Java SE utils

Rudy De Busscher

Version 1.1.0, 20/03/2022

Table of Contents

Release notes

1.1.0

- Added `ResourceUtil.getResources` method to determine URI for a resource.
- `CollectionUtils.iteratorToIterable`

1.0.0

- Removed support for Java 7. (this version is tested on Java 8 and Java 11)
- Use of JUnit 5 for testing
- Fixed bug in FileResourceReader when using file:
- added AnnotationUtil.findAnnotatedFields()

Breaking changes

- Removal of Base64Codec, please us JDK `Base64`.

0.9.3.1

- Addition of Base32 encoding/decoding.

0.9.3

- Addition of a, extensible, resource API which can open a resource in classpath, URL target and file.
- ResourceScanner, find resources on classpath
- AnnotationUtil.
- @SecurityReview to indicate special attention for the developer related to some possible security issues when not correctly used.

0.9.2

- Added methods taking byte[] for Base64 encode/decoding
- Added HEX encoding / decoding
- Added ByteSource, taken byte[] representation of some classes and create encoded (BASE64, HEX) versions
- isEmpty() on char[]
- Define outcome of CDICheck.withinContainer by defining system JVM parameter atbash.utils.cdi.check.

0.9.1

- Renamed StringUtils#join to StringUtils.toDelimitedString
- Added TestReflectionUtils
- Added CDICheck.withinContainer
- Added ProxyUtils

Breaking changes

- Some methods of *be.atbash.util.StringUtils* have changed parameters or are renamed.

0.9.0

- Initial version with classes from various sources usable in any environment.

Setup

Add the following artifact to your maven project file

```
<dependency>
    <groupId>be.atbash.utils</groupId>
    <artifactId>utils-se</artifactId>
    <version>0.9.2</version>
</dependency>
```

Requirements

Runs on JDK 8 and JDK 11 ClassPath.

Features

Utilities compiled for Java 8 and does not have any dependencies (only using SLF4J api).

The most important methods are described here. For the full list of methods, see the Javadoc or the source code.

HEX

Converts to and from HEX encoded Strings.

```
byte[] be.atbash.util.codec.Hex.encodeToString(byte[])
```

Converts a byte array to a HEX encoded String.

```
byte[] be.atbash.util.codec.Hex.decode(String)
```

Converts the Hex representation to a byte array. Throws an *IllegalArgumentException* when the value isn't a HEX encoded value.

BASE32

Converts to and from Base32 encoded Strings.

```
String be.atbash.util.codec.Base32Codec.encodeToString(byte[])
```

Converts a byte array to a Base32 encoded String.

```
byte[] be.atbash.util.codec.Base32Codec.decode(String)
```

Converts the Base32 representation to a byte array. Throws an *IllegalArgumentException* when the value isn't a Base32 encoded value.

ByteSource

This is an interface which provides a source of bytes which can be used to create a BASE64 or HEX representation from it.

The SE Utils project provide an implementation with support for File and InputStream next to the obvious String and char array.

A ByteSource can be created by the following snippet

```
  ByteSource.creator.bytes(data);
```

Where *data* is an instance of one of the supported types as explained above.

When you need you need to support some additional types, which also can be converted to a byte array, you can use the SPI for this purpose.

Start by implementing the interface be.atbash.util.codec.ByteSourceCreator and define it as a service by specifying the fully qualified class name in the file */META-INF/services/be.atbash.util.codec.ByteSourceCreator* .

That way, your creator is used by the statement *ByteSource.creator.bytes()* and thus your logic is executed.

Your custom creator can use instances of DefaultByteSource or an implementation of the interface ByteSource.

Instantiations

The dynamic instantiation of classes is important when you define the class name within configuration values.

With the **ClassUtils** utility you can verify if the class name effectively exists and instantiate it with some arguments.

