

# Atbash Configuration

Rudy De Busscher

Version 1.0.0, 24/05/2022

# Table of Contents

Release Notes .....	1
1.0.0 .....	1
0.9.3 .....	1
0.9.2 .....	1
0.9.1.1 .....	1
0.9.1 .....	1
0.9 .....	2
Introduction .....	2
Getting started .....	2
Atbash configuration extension .....	2
Atbash configuration features .....	3
Define the file containing the properties .....	3
Support for multiple <i>stages</i> .....	3
Logging of configuration entries .....	4
Specify the configuration name (0.9.1) .....	5
Alternative configuration keys (0.9.3) .....	7
Advanced ClassConverter .....	7
Advanced logging features .....	8
Flexible Date format .....	9
YAML support .....	10
AbstractConfiguration (0.9.1) .....	10
Java SE support .....	10
Logging in Java SE (0.9.1) .....	10
Test module (0.9.1) .....	11
Backwards Compatibility .....	12
1.0.0 .....	12
0.9.1.1 .....	12
0.9.1 .....	12

# Release Notes

## 1.0.0

1. Compiled for and requires Java 8.
2. Dropped Java 7 port of MicroProfile API and Geronimo Configuration Implementation. (Method signature changes around Optional)
3. Updated to Java EE 8 versions
4. Using JUnit 5 for testing

Remark: If you need a Java 7 compatible version please use version 0.9.3.

## 0.9.3

1. Support for *alternative* configuration keys.

Breaking change: ResourceUtils moved to Utils SE project.

## 0.9.2

1. Defining base name with config is now optional for Config extensions (ServiceLoader config not required) Only date format can be used on default *microprofile-config.properties* file for the moment.
2. Added missing ClassConverter for TestConfig
3. Updated to MicroProfile Config 1.3 (integrated updates from Apache Geronimo Config 1.2)

## 0.9.1.1

1. Exposed `be.atbash.config.util.ResourceUtils#getInputStreamForPath()`
2. Better integration with Atbash Utils SE

Remark : ResourceUtils will be moved to Utils SE project.

## 0.9.1

1. Upgrade to MicroProfile Config 1.2 compatible (Class converter and implicit converters)
2. *atbash-config-test* artifact containing a Map based Config implementation for Unit testing.
3. *AbstractConfiguration* containing some helper methods to retrieve optional values programmatically.
4. *@ModuleConfigName* to define the name of the logging instead of the class name.

5. Logging in Java SE.
6. Parameter *atbash.config.log.disabled* to completely disable logging.
7. Logging no longer contain class name when it contains no *@ConfigEntry* entries.
8. Fix issue retrieving optional configuration parameter using 'real' MP Config implementation

## 0.9

1. Initial release, based on some extensions written for DeltaSpike logging.
2. MicroProfile Config 1.1 compatible

# Introduction

MicroProfile (MP) has done a great job in standardizing the configuration for applications.

However, there are a few improvements which I need for my applications and let me create Atbash configuration:

- Configuration for multiple *stages* within the artifact.
- Logging of configuration entries belonging to a *module*.
- Flexible date format support
- And optionally using YAML layout for configuration values to indicate relations between configuration properties.

Atbash Config is compliant with MP Config and can be used in Java SE and Java EE (only dependency on CDI) environments.

## Getting started

### Atbash configuration extension

By adding the Atbash configuration Maven artifact, you add some additional features as described in the *introduction* section.

This artifact can be used with any MP Config 1.1 compliant implementation.

*Atbash extension*

```
<dependency>
  <groupId>be.atbash.config</groupId>
  <artifactId>atbash-config</artifactId>
  <version>${atbash.config.version}</version>
</dependency>
```

The list of features is described in the *Atbash configuration features* section.

# Atbash configuration features

## Define the file containing the properties

In the MicroProfile Configuration specification, the file name which contains the configuration values is fixed and defined as **META-INF/microprofile-config.properties**.

However, Atbash configuration should also be available within applications which aren't categorized as micro-services, but general Java EE application (perhaps a Web application using JSF framework)

Therefore the *base* name of the configuration file is specified by implementing the **be.atbash.config.spi.BaseConfigurationName** interface and defining this class for usage with the ServiceLoader mechanism of Java.

