Bean Validation Annotations

The Date4u application relies on various datasets such as login information, profile information, likes, and more to function as a dating application. Among these, the `Photo` class is used to store information about photos, which was introduced in Section 4.2.9. To ensure that invalid photo information can be easily detected, we can annotate the `Photo` datatype with validation annotations.

```
import jakarta.validation.constraints.*;
import java.time.LocalDateTime;

public class Photo {

  public Long id;
  @Min( 1 ) public long profile;
  @NotNull @Pattern( regexp = "[\\w_-]{1,200}" ) public String name;
  public boolean isProfilePhoto;
  @NotNull @Past public LocalDateTime created;

  public Photo() { }
  public Photo( … ) { … }
  @Override public String toString() { … }
}
```

Listing 4.17 Photo.java Extension

These annotations are applied to instance variables, and they are also valid for setters and getters. Let's review the configuration settings:

- **`profile`**
  The `long` variable stores the foreign key on a corresponding profile. The annotation states that the number must always be greater than or equal to `1`. As a primitive data type `long`, the variable can never be `null`.
- **`name`**
  The name of the photo is a `String` and must be given, that is, not `null`. The regex expression dictates that any word character, underscore character, or minus character is allowed, and the length must be between 1 and 200. The word characters include all digits and all uppercase/lowercase letters—but only the letters of the English alphabet from A to Z without special characters. Certainly, file names are likely to contain additional characters in practice. We should change this in a larger example.
- **`created`**
  For `LocalDateTime`, the value should not be `null` and must be in the past because photos can't be created in the future.

If the photo is a profile photo, the `boolean` variable `isProfilePhoto` is set. Here you can't validate anything because an annotation is missing.

The `id` instance variable of the `Photo` class is left without any validation annotation, as its assignment is dependent on the state of the object. A newly created `Photo` object doesn't have an assigned ID, which happens only after it's saved to the database. An ID assigned to a saved photo must be a number greater than or equal to 1. The Jakarta Bean Validation framework provides groups to handle cases where values must be validated differently in different scenarios.

Note

The annotation `@NotNull` is a necessary addition to `@Pattern` and `@Past`, because if the variable is `null`, the other checks aren't made.

We added a parameterized and parameterless constructor to `Photo` to make it easier to build the `Photo` object, as well as a `toString()` method. But this has nothing to do with validation.

Note

The annotation `jakarta.validation.constraints.`**`NotNull`** (for validation proxies) can easily be confused with `org.springframework.lang.`**`NonNull`** or `javax.annotation.`**`Nonnull`** (for static analysis tools).

The class in question doesn't possess any annotations, particularly the Spring annotation `@Component` or any similar ones. This object isn't meant to function as a Spring component, but instead serves as a simple container. Typically, these objects don't perform any significant actions and lack domain model intelligence or critical behavior. Rather, they exist solely for data exchange purposes, functioning as data classes.

IntelliJ Tip

IntelliJ Ultimate Edition has a special view for validators, which can be activated under **View • Tool Windows • Bean Validation**.

4.5.5 Inject and Test a Validator

Validation of objects can be done in two ways: declaratively or programmatically. We'll begin by exploring the programmatically driven approach.

Spring Boot automatically configures a Spring-managed bean of type `Validator`,[27] which we can inject:

```
@Autowired Validator validator;
```

The `Validator` interface focuses on `validate*(…)` methods, such as Figure 4.5 shows.

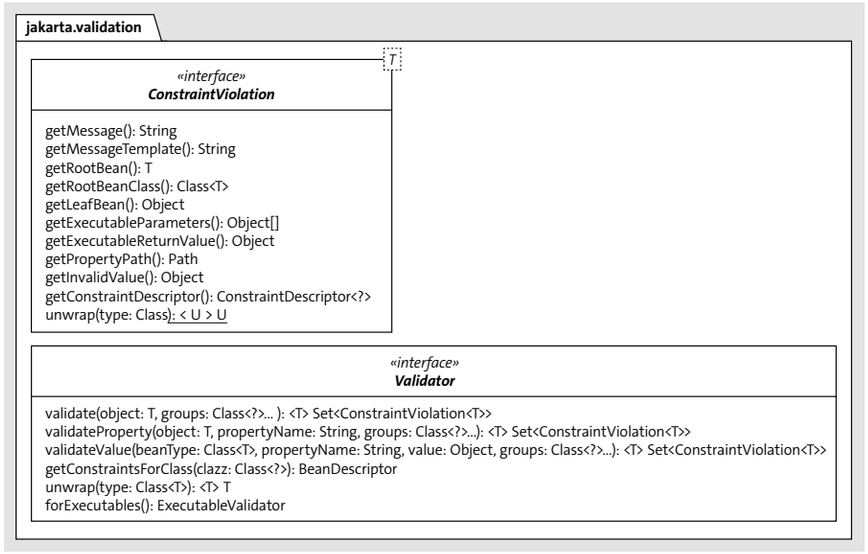


Figure 4.5 "Validator" Triggering Validation and "ConstraintViolation" Representing Validation Error