The classes, but also the resources, are searched in the following order

1. context classloader attached to the current thread
2. classloader who has loaded the ClassUtils class
3. system class loader

```
be.atbash.util.reflection.ClassUtils#isAvailable(String)
```

Verifies if the class defined by its FQCN (a fully qualified class name which is package name and class name) is found by one of the 3 class loaders.

```
be.atbash.util.reflection.ClassUtils#newInstance(String)
be.atbash.util.reflection.ClassUtils#newInstance(Class)
```

Creates an instance of the class (specified by the FQCN or the class instance) using the no-args constructor. When such a constructor is not available or there was an *Exception* thrown during the instantiation of the class, an **be.atbash.util.reflection.InstantiationException** is thrown.

```
be.atbash.util.reflection.ClassUtils#newInstance(String, Object...)
be.atbash.util.reflection.ClassUtils#newInstance(Class, Object...)
```

Creates an instance of the class (specified by the FQCN or the class instance) using a Constructor which matches the arguments.

The *Constructor* which will be used to instantiate the class is not determined by the *Class.getConstructor(argTypes)* method as it doesn't work when one of the arguments is *null*. The following algorithm is used to find the *Constructor*.

1. Loop over all *Constructor*s
2. Consider a *Constructor* when it has the same number of arguments
3. Check if the argument types have the same class (using *equals*) as the parameter type. When the argument is *null*, it is considered as a match.
4. When no *Constructor* is found, all *Constructor*s with the correct number of arguments is verified again but now a less strict match is used (using *isAssignableFrom* to allow subtypes)
5. When there is not exactly 1 Constructor found, an **be.atbash.util.reflection.NoConstructorFoundException** is thrown.

When an *Exception* is thrown during the instantiation of the class, an **be.atbash.util.reflection.InstantiationException** is thrown.

```
be.atbash.util.reflection.ClassUtils#getResourceAsStream(String)
```

Returns the resource using the 3 class loaders as described above.

Resource API (0.9.3)

On various occasions, you need to retrieve the contents of a resource. The resource can be located on the classpath, on the file system, accessible with HTTP etc ...

When you are reading some fixed resources from a certain type like classpath, then you can do it of course in a very performant way using the dedicated methods. But when you are reading some configuration values, for which the location can be changed by the developer at runtime, it might by a good idea to have some kind of API available for this.

With the class **be.atbash.util.resource.ResourceUtil** you are able to read a resource from multiple locations. In the sense that for example by using prefixes, we can indicate where the resource needs to be searched. The most obvious prefix is of course **http://** for a remote resources.

The following methods are defined on the class.

```
boolean resourceUtil.isSupported(java.lang.String);
```

Can be called to determine if the resource locator is supported by the API. Because it is extensible, see further on, it is possible to add custom location types. Method is mostly used by the internal implementation by other methods.

```
boolean resourceUtil.resourceExists(java.lang.String);
```

Determines if the resource exists and can be read.

```
InputStream resourceUtil.getStream(java.lang.String);
```

Return the *InputStream* for the resource. The method is also allowed to return null when it not able to open.

For the above methods, there exists also an overloaded variant which takes a *Object* as parameter. This is the context to which the resource location is constrained. It is not used by the default implementations, but a custom implementations can use it for retrieving resources from the ServletContext for example.

The Resource API can be accessed from the singleton **ResourceUtil** retrieved by *ResourceUtil.getInstance()*. When you also add the Atbash CDI utils, you can also inject an instance.

Resource API extensions

By default, the following implementations are supported

- ClassPath resources, with prefix *classpath:*.
- URL resources, with prefix *http:*.
- File resources, can be explicitly stated by using *file:* but not needed.

Other, custom, implementations can be created by implementing the **be.atbash.util.resource.ResourceReader** interface. The class must be registered for loading with the ServiceLoader mechanism (use file /META-INF/services/be.atbash.util.resource.ResourceReader file) and the class must have the annotation **be.atbash.util.ordered.Order** to determine the position within the list of all known readers.

Please use a positive value for your custom implementation for not interfering with the default implementations.

The interface has the following methods, corresponding to the one explained above.

```
boolean canRead(String, Object);
boolean exists(String, Object);
InputStream load(String, Object) throws IOException;
```

Be aware that the methods *exists()* and *load()* can be called also for resources which cannot be handled by the resource reader. So check the String parameter if it contains a prefix for example which indicates that the resource can be handled.

Resource Scanner (since v0.9.3)

Based on the org.reflection code, but a very limited version which can scan for resources on the classpath.