*Defining the base name of the configuration file*

```
public class DemoBaseName implements BaseConfigurationName {  
    @Override  
    public String getBase() {  
        return "demo";  
    }  
}
```

Define classname for ServiceLoader mechanism within *src/main/resources/META-INF/services/be.atbash.config.spi.BaseConfigurationName*

```
be.atbash.config.examples.se.DemoBaseName
```

In the above example, the file **demo.properties** (but also demo.yaml; see further on) on the classpath is used as configuration source.

Multiple classes implementing the interface (and specified within the ServiceLoader file) are supported.

## Support for multiple stages

Everyone agrees that your artifact (thin war or fat jar) shouldn't be changed between the different stages like *Testing*, *Acceptance* and *Production*.

Most people achieve this by externalizing the configuration properties which changes in the different environment and specifies them as environment properties or System Properties.

But it is better that all configuration values of your application are also under version control, just as your code.

Therefore your artifact could contain the following files (on the classpath)

- `demo.properties` → Configuration properties which do not change between the different environments and/or default values for those properties that do change.
- `demo-test.properties` → Configuration property values for the test environment/stage
- `demo-production.properties` → Configuration property values for the production environment/stage

If the application is started with the stage *test*, the files *demo.properties* and *demo-test.properties*. When configuration properties are defined in both files, the one in the stage-specific file (*demo-test.properties* in the example) has priority.

In fact, Atbash adds 3 levels to the *ConfigSources* defined with the Configuration spec.

Priority	ConfigSource
400	JVM System properties based ConfigSource (From MP Config)
300	System environment properties based ConfigSource (From MP Config)
250	Configuration file(see remark) to overrule application property, specified by -Ds JVM System Property.
200	Stage/environment specified file (classpath only), specified by -DS JVM System property.
150	'Default' configuration file for application defined by <i>base</i> name.
100	microprofile-config.properties file based ConfigSource (From MP Config)

Remark: There are 3 prefixes supported to specify the location type of the configuration file, **classpath:**, **file:** and **url:**.

This feature is modeled based on WildFly Swarm configuration principles.

## Some examples

TODO

## Logging of configuration entries

Atbash configuration will also be used in the rewrite of the Octopus security framework. There we have several modules which each have their separate configuration values and they are logged at startup of the application.

But also in general, it can be handy to have a list within the logs of all the configuration values which are used.

This can be achieved by using the **ModuleConfig** marker interface, as shown in the example.

*ModuleConfigs which are logged during application startup.*

```
@ApplicationScoped
public class ApplicationConfiguration implements ModuleConfig {

    @Inject
    private Config config;

    @Inject
    @ConfigProperty(name = "value1")
    private String value1;

    @ConfigEntry
    public String getValue1() {
        return config.getValue("value1", String.class);
    }

    @ConfigEntry
    public Integer getValue2() {
        return config.getValue("value2", Integer.class);
    }
}
```

```
INFO [be.atbash.config.logging.StartupLogging] (ServerService Thread Pool -- 22)
Config implementation: be.atbash.config.examples.ee.ApplicationConfiguration
  method:    getValue2
  value:     500

  method:    getValue1
  value:     Stage based Value
```

Of course, this feature only works in CDI based environment and is triggered by the initialization event linked to the CDI `@ApplicationScoped`.

Since 0.9.1, the logging can be disabled by specifying the value *true* for the configuration parameter **atbash.config.log.disabled**. This is a regular parameter, resolved from configuration file, or environment based on the rules described in the section of the multiple stages.

So we can for instance disable the logging completely in production, but not in test for example.

## Specify the configuration name (0.9.1)

With the `@ModuleConfigName`, we can define the name which is showed above the config values within the log.