27 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/Validator.html*

When invoking the `validate(…)` method, the object that needs to be validated is supplied as an argument. The resulting `Set` contains `ConstraintViolation`[28] objects, which indicate a problem if the set isn't empty. Conversely, an empty set implies that there are no validation errors.

Assume that a photo of the shell is to be entered into the database:

```
@Autowired Validator validator;

@ShellMethod( "Insert a new photo" )
String insertPhoto( Long id, long profile, String name,
                    boolean isProfilePhoto, String created ) {

  Photo photo = new Photo( id, profile, name, isProfilePhoto,
                           LocalDateTime.parse( created ) );

  Set<ConstraintViolation<Photo>> violationSet = validator.validate( photo );

  // Insert photo into database if violationSet.isEmpty()

  return violationSet.isEmpty() ? "Photo inserted"
                                : "Photo not inserted\n" + violationSet;
}
```

This reference to the `Validator` is used to perform the validation of the `Photo` object. The result of the validation is stored in a `Set` of `ConstraintViolation` objects. If the set is empty, there are no violations of the validation rules, and, theoretically, the photo can be inserted into the database. If there are validation errors, a different string is returned along with the details of the violations.

Let's call the shell method once with correct and with broken data:

```
shell:>insert-photo 12 1 bella false 2020-09-01T12:20
Photo inserted
shell:>insert-photo 12 0 üüüüä false 2222-09-01T12:20
Photo not inserted:
[ConstraintViolationImpl{interpolatedMessage=
'must be greater than or equal to 1', ↩
propertyPath=profile, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
messageTemplate='{jakarta.validation.constraints.Min.message}'}, ↩
ConstraintViolationImpl{interpolatedMessage='must match "[\w_-]{1,200}"', ↩
propertyPath=name, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
```

28 *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraintviolation*

```
messageTemplate='{jakarta.validation.constraints.Pattern.message}'}, ↩
ConstraintViolationImpl{interpolatedMessage='must be a past date', ↩
propertyPath=created, rootBeanClass=
class com.tutego.date4u.interfaces.shell.Photo, ↩
messageTemplate='{jakarta.validation.constraints.Past.message}'}]
```

In the second string, the `Validator` finds a number of problems:

- `profile` must not be 0 because, according to the rule, foreign keys must always be greater than or equal to 1.
- The `name` contains German umlauts. This is wrong because `\w` only covers uppercase/lowercase letters and digits, but not German umlauts.
- The timestamp `created` isn't in the past.

The `toString()` representation may of course be different depending on the implementation, but at first glance, more details become clear:

- **`interpolatedMessage`**
  This is an automatically generated error message in the respective local language.
- **`propertyPath`**
  This is the originator; in our case, these are `profile`, `name`, and `created`.
- **`rootBeanClass`**
  This is the `Photo` class with the invalid properties.
- **`messageTemplate`**
  This is the key for a translation table.

In summary, when a Jakarta Bean Validator implementation is present in the classpath, Spring auto-configuration constructs a `Validator` as a managed bean that can be injected by a client for use. The `Validator` includes a `validate(…)` method that returns a `Set<ConstraintViolation>` rather than an exception.

It all depends on how we utilize this information. Certain web frameworks, such as Jakarta Faces, are capable of directly binding the error messages to the GUI.

4.5.6 Spring and the Bean Validation Annotations

In addition to programmed validation, there is another way to use Jakarta Bean Validation through declarative validation using proxies that verify the validity of objects or parameters. This can be particularly useful in the `download(…)` method of the `PhotoService`, where only valid photos should be accepted.

```
@Service
@Validated
public class PhotoService {
```



```
  public Optional<byte[]> download( String imageName ) { … }

  public Optional<byte[]> download( @Valid Photo photo ) {
    return download( photo.name );
  }

  // …
}
```

Listing 4.18 PhotoService.java Extension

Two annotations are used:

- `@Validated`[29] from the `org.springframework.validation.annotation` package leads to a new proxy that works internally with a `Validator`.
- `@Valid`[30] from the `jakarta.validation` package before the parameter tells the proxy that a valid object must be passed; the type comes from Jakarta Bean Validation. If the annotation is missing, nothing is validated because it's fine in other scenarios to get empty photos and then initialize and fill them, for example.