The ResourceScanner makes it possible to find all resource files within a certain directory within the classpath.

Basic usage

```
   ResourceScanner scanner = ResourceScanner.getInstance();
   Pattern pattern = Pattern.compile("someDirectory" + ".*");
   Set<String> resources = scanner.getResources(pattern)
```

The above example returns all resources (non class resources) in the *someDirectory* directory and all subdirectories.

Some important things to know

- The resources within the META-INF directory are excluded.

- JARs on the classpath are only taken into account when it contains a Manifest file (*/META-INF/MANIFEST.MF*) (Java SE only)

Another useful method in some situations are the *geResourcePaths()* methods. They return the actual location (the URL) of the resource.

If you notice that scanning of the resources takes a lot of time, you can increase the performance by supplying an instance of an *ExecutorService* so that classpath URL are scanned in a multi-threaded fashion.

You can evaluate if a multi-threaded approach is required by looking at the log entry (info level) in the format of

```
Reflections took 69 ms to scan 35 urls, producing 1843 keys and 1844 values
```

If you want to supply an *ExecutorService*, implement the **ResourceWalkerExecutorServiceProvider** and define it through the *service loader* mechanism. The interface has 1 method which needs to return the instance (but is allowed to return null)

```
ExecutorService getExecutorService();
```

By default, the *ResourceScanner* supports directories, zip and jar files and the JBoss VFS protocol. Additional types can be registered by calling the method:

```
   ResourceScanner.registerURLType(UrlType);
```

This needs to be done of course before the first call to **ResourceScanner.getInstance()** as this initializes the scanning.

Reading version

With the class **be.atbash.util.version.VersionReader**, you can read the version information stored within the *META-INF/MANIFEST.MF* file.

Define the version information by configuring the *maven-jar-plugin* or *maven-war-plugin* in the maven build section.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <version>2.5</version>
    <executions>
        <execution>
            <id>manifest</id>
            <goals>
                <goal>jar</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <archive>
            <manifestEntries>
                <Release-Version>${project.parent.version}</Release-Version>
                <buildTime>${maven.build.timestamp}</buildTime>
            </manifestEntries>
        </archive>
    </configuration>
</plugin>
```

This information can be read by using the following snippet

```
VersionReader versionReader = new versionReader("atbash-config");
versionReader.getReleaseVersion();
versionReader.getBuildTime();
```

The constructor argument is the artifact from which we want to read this information (actually it is the first part of the name of the jar file but these are in most cases the same).

Base exceptions

There are 3 exception classes defined which can be handy in all applications.

- be.atbash.util.exception.AtbashException

This is a *RuntimeException* used as a parent class for all Atbash defined exceptions. It makes it possible to define a generic Exception handler (within JSF or JAX-RS) to handle all the Exceptions uniformly (logging, showing info to end user, ...)

- be.atbash.util.exception.AtbashIllegalActionException

This exception is thrown when the Atbash code detects a wrong usage of the framework by the developer. An example is a usage of a non-existing URL filter name in the Octopus framework (maybe a typo).

It is recommended that the error message starts with a code (like *(OCT_DEV_001)*) and the

documentation describes then the situation and what actually is done wrong and how it can be fixed.

- be.atbash.util.exception.AtbashUnexpectedException

Can be used to convert a checked exception (like an IOException) into an *AtbashException* so that it can be handled by the general exception handler. Most checked exceptions never occur during the execution of the application, but they need to be caught or thrown.

String utils

```
be.atbash.util.StringUtils.hasText(String)
be.atbash.util.StringUtils.isEmpty(String)
```

Verifies if the String contains something meaning full (something different then whitespace) or not.

When the argument is *null*, empty String ("") or contains only whitespace (" ") it is considered as empty.

```
be.atbash.util.StringUtils.hasLength(String)
```

Verifies if the String contains characters or not but handles null as the empty String. Whitespace characters are counted as a real character.

```
be.atbash.util.StringUtils.clean(String)
```

Cleans the argument, this are the rules

Argument	**Result**
null	null
""	null
other cases	.trim()

```
be.atbash.util.StringUtils.startsWithIgnoreCase(String, String)
```

Verifies if the String starts with a certain prefix, case insensitive. Method handles correctly the situation where one or both arguments are *null*.