For this example, we have the following 2 classes

*Parents class.*

```
public class ParentConfig {

    @ConfigEntry
    public String defineParentValue() {
        return "Parent Config Value";
    }
}
```

Childs class.

```
@ApplicationScoped
public class ChildConfig extends ParentConfig implements ModuleConfig {

    @ConfigEntry
    public String defineChildValue() {
        return "Child Config Value";
    }

}
```

Depending on what we define on the ChildConfig class, we have a slightly different output.

```
@ModuleConfigName("Module Config Name example")

INFO [be.atbash.config.logging.StartupLogging] (ServerService Thread Pool -- 66)
Module Config Name example :
  method: defineChildValue
  value:   Child Config Value

  method: defineParentValue
  value:   Parent Config Value
```

```
@ModuleConfigName(value = "Module Config with classes", className = true)

INFO [be.atbash.config.logging.StartupLogging] (ServerService Thread Pool -- 24)
Config implementation: Module Config with classes (
be.atbash.config.examples.ee.configname.ChildConfig )
  method: defineChildValue
  value:   Child Config Value

Config implementation: Module Config with classes (
be.atbash.config.examples.ee.configname.ParentConfig )
  method: defineParentValue
  value:   Parent Config Value
```



```
without @ModuleConfigName
```

```
INFO [be.atbash.config.logging.StartupLogging] (ServerService Thread Pool -- 63)
Config implementation: be.atbash.config.examples.ee.configname.ChildConfig
  method: defineChildValue
  value:   Child Config Value
```

```
Config implementation: be.atbash.config.examples.ee.configname.ParentConfig
  method: defineParentValue
  value:   Parent Config Value
```

## Alternative configuration keys (0.9.3)

The support for alternative key values is introduced for supporting functionality to change configuration key values in a backwards compatible way.

Suppose you have a configuration key `loginPage` but in the new version of your tool, you like to change this to `atbash.octopus.web.loginPage`. And you would like to support also the old configuration key for those developers who didn't update their configuration file yet.

Within your code, you can already use the new configuration key, `atbash.octopus.web.loginPage` in our example, and define the mapping in a file like `config/alias-demo.properties`.

```
atbash.octopus.web.loginPage=loginPage
```

The special ConfigSource created for this scenario comes always last in the list of all ConfigSources. It will check if the key is one of the defined keys for which an alternative is defined and tries to read the value for this alternative value.

The alternative key names needs to be defined in a classpath file with name and location like `config/alias-<someName>.properties` where someName can be anything but should be unique (no multiple resources with the same name should be found)

When the ConfigSource detects that a value for the `oldKey` is used, an INFO message is written to the log to inform the developers.

## Advanced ClassConverter

Atbash Configuration Class converter uses the advanced Class loading features of Atbash SE.

So the following example

```
Class<?> aClass = config.getValue("someClassName", Class.class);
```

or

```
@Inject  
@ConfigProperty(name = "someClassName")  
private Class<?> aClass;
```

will search for the Fully Qualified Class name with

1. Current thread Class Loader
2. ClassLoader containing the class `be.atbash.util.reflection.ClassUtils`
3. System class loader

## Advanced logging features

### @ConfigEntry(noLogging)

The config value can contain sensitive information so it is not always desirable to have this value in the log. By specifying the member *noLogging* one can indicate that the configuration value will not be logged, only if it value is specified (non null) or not.

*Not logging sensitive information*

```
@ConfigEntry(nologging = true)  
public String getSecretValue() {return "secret";}
```

The above configuration parameter will then be shown as follow in the log file.

```
method: getSecretValue  
value: No logging parameter active [non null value]
```

You can overrule this hiding of configuration value by defining the JVM system value **atbash.config.log.all** (like in `-Datbash.config.log.all=true`) and the value will be shown in the log.

### @ConfigEntry(value)

There are various use cases where it doesn't make sense to show the configuration parameter value.

1. There are some cases that the exact value of the configuration parameter is only known after the application is fully deployed. Or that the value is based on some method calls which aren't available during the logging of the parameters (like calculated URLs of the deployed Web applications)

When we define a value for the member **value**, this value is shown instead of executing the

method. The text you can place there is anything you like but should be informative why it is not the real value.

2. A second use-case, although even more rare, is that the method has a parameter (after all those methods which provide configuration values are regular methods)

In this case it is impossible for the code to know what the parameter should be. With the usage of the *value* member, we can put some info into the log, otherwise following message is shown.

```
method: methodNameWithParameter
value:  unknown - Method has a parameter
```

## Dynamic values

TODO

## Flexible Date format

The converters for the **Date.class** use a default format, but the default format is Locale dependent.