If a program injects a `PhotoService`, the result is a validation-enabled proxy:

```
@Autowired PhotoService photoService;
```

The `download(Photo)` method undergoes a validation check through a proxy when called on this object. The proxy verifies if the photo obeys the rules specified for its validity. If the photo passes the validation, the proxy allows the caller to access the core logic. However, if the photo fails to meet the validation rules, a `ConstraintViolationException`[31] is thrown by the proxy.

This is how the call could look with a logging of the violations:

```
try {
  Photo photo = new Photo();
  photoService.download( photo );
}
catch ( ConstraintViolationException e ) {
  Set<ConstraintViolation<?>> violations = e.getConstraintViolations();
  log.info( violations.toString() );
}
```

29 *https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/validation/annotation/Validated.html*
30 *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/valid*
31 *https://jakarta.ee/specifications/bean-validation/3.0/apidocs/jakarta/validation/constraintviolationexception*

The newly created `Photo` object violates various rules, so the proxy validator will report the problem and throw a `ConstraintViolationException`—the unchecked exception type comes from Jakarta Bean Validation, not Spring. The `ConstraintViolationException` object returns the known set of small `ConstraintViolation` objects with `getConstraintViolations()`.

4.5.7 Bean Validation Annotations to Methods

Annotations from Jakarta Bean Validation can be directly used on both return values and parameters. Consider the following example:

```
@Service
@Validated
class NiceService {

  public @NotNull Object aValidMethod( @NotNull String param1,
                                       @Max(10) int    param2 ) {
    return "";
  }

}
```

A class such as `Photo` with annotated properties isn't strictly necessary, as the example shows: the proxy validator will check that the method doesn't return `null`, `arg1` wasn't `null` when called, and the values at `arg2` are at most 10.

Note

It's important to note that object validation only occurs through the proxies. If a program doesn't go through the proxies, for example, a self-programmed test case without Spring's testing infrastructure, the validation won't be executed. Therefore, if validation is critical to the application, it's recommended to program the validation check at the beginning of each method, regardless of whether it's also being checked by the proxy. This approach ensures that the validation is always performed, no matter which calling method is used.

4.5.8 Validation Everywhere Using a Configuration Example

Jakarta Bean Validation can be found in many places: in Jakarta EE as well as in the Spring environment. Popular examples are the *Jakarta Persistence API* for object-relational mapping, *Jakarta Faces* for dynamic web pages, or in *Jakarta RESTful Web Services*.

Spring Boot can also check configuration properties passed to an object via `@ConfigurationProperties` using Jakarta Annotations.

The Project Lombok and Jakarta Bean Validation technologies lead to the following compact example, where `GeometryProperties` references a `Box` and a `Circle` with data:

```
@Component @ConfigurationProperties( "app" ) @Data @Validated
class GeometryProperties {
  @NotNull @Valid private GeometryProperties.Box    box;
  @NotNull @Valid private GeometryProperties.Circle circle;

  @Data
  public static class Box {
    @Min( 50 ) @Max( 1000 ) private int width;
    @Min( 50 ) @Max( 600 )  private int height;
  }

  @Data
  public static class Circle {
    @Min( 10 ) @Max( 1000 ) private int radius;
  }
}
```

The `@Data` annotation[32] comes from *Project Lombok* (*https://projectlombok.org*). The open-source software is in essence a compiler hack that automatically puts setters, getters, parameterized constructors, loggers, and so on into the bytecode. This is always used if you want to save setter/getter and other "boilerplate code," but it's necessary in the bytecode—just for the `@ConfigurationProperties`.

If `@Data` is applied, you can imagine setter/getter in the following:

- **`GeometryProperties`**: `setBox(Box)`, `getBox()`, `setCircle(Circle)`, `getCircle()`
- **`Box`**: `setWidth(int)`, `getWidth()`, `setHeight(int)`, `getHeight()`
- **`Circle`**: `setRadius(int)`, `getRadius()`

The `GeometryProperties` object must reference `Box` and `Circle` because validation doesn't allow `null` assignments with `@NotNull`. In the subobjects, the properties would have to be in certain value ranges.