```
be.atbash.util.StringUtils.split(String)
```

Break down the String within items, delimited by , by default (there exist an overloaded method to define also the delimiter. You can use " to define the start and end of an item. The following example has thus only 2 items

```
key , "value1,value2"
```

The quotes are removed and the item is trimmed before the placed in the return array.

```
be.atbash.util.StringUtils.toDelimitedString(Collection, String)
be.atbash.util.StringUtils.toDelimitedString(Object[], String)
```

Converts the collections or array of Objects to a String where each item is separated by the 2nd parameter.

Collection utils

```
be.atbash.util.CollectionUtils.asSet(E...)
be.atbash.util.CollectionUtils.asList(E...)
```

Returns the items specified in the argument as *Set* or *List* respectively.

```
be.atbash.util.CollectionUtils.isEmpty(Collection)
be.atbash.util.CollectionUtils.isEmpty(Map)
```

Verifies if the argument is null or contains no elements.

```
be.atbash.util.CollectionUtils.size(Collection)
be.atbash.util.CollectionUtils.size(Map)
```

Returns the size of the *Collection* or *Map* but handles null argument correctly.

Proxy Utils

A few methods related to proxied classes when they are generated by (CDI) libraries.

```
be.atbash.util.ProxyUtils.isProxiedClass(Class)
```

Test if the class is a proxy class based on the name. Because proxied classes have a specific suffix.

```
be.atbash.util.ProxyUtils.getUnproxiedClass(Class)
```

Returns the 'real' class for the proxied class by returning the super class of the parameter. When the parameter isn't a proxied class, it return the parameter itself.

```
be.atbash.util.ProxyUtils.getClassName(Class)
```

Returns the 'real' class name for the parameter. When it is a proxied class, it return the name of the super class, otherwise it returns the name of the class itself.

AnnotationUtil (since v0.9.3)

With the AnnotationUtil.getAnnotation one can search if the class or one of his parent has an annotation defined on it.

```
   be.atbash.util.AnnotationUtil.getAnnotation(aClass, Annotation);
```

The difference with the *Class.getAnnotation* method is that also the object hierarchy is searched until found or the top level Object is reached.

With AnnotationUtil.findAnnotatedFields()` (Added in version 1.0.0) you can search all the fields that are annotated with the annotation specified as parameter.

```
be.atbash.util.AnnotationUtil.getAnnotation(aClass, Annotation);
```

CDICheck

Probably only useable in advanced use cases where you create a library which must be able to run within plain Java SE and within a CDI container.

```
be.atbash.util.reflection.CDICheck.withinContainer
```

This methods return true or false depending on the context and library can select code path accordingly (like retrieving beans through CDI or ServiceLoader)

TestReflectionUtil

Utility class for unit tests to help with injection and setting values of instances used during the test.

Add the following artifact to your maven project file

```
<dependency>
    <groupId>be.atbash.utils</groupId>
    <artifactId>utils-se</artifactId>
    <version>1.0.0</version>
    <classifier>tests</classifier>
    <scope>test</scope>
</dependency>
```

When you are using an instance of a class during your unit test, and that class should have some dependencies (which are normally set by some kind of injection), the *injectDependencies* can be very useful in those situations.

```
public class Foo {
```

```
private Bar bar;
```

```
}
```

Then within a test you can have the following code;

```
Foo foo = new Foo();
TestReflectionUtils.injectDependencies(foo, new Bar());
```

The injection is done based on the compatible type assignments. So you can also inject a subclass of Foo in the same manner (thus also a Mock created by Mockito for instance)

However, you should always consider the default supported functionality from Mockito for example.

```
@RunWith(MockitoJUnitRunner.class)
public class FooTest {

   @Mock
   private Bar barMock;

   @InjectMocks
   private Foo foo;

}
```

Other useful methods in the class *TestReflectionUtils*

- setFieldValue() sets the value of a specific property in an instance (when injectDependencies could inject it into multiple properties because they have assignable types)
- getValueOf() return the value of property by name (when their is no getter for instance)
- resetOf() sets the property with a null value.