This means that when the application runs on multiple servers where, for whichever reason, the Locale information of the OS is not identical, the parsing of the dates can fail.

Therefore, support is foreseen to define the Date pattern within the configuration file itself, as a configuration value.

```
atbash.date.pattern:dd-MM-yyyy
```

or in YAML format

```
atbash :
  date :
    pattern : dd-MM-yyyy
```



This date format will be used for all Date values within all configuration files, not only the file where the pattern is defined.

Being global can have some nasty unwanted effects when you include artifacts from other developers containing also configuration files but specifying Date values in another format.

Therefore, the format can be specified for each Date value separately as follows

```
dateValue : 16-11-2017,dd-MM-yyyy
```

# YAML support

TODO

## AbstractConfiguration (0.9.1)

An abstract class which can be used to retrieve optional configuration values with or without a default value.

Using the CDI Qualifier `@ConfigProperty`, this can already be achieved, but not in a programmatic way.

```
protected <T> T getOptionalValue(String propertyName, T defaultValue, Class<T>
propertyType) {
```

## Java SE support

Since the core of MicroProfile Configuration is created around the *ServiceLoader* principal of Java SE, it can also be used within Command Line programs for example.

*Using Configuration with plain Java SE programs*

```
Config config = ConfigProvider.getConfig();
config.getValue("value1", String.class);
```

Next to the basic functionality of MP Configuration (like converts), following Atbash extension features are also available

- Configuration for multiple *stages* within the artifact.
- Flexible date format support.
- YAML layout for configuration values.

## Logging in Java SE (0.9.1)

All CDI beans implementing the *ModuleConfig* marker interface containing methods specifying some configuration values (with `@ConfigEntry`) are logged automatically during startup of the application.

This is not possible in Java SE (unless using CDI 2.0) but the developer can decide to manually log the configuration values.

The class must only implement the marker *ModuleConfig* interface and with the following command the values are logged.

```
ModuleConfiguration moduleConfiguration = new ModuleConfiguration();
StartupLogging.logConfiguration(moduleConfiguration);
```

## Test module (0.9.1)

When other projects are using the *Atbash config* resources, for their configuration through parameter values, tests are probably failing with the message.

```
Caused by: java.lang.IllegalStateException: No ConfigProviderResolver implementation
found!
    at
    org.eclipse.microprofile.config.spi.ConfigProviderResolver.instance(ConfigProviderReso
    lver.java:121)
```

This is because *Atbash-config* is only bundled with the API and not any implementation (so that it can be used with any Eclipse MicroProfile Config implementation)

For testing, a simple implementation based on a HashMap which can be filled according to the needs of the test, is available in the *Atbach Config test* artifact.

```
<dependency>
  <groupId>be.atbash.config</groupId>
  <artifactId>atbash-config-test</artifactId>
  <version>${atbash.config.version}</version>
  <scope>test</scope>
</dependency>
```

How to use this module within your unit tests?

Use the **addConfigValue()** method to define some values for Configuration parameters.

```
TestConfig.addConfigValue("someConfig", "configValue");
```

When the test is finished (for example within the *@After* annotated method with JUnit), don't forget to reset the configuration source so that the values don't influence other tests.

```
TestConfig.resetConfig();
```

Probably you want the default converters in place so that you can retrieve the configuration values as String, Boolean, Long, Float, Double, Integer or Date instance. This can be achieved by executing the **registerDefaultConverters()** method.

```
TestConfig.registerDefaultConverters();
```

Additional converters can be registered by the **registerConverter(Converter<?>)** method. And all the converters are removed by the **deregisterAllConverters()** method.

# Backwards Compatibility

## 1.0.0

Dropped the Java 7 support and introduced the usage of Optional. This will lead to some changes in methods and behaviour since MicroProfile Config is now used as is.

## 0.9.1.1

No issues on classes marked as `@PublicAPI`.

Internal ProxyUtils class moved to utils-se.

## 0.9.1

No blocking issues except you created a custom ConfigBuilder or custom ConfigSource

(See also pom.xml for the CLIRR analysis)

- ConfigBuilder interface has new method withConverter()
- ConfigSource interface has additional method getPropertyNames().