When Spring Boot has read the `Environment` data and transferred it to the bean, validation follows. If there is an error, the container doesn't start and reports the problem.

4.5.9 Test Validation *

Suppose we want to test the `download(Photo)` method. The test should show that there is no exception for a correct photo and that a broken, not valid photo leads to an exception and setting special error conditions.

32 *https://projectlombok.org/features/Data*

The class `PhotoServiceTest` will get a nested class for the validator-related tests, and two new methods—`photo_is_valid()` and `photo_has_invalid_created_date()`—will check that an error-free `Photo` doesn't lead to an exception, but a broken photo does:

```
@SpringBootTest
class PhotoServiceTest {

  …

  @Nested
  class Validator {

    @Test
    void photo_is_valid() {
      Photo photo = new Photo( 1, 1, "fillmorespic", false,
                               LocalDateTime.MIN );
      assertThatCode( () -> photoService.download( photo ) )
          .doesNotThrowAnyException();
    }

    @Test
    void photo_has_invalid_created_date() {
      …
    }
      …
    }
  }
}
```

The first method, `photo_is_valid()`, is simple: it builds a `photo` with valid states and then passes it to the method `download(Photo)`. Actually, the test method could end here, but `assertThatCode(…).doesNotThrowAnyException()` makes it more explicitly readable that no exception must occur.

Note

The `PhotoService` will use `FileSystem`, and the `FileSystem` must also return something. Therefore, it's important that Mockito has the appropriate objects built correctly. How to build mock objects was shown in Chapter 3, Section 3.8.5.

The second method, `photo_has_invalid_created_date()`, shows that the validator reports a certain error in case of an incorrect date—which isn't in the past but in the future:

```
LocalDateTime future = LocalDateTime.of( 2500, 1, 1, 0, 0, 0 );
Photo photo = new Photo( 1L, 1L, "fillmorespic", false, future );
assertThatThrownBy( () -> photoService.download( photo ) )
```



```
  .isInstanceOf( ConstraintViolationException.class )
  .extracting(
    throwable ->
      ((ConstraintViolationException) throwable).getConstraintViolations(),
    as( InstanceOfAssertFactories.collection( ConstraintViolation.class ) )
  )
  .hasSize( 1 )
  .first( InstanceOfAssertFactories.type( ConstraintViolation.class ) )
  .satisfies( vio -> {
    assertThat( vio.getRootBeanClass() ).isSameAs( PhotoService.class );
    assertThat( vio.getLeafBean() ).isExactlyInstanceOf( Photo.class );
    assertThat( vio.getPropertyPath()).hasToString("download.photo.created");
    assertThat( vio.getInvalidValue() ).isEqualTo( future );
    // assertThat( vio.getMessage() ).isEqualTo( … );
  } );
```

While the readability may not be exceptional, the code example effectively showcases the capabilities of AssertJ. It's worth noting that using AssertJ doesn't always guarantee optimal readability, but it does provide a powerful toolset for writing comprehensive and expressive assertions.

The photo in year 2500 will result in the following:

1. A `ConstraintViolationException` will occur.
2. The set returned by `getConstraintViolations()` will contain exactly one element.
3. The element will be of type `ConstraintViolation`.
4. Of these, `getRootBeanClass()` will return a `PhotoService`, `getLeafBean()` will return `Photo`, and so on.
5. Because the message is always translated, that is, localized, this is a disadvantage when testing, and therefore no testing is done.

In practice, you would omit certain checks. For example, it's clear that `getConstraintViolations()` only contains `ConstraintViolation` objects, so the validation implementation must take care of that. The rule is "never test framework code."

Refactoring the Photo Class

Because we'll later use the `Photo class` as an entity bean, we refactor it so that the instance variables become private and end up with setters/getters:

```
public class Photo {

  private Long id;

  @Min( 1 )
  private Profile profile;
```

```
  @NotNull @Pattern( regexp = "[\\w_-]{1,200}" )
  private String name;

  private boolean isProfilePhoto;

  @NotNull @Past
  private LocalDateTime created;

  protected Photo() { }

  public Photo( Long id, Profile profile, String name, boolean isProfilePhoto,
LocalDateTime created ) {
    this.id = id;
    this.profile = profile;
    this.name = name;
    this.isProfilePhoto = isProfilePhoto;
    this.created = created;
  }

  public Long getId() {
    return id;
  }

  public Profile getProfile() {
    return profile;
  }

  public String getName() {
    return name;
  }

  public void setName( String name ) {
    this.name = name;
  }

  public boolean isProfilePhoto() {
    return isProfilePhoto;
  }

  public void setProfilePhoto( boolean profilePhoto ) {
    isProfilePhoto = profilePhoto;
  }

  public LocalDateTime getCreated() {
```



```
    return created;
  }

  public void setCreated( LocalDateTime created ) {
    this.created = created;
  }

  @Override public String toString() {
    return "Photo[" + id + "]";
  }
}
```

This makes the `toString()` method a little simpler.

4.6 Spring Retry *

In today's interconnected world, most services are connected remotely, such as databases, messaging systems, and email servers. However, the connection to these servers can be unreliable, and the system may become unavailable. In such a scenario, simply timing out the program isn't a reliable solution as we expect some *resilience* from our system. To address this problem, one possible solution is for the client to wait a bit and then try again, as the network glitch may be resolved by then.

4.6.1 Spring Retry Project

In the Spring ecosystem, various approaches exist to tackle a problem. In this section, we'll explore a compelling proxy from the *Spring Retry project* (*https://github.com/*

spring-projects/spring-retry) that can rerun code blocks in case of exceptions. This feature proves beneficial for services that rely on network connections, as the network can experience brief outages due to topology changes or routing table adjustments.

To deploy the Spring Retry project, two dependencies are placed in the POM:

```
<dependency>
  <groupId>org.springframework.retry</groupId>
  <artifactId>spring-retry</artifactId>
</dependency>

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
</dependency>
```

The first dependency references the Spring Retry project itself. The second dependency is from the area of AOP and introduces *Spring Aspects* so that proxies can be generated.

Job: Determine a Random Image via a Web Service

The *Random User Generator* is a website and web service (*http://randomuser.me*) that generates different sets of personal information. For instance, we aim to create a simple program that requests random images. To achieve this, we can customize the web service using parameters: *https://randomuser.me/api/?inc=picture&noinfo*. The API endpoint is available through both HTTP and HTTPS.

A call to the URL in the browser returns the following (formatted) JSON document:

```
{
  "results": [
    {
      "picture": {
        "large": "https://randomuser.me/api/portraits/men/0.jpg",
        "medium": "https://randomuser.me/api/portraits/med/men/0.jpg",
        "thumbnail": "https://randomuser.me/api/portraits/thumb/men/0.jpg"
      }
    }
  ]
}
```

Behind `results` is an array, and because only one picture was requested, there is only one `picture`. The picture is available in three different resolutions: `large`, `medium`, and `thumbnail`.

The image can be queried for one gender; then the query parameter `gender` can be supplemented with `female` or `male`:

- *https://randomuser.me/api/?inc=picture&noinfo&gender=female*
- *https://randomuser.me/api/?inc=picture&noinfo&gender=male*

A Java program is to call this endpoint and obtain the result. It should consider that there may be connection errors. In case of exceptions, Spring Retry should automatically invoke the web service again.

4.6.2 @Retryable

When attempting to retrieve a string containing JSON, the new `RandomPhoto` component throws an exception, simulating network errors.

```
@Component
class RandomPhoto {

  private final Logger log = LoggerFactory.getLogger( getClass() );
  private int attemptCounter = 1;

  @Retryable
  public String receive( String gender ) throws IOException {
   log.info( "Attempts: {}", attemptCounter++ );
   if ( attemptCounter <= 3 )
      throw new IOException( "Not yet ready to serve" );
   var url="https://randomuser.me/api/?inc=picture&noinfo&gender="+gender;
   try ( var inputStream = URI.create( url ).toURL().openStream() ) {
     return StreamUtils.copyToString( inputStream, StandardCharsets.UTF_8 );
   }
  }
}
```

Listing 4.19 RandomPhoto.java

The public `receive(…)` method is annotated with `@Retryable`,[33] and this tells the retry proxy to call the method again if an exception occurred. The method expects a gender as a parameter, and the return is the string with the JSON document.

The code takes three tries for a random image; thus, we simulate network/service failures. The client won't notice any of this.

The actual call of this web service is very simply implemented via the `URL` class. There are various classes for calling web services, such as the `HttpClient` of Java SE and the `WebClient` or the `RestTemplate` of Spring, but this is actually one with the shortest code.

33 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Retryable.html*

Via the `URL` class, the program gets an `InputStream`, and the Spring utility class `StreamUtils` receives the data via `copyToString(…)` and puts it into a string.

@EnableRetry

For the `RandomPhoto` component, a retry proxy must be enabled that responds to `@Retryable`. For this purpose, `@EnableRetry`[34] is set to a `@Configuration` (like `@SpringBootApplication`). This is how it might look:

```
@Configuration
@EnableRetry
public class RetryConfig { }
```

This builds up corresponding retry proxy objects. If the client calls the `receive(…)` method via the retry proxy, the proxy intercepts the annotated methods and repeatedly goes to the core in case of exceptions.

Watching the Proxy Repeats

We can inject `RandomPhoto` into a client and get a retry proxy:

```
@Autowired RandomPhoto randomPhoto;
```

In the retry proxy, there is an imagined `try-catch` block that goes to the core. In our case, the core throws an exception three times. That is, the `catch` block of the retry proxy catches the exception, and the proxy calls the core again.

Let's obtain the image:

```
log.info( "Before receive()" );
String json = randomPhoto.receive( "male" );
log.info( "After receive(): {}", json );
```

No exception appears on the console, and the log indicates that the retry proxy goes to the core several times:

```
… 18:39:58.555  INFO 22222 --- [main] com.….RetryTest : Before receive()
… 18:39:59.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 1
… 18:40:00.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 2
… 18:40:01.000  INFO 22222 --- [main] com.….RandomPhoto : Attempts: 3
… 18:40:02.000  INFO 22222 --- [main] com.….RetryTest : After receive(): ↩
{"results":[{"picture":{"large":"https://randomuser.me/api/portraits/…
```

The outputs show the three attempts at retrieving the image. The client doesn't notice the repeated calls. After the third attempt, the operation succeeds, and the `receive(…)` method in the core outputs a JSON string as a response—unless the service is actually unreachable.

There is an interesting detail that can be read in the log output. It's noticeable from the times that the retry proxy waits 1 second between each method call. Is this a coincidence?

34 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/EnableRetry.html*

Annotation Type @Retryable

There are several ways to configure `@Retryable` because this annotation type has many annotation attributes:

```
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Retryable {
    String recover() default "";
    String interceptor() default "";
    @AliasFor("include") Class<? extends Throwable>[] retryFor() default{};
    @AliasFor("exclude") Class<? extends Throwable>[] noRetryFor() default{};
    Class<? extends Throwable>[] notRecoverable() default {};
    String label() default "";
    boolean stateful() default false;
    int maxAttempts() default 3;
    String maxAttemptsExpression() default "";
    Backoff backoff() default @Backoff();
    String exceptionExpression() default "";
    String[] listeners() default {};
}
```

Each annotation attribute has a default value, including `maxAttempts`. This attribute determines the number of retry attempts that the retry proxy should make before aborting. The default value for `maxAttempts` is `3`. In the previous example, three retries were attempted because this is the default value. If the number of retries in our example had been set to `4`, the proxy would have thrown an exception.

The number can be set to an `int` via the `maxAttempts` annotation attribute or written as a SpEL expression via `maxAttemptsExpression`.

The annotation type `@Retryable` has an annotation attribute `backoff default @Backoff()` that describes what should happen after an error.

@Backoff: How to Continue after the Error

If the core could not deliver due to an exception, the annotation `@Backoff`[35] tells after which delay it should continue and how. Here, too, there are many setting options:

35 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Backoff.html*

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface Backoff {
    @AliasFor("delay") long value() default 1000;
    @AliasFor("value") long delay() default 1000;
   String delayExpression() default "";
    long maxDelay() default 0;
    String maxDelayExpression() default "";
    double multiplier() default 0;
    String multiplierExpression() default "";
    boolean random() default false;
}
```

The interesting part is a `delay` (alias for `value`), and this specifies a delay of 1,000 milliseconds, or 1 second, by default. We could see this in the log output: when the retry proxy catches the exception, it waits 1 second before going back to the core. The delay can also be written as a SpEL expression.

[+]

@Retryable Example with Exceptions and @Backoff

An adjustment of the default values might look like this:

```
@Retryable(
  value       = { TypeOneException.class, TypeTwoException.class },
  maxAttempts = 4,
  backoff     = @Backoff( 2000 /* ms */ )
)
```

It's essential to explicitly mention the exceptions that the retry proxy should handle. In this case, the proxy should respond to the fictional exceptions `TypeOneException` and `TypeTwoException`. Otherwise, the proxy will repeat the code block for every exception, which isn't desirable. For instance, it's unlikely that a `NullPointerException` should trigger code repetition.

4.6.3 Fallback with @Recover

If the call works after as many attempts as defined in `maxAttempts`, there is a happy ending. If the retry proxy fails after the `maxAttempts` number of attempts, a "rescue method" can return a default value. This method is annotated with `Recover`.[36]

In the following, the `@Retryable` method will always fail—it will result in a call to `recover(…)`:

36 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/annotation/Recover.html*


```
@Retryable
public String read() throws IOException {
  throw new IOException( LocalTime.now().toString() );
}

@Recover
public String recover( IOException e ) {
  System.err.println( "Aarrrrg: " + e );
  return …
}
```

If the method in the core always fails, the method annotated with `@Recover` is called at the very end as a last resort. This method, also called a *fallback* method, receives the exception and returns to the client the result that the `@Retryable` method could not return. The name of the method is irrelevant.

[«]

Note

The `@Recover` method must have the identical return type as the `@Retryable` method because the `@Recover` method returns the result that `@Retryable` could not return.

What Does an @Recover Method Return?

An interesting question arises when deciding whether to write methods that conceal the fact that an operation could fail. This can be problematic, especially if remote calls are transparent to the caller. If the caller isn't aware of potential network exceptions, it can lead to unexpected behavior and errors.

While exceptions are a possible solution, it's worth considering other alternatives as well:

- The return type could be an `Optional` or an empty data structure to prevent the receiver from receiving any data in case of a system error.
- Another option is to fall back on cached values. For example, if the program is meant to load a random image but fails, it could retrieve the last successful image from a cache.
- Default values can also be provided to handle failures. In the case of random images, a default value of a URL to an image with ID 0 could be returned.

Ultimately, the decision of how to handle failure depends on the specific use case and requirements of the program.

The solution with the default random image could be programmed in code like this in a `@Recover` method:

```
@Recover
public String recoverFromRandomUserMeConnectionProblems() {
  return """
      {"results":[{"picture":{
      "large":"https://randomuser.me/api/portraits/lego/1.jpg"
      "medium":"https://randomuser.me/api/portraits/med/lego/1.jpg"
      "thumbnail":"https://randomuser.me/api/portraits/thumb/lego/1.jpg"
      }}]}""";
}
```

4.6.4 RetryTemplate

So far, we've explored the declarative approach that employs annotations to generate a proxy in the background. However, it's also possible to implement code repetition on errors without annotations. This approach simplifies the process of repeating arbitrary code blocks without additional steps.

The focus is on the class `RetryTemplate`[37] and the `execute(…)` method that executes the code block. Internally, the retry proxy also works with the `RetryTemplate`.

Schematically, the usage looks like this:

```
RetryTemplate retryTemplate = new RetryTemplate();
retryTemplate.execute( context -> {
    …
    return result;
} );
```

The code block is of type `RetryCallback`[38] and a functional interface:

```
public interface RetryCallback<T> {
  T doWithRetry( RetryContext context ) throws Throwable;
}
```

The framework passes a `RetryContext` object to the method and returns a result. Any exceptions can be thrown—you can't get higher in the exception inheritance hierarchy than `Throwable`.

The `RetryTemplate` class has an overloaded `execute(…)` method that includes an additional parameter for accepting an object of type `RecoverCallback`. This allows options such as the number of retries to be set.

37 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/support/RetryTemplate.html*

38 *https://docs.spring.io/spring-retry/docs/api/current/org/springframework/retry/RetryCallback.html*

RetryTemplate Builder

A more user-friendly configuration option is available through the `RetryTemplate` builder. Here is an example of how it can be used:

```
RetryTemplate tryTo = RetryTemplate.builder()
                        .infiniteRetry()
                        .retryOn( IOException.class )
                        .uniformRandomBackoff( 1000 /* ms */, 5000 /* ms */ )
                        .build();
tryTo.execute( context -> {
  return …;
} );
```

The `RetryTemplate` has a static `builder()` method that allows you to parameterize the `RetryTemplate` and build it using `build()`. The fluent API is elegant. For example, `infinityRetry()` can be used to configure an unlimited number of retries, while `retryOn(…)` defines which exceptions the `RetryTemplate` should respond to. `uniformRandomBackoff(…)` configures a random wait time between 1 and 5 seconds.

At the end, there is a `RetryTemplate`, and the `execute(…)` method executes the code block.

4.7 Summary

In the world of wiring, we often encounter proxies that wrap themselves around our objects, rather than instances of our own classes. This chapter has delved into the important proxies in practical use, with a brief touch on Jakarta Bean Validation. Although not the most exciting read, it's worth familiarizing oneself with the comprehensive standard, which can be found at *https://jakarta.ee/specifications/bean-validation/3.0/jakarta-bean-validation-spec-3.0.html*. For a more practical take, the documentation of the reference implementation of Hibernate at *https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/* is recommended.

Caching is another extensive topic that can't be avoided when dealing with caching implementations. Factors such as memory size, usage patterns, and access times become crucial in this area. Furthermore, distributed caching is a fascinating topic that holds significant importance in the cloud environment. As you continue your exploration of these topics, keep an eye out for these crucial aspects.

Contents

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

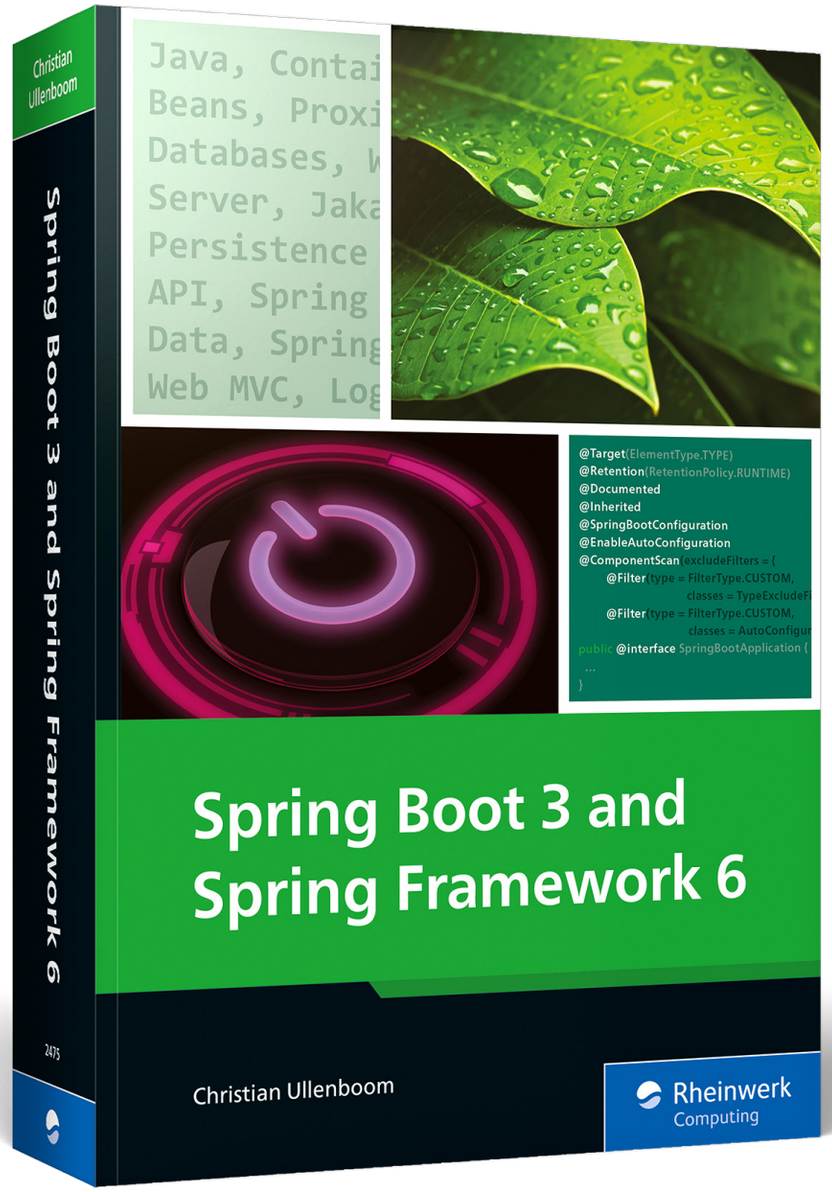
U

V

W

X

Z



Christian Ullenboom is an Oracle-certified Java programmer and has been a trainer and consultant for Java technologies and object-oriented analysis and design since 1997.

Christian Ullenboom

Spring Boot 3 and Spring Framework 6

934 pages | 10/2023 | $59.95 | ISBN 978-1-4932-2475-3

www.rheinwerk-computing.com/